TECHNICAL WHITEPAPER

Decred

A hybrid proof-of-work and proof-of-stake cryptocurrency with on-chain consensus governance and a self-funding treasury

ABSTRACT

Decred is a cryptocurrency whose consensus rules are enforced, and amended, by the people who hold it. Blocks are produced by proof-of-work miners, but a block only becomes part of the chain once a pseudorandomly selected quorum of stakeholders, who have time-locked funds to buy *tickets*, has voted to approve it. The same tickets vote on changes to the consensus rules themselves and on every expenditure from a treasury that is funded by a fixed share of each block reward.

This paper specifies the complete protocol as deployed on the Decred main network: the cryptographic primitives, the serialization of every consensus-critical structure, the script language, the transaction and block validation rules, the stake system and its ticket lottery, both difficulty adjustment algorithms, the issuance schedule, the decentralized treasury, the on-chain voting mechanism for rule changes together with every change activated to date, and the peer-to-peer protocol. It is written so that an independent implementation built from this document alone accepts and rejects exactly the same blocks as the reference implementation.

Document version	1.0
Date	16 November 2026
Reference implementation	dcrd, commit `46e9df48109a` (master, 5 November 2026)

Contents

1 Introduction **2**
- 1.1 Motivation . . . 2
- 1.2 The System at a Glance . . . 2
- 1.3 The Life of a Block . . . 3
- 1.4 Tickets . . . 4
- 1.5 Issuance . . . 4
- 1.6 Governance . . . 4
- 1.7 Design Properties . . . 5
- 1.8 Scope and Structure of this Document . . . 5

2 Conventions and Notation **7**
- 2.1 Normative Language . . . 7
- 2.2 The Reference Implementation . . . 8
- 2.3 Integers and Arithmetic . . . 8
- 2.4 Bytes, Byte Order and Hashes . . . 9
- 2.5 Amounts . . . 9
- 2.6 Heights, Time and Intervals . . . 9
- 2.7 Pseudocode . . . 9
- 2.8 Consensus Votes and Rule Activation . . . 10
- 2.9 Test Vectors . . . 10

3 Cryptographic Primitives **12**
- 3.1 Notation . . . 12
- 3.2 BLAKE-256 . . . 13
- 3.3 Hash Functions Derived from BLAKE-256 . . . 16
- 3.4 BLAKE3 . . . 16
- 3.5 SHA-256, SHA-1 and RIPEMD-160 . . . 18
- 3.6 The secp256k1 Curve . . . 18
- 3.7 Signature Schemes in Consensus . . . 20
- 3.8 ECDSA . . . 21
- 3.9 EC-Schnorr-DCRv0 . . . 25
- 3.10 Ed25519 . . . 27
- 3.11 Merkle Trees . . . 29
- 3.12 Other Primitives . . . 31

4 Data Encoding and Structures **33**
- 4.1 Decoding Principles . . . 33
- 4.2 Primitive Encodings . . . 33
- 4.3 Transactions . . . 35
- 4.4 The Block Header . . . 43
- 4.5 Transaction Trees and Merkle Roots . . . 44
- 4.6 Block Serialization . . . 46
- 4.7 Amounts . . . 47
- 4.8 Summary of Constants and Decoding Limits . . . 48
- 4.9 Test Vectors . . . 48

5 Keys and Addresses **53**

5.1 Overview ... 53
5.2 Base58 and Base58Check ... 54
5.3 Network Prefixes ... 55
5.4 Address Types ... 56
5.5 Decoding Addresses ... 61
5.6 Private Keys: Wallet Import Format ... 62
5.7 Signed Messages ... 63
5.8 Hierarchical Deterministic Keys ... 64
5.9 Amounts ... 69
5.10 Bech32 ... 69

6 The Script System 71
6.1 Overview and Terminology ... 71
6.2 Script Versions ... 72
6.3 Parsing ... 72
6.4 Limits ... 73
6.5 Engine Construction ... 74
6.6 Execution Model ... 75
6.7 Data Representation ... 79
6.8 Opcode Semantics ... 80
6.9 Signature Verification ... 87
6.10 Signature Hash ... 91
6.11 Pay-to-Script-Hash ... 93
6.12 Stake and Treasury Tags ... 94
6.13 Verification Flags ... 95
6.14 Signature Operation Counting ... 96
6.15 Other Consensus Predicates ... 97
6.16 Standard Script Classes ... 98
6.17 Test Vectors ... 99
6.18 Error Kinds ... 106

7 Transactions 107
7.1 Overview ... 107
7.2 The UTXO Model ... 108
7.3 Classification ... 110
7.4 Context-Free Sanity Rules ... 112
7.5 Contextual Structure Rules ... 114
7.6 Expiry ... 117
7.7 Lock Time Finality ... 117
7.8 Relative Lock Times (Sequence Locks) ... 118
7.9 Input Validation ... 121
7.10 Connecting Transactions and In-Block Spending ... 126
7.11 Signature Operations ... 127
7.12 Script Validation ... 128
7.13 Error Kinds ... 130

8 The Stake System 132
8.1 Overview ... 132
8.2 Classification and Common Script Forms ... 134
8.3 Ticket Purchase Transactions ... 136
8.4 Vote Transactions ... 138
8.5 Revocation Transactions ... 141
8.6 Input and Amount Validation ... 142
8.7 The Lottery ... 148
8.8 The Ticket Database ... 152
8.9 Block-Level Stake Rules ... 155
8.10 Vote Bits and Vote Version ... 158

8.11 Non-Consensus Behavior 159
8.12 Error Kinds (informative) 159

9 Proof of Work and Difficulty 162
9.1 Overview and Selection of Algorithms 162
9.2 The Proof-of-Work Check 164
9.3 The Compact Target Encoding 165
9.4 Work and Chain Work 167
9.5 Required Difficulty Bits 168
9.6 The EMA Difficulty Algorithm 169
9.7 ASERT 172
9.8 Test Network Rules 175
9.9 Stake Difficulty 176
9.10 Summary of Header Rules 180

10 Blocks and Chain Selection 182
10.1 Overview 182
10.2 Header Rules 184
10.3 Block Body Rules 189
10.4 Header Commitments 194
10.5 Stakeholder Approval of the Previous Regular Tree 194
10.6 Connecting a Block 198
10.7 Chain Selection 204
10.8 Constants and Error Kinds 209

11 Issuance and Block Subsidy 212
11.1 Overview 212
11.2 The Full Block Subsidy 214
11.3 Splitting the Subsidy 215
11.4 The Coinbase Transaction 218
11.5 Subsidy Checks in the Stake Tree 223
11.6 The Block-One Ledger 224
11.7 Total Supply 225
11.8 Summary of Checks 227

12 The Decentralized Treasury 229
12.1 Background and Overview 229
12.2 Parameters and Constants 231
12.3 Classification of Treasury Transactions 232
12.4 The Treasury Balance 233
12.5 Treasurybase 234
12.6 Treasury Add 236
12.7 Treasury Spend 238
12.8 Treasury Spend Voting 242
12.9 Expenditure Policy 246
12.10 Interactions with Other Rules 250
12.11 Error Kinds 251

13 Consensus Rule Changes 253
13.1 Overview 253
13.2 Interval Arithmetic 255
13.3 Block Versions 256
13.4 Votes, Vote Versions and Vote Bits 257
13.5 Stake Version 259
13.6 The Threshold State Machine 261
13.7 Agenda Activation 265
13.8 Explicit Version Upgrades (DCP-0008) 268
13.9 Deployments 268

13.10 Main-Network History 270
13.11 Deriving Agenda Status in an Implementation 271

14 Network Parameters **273**
14.1 The Four Networks 273
14.2 Reading the Parameter Tables 274
14.3 Parameter Tables 274
14.4 Derived Quantities 284
14.5 Network-Specific Rules Outside the Parameters 285
14.6 Genesis Blocks 286
14.7 Block-One Ledger 290

15 The Peer-to-Peer Protocol **292**
15.1 Normative Status 292
15.2 Transport 292
15.3 The Message Envelope 293
15.4 Common Encodings 295
15.5 Protocol Versions and Services 298
15.6 The Connection Handshake 299
15.7 Message Reference 303
15.8 Synchronization and Relay 311
15.9 Peer Management 316

16 Compact Filters and Light Clients **321**
16.1 Overview and Constants 321
16.2 SipHash-2-4 322
16.3 Golomb-Coded Sets 323
16.4 Block Filter Contents 327
16.5 Header Commitments 331
16.6 Light Clients 333
16.7 Test Vectors 334

17 Relay Policy and Mining **338**
17.1 Pools and Transaction Classes 338
17.2 Mempool Acceptance 339
17.3 Stake Transaction Policy 345
17.4 Mempool Maintenance 347
17.5 Block Template Construction 348
17.6 Background Template Generation 353
17.7 Mining Interfaces 354
17.8 Fee Estimation 357
17.9 Policy Constants 357

18 StakeShuffle Mixing **360**
18.1 Overview 360
18.2 Primitives and Notation 361
18.3 Identities, Hashes and Signatures 363
18.4 Wire Messages 364
18.5 Pair Requests 368
18.6 Epochs and Session Formation 370
18.7 The Protocol Run 373
18.8 Blame and Failure Handling 378
18.9 The Mixpool: Relay Policy 379
18.10 Constants 382
18.11 Test Vectors 382

19 Governance and Ecosystem **386**
19.1 The Governance Model 386
19.2 Politeia 387

19.3 Voting Service Providers 390
19.4 Wallets and Light Clients 391
19.5 Payment Channels and the Lightning Network 392
19.6 Atomic Swaps and DCRDEX 393
19.7 Timestamping with dcrtime 394
19.8 Dependence on the Consensus Protocol 395

A Opcode Reference **397**
A.1 Conventions 397
A.2 Map of Byte Values 398
A.3 Opcode Table 398
A.4 Remarks 401

B Index of Consensus Rules **403**

C Differences from Bitcoin **407**
C.1 Consensus Model 407
C.2 Hashes and Identifiers 407
C.3 Transactions and Blocks 408
C.4 Script 408
C.5 Network and Wallets 409

References **410**

PART I

Foundations

What Decred is, how this document is written, and the primitives every later chapter relies on: hash functions, signature schemes, serialization and addresses.

CHAPTER 1

Introduction

Decred is a cryptocurrency that has been in continuous operation since its genesis block of 8 February 2016. It descends from Bitcoin [1], from which it inherits the unspent-output ledger, the script-guarded transaction model, and a chain of blocks secured by proof of work. Its distinguishing feature is who has the final say. In Decred that is the holders of the currency, acting through an on-chain mechanism that lets them approve or reject each block that miners produce, vote on changes to the consensus rules, and authorize every payment from a common treasury.

This chapter is informative. It describes the system as a whole so that the normative chapters that follow can be read in context, and it ends with a guide to the rest of the document.

1.1 Motivation

A pure proof-of-work currency places two distinct powers in the same hands. Miners order transactions, which is the job proof of work was designed for, but they also hold an effective veto over changes to the rules, because a rule change only takes hold if the blocks that follow it are mined. Those who hold the currency and bear the consequences of a rule change have no formal voice. In practice, contested changes are settled by informal negotiation between developers, miners, exchanges and users, with no agreed procedure and no agreed way of telling when a decision has been made.

A second difficulty is funding. A decentralized currency has no owner to pay for its development. Projects have relied on donations, on companies whose interests may diverge from the network's, or on a founders' allocation that is eventually spent.

Decred addresses both with one construction. Earlier proposals had shown how proof of stake could be layered on proof of work so that stakeholders participate in block production [2], [3], and Decred's original design [4] adopted a scheme of that kind. Coins are time-locked in exchange for *tickets*; tickets are called at random to vote; and a block is only valid if a majority of the tickets called for it have voted. Once every block carries stakeholder votes, the same votes can carry other decisions, and Decred uses them for three things:

1. **Block approval.** Each vote states whether the previous block's ordinary transactions should stand. A block that is voted down keeps its place in the chain, but its ordinary transactions are undone and its miner is not paid.
2. **Consensus rule changes.** New rules are deployed in software in a dormant state and take effect only if a supermajority of votes over a fixed period approve them (Chapter 13).
3. **Treasury expenditure.** A fixed share of every block reward accrues to a treasury. Since the activation of DCP-0006 the treasury exists only as a balance in the consensus state and can be spent only by a transaction that a supermajority of votes has approved (Chapter 12); before that, the share was paid to a fixed script controlled by the project's developers.

1.2 The System at a Glance

Four roles interact through the protocol.

Miners assemble transactions and votes into blocks and expend computation to find a block header whose hash meets the proof-of-work target (Chapter 9). The hash function has been BLAKE3 since a 2023 rule change; before that it was BLAKE-256.

Stakeholders lock funds to buy tickets, and vote when their tickets are called (Chapter 8). A ticket holder may delegate the act of voting to an always-online service without giving up custody of the funds (Chapter 19).

Full nodes verify every block against every rule in this document and relay valid blocks and transactions to their peers (Chapter 15). A full node trusts no one: it accepts only what it has verified.

Light clients verify block headers only and use compact filters, to which every header since DCP-0005 commits, to find the transactions that concern them without revealing which those are (Chapter 16).

The ledger itself is a set of unspent transaction outputs. Each output holds an amount and a script; a transaction consumes existing outputs by satisfying their scripts and creates new ones (Chapter 6, Chapter 7). Every block contains two separate lists of transactions. The **regular tree** holds ordinary payments and the miner's reward. The **stake tree** holds ticket purchases, votes, ticket revocations and treasury transactions. The split exists because the two are treated differently when a block is voted down: the stake tree always stands, while the regular tree is subject to the approval of the next block's voters.

1.3 The Life of a Block

The target interval between blocks on the main network is five minutes. Figure 1.1 shows how proof of work and proof of stake alternate.

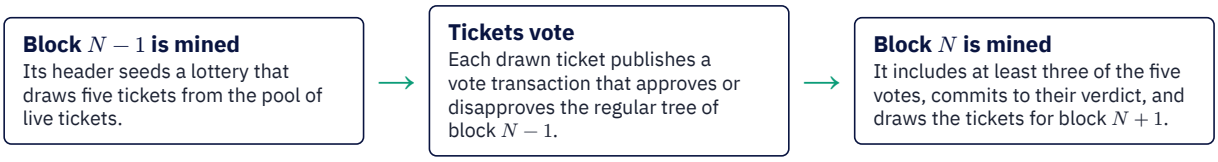


Figure 1.1 One step of the chain. Block N cannot be built until a majority of the tickets drawn by block $N-1$ have voted on block $N-1$.

In more detail:

1. When a block is connected to the chain, a deterministic pseudorandom generator seeded with the block's header selects `TicketsPerBlock` $= 5$ winning tickets from the pool of live tickets. Every node computes the same winners.
2. The holders of the winning tickets validate the block and broadcast a vote for it. A vote is a transaction that spends the ticket. It names the block being voted on, carries a bit approving or disapproving that block's regular tree, carries the holder's choices on any rule changes being voted, and pays the holder back the ticket price plus a share of the block reward.
3. A miner collects the votes and builds the next block on top of the block voted on. The new block is valid only if it contains votes from more than half of the winning tickets, that is at least three. The miner's own reward shrinks in proportion to any votes left out, so miners have reason to include all five.
4. Unless more of the included votes approve than disapprove, the previous block's regular tree is treated as though it had never been mined: its transactions return to the pool of unconfirmed transactions and its miner's reward is never created. The chain nevertheless continues on top of it.

The effect is that miners cannot extend the chain alone. A miner with a large share of the hash power who withholds blocks, mines empty blocks, or censors transactions can be denied the reward by the stakeholders, and a chain built in secret cannot be extended without the cooperation of tickets that its author does not control. Conversely, stakeholders cannot produce blocks without proof of work.

1.4 Tickets

A ticket is bought by a transaction that locks the **ticket price** in an output that only a vote or a revocation can spend. The minimum price is the same for every ticket bought in a given block and is recomputed every `StakeDiffWindowSize` = 144 blocks by an algorithm that steers the number of live tickets toward a target of `TicketPoolSize` × `TicketsPerBlock` = 40960 (Chapter 9). At most `MaxFreshStakePerBlock` = 20 tickets may be bought per block.

A ticket then passes through the following stages.

Immature for `TicketMaturity` = 256 blocks after purchase the ticket cannot be drawn.

Live the ticket is in the pool and is drawn with equal probability to every other live ticket. With the pool at its target size a ticket waits 8192 blocks, about 28 days, on average.

Voted when drawn, the ticket votes; the price is returned together with the reward.

Missed if the ticket is drawn and its vote is not included in a block, it loses the reward for that drawing and can never vote.

Expired a ticket that has not been drawn within `TicketExpiry` = 40960 blocks, about 142 days, leaves the pool. With the pool at target this happens to fewer than one ticket in a hundred.

Revoked a missed or expired ticket is revoked, returning the ticket price to its buyer without a reward. Since 2022 revocations are created automatically by the block that causes them.

Because the funds are locked for an unpredictable period, and are returned without reward if the holder fails to vote, tickets represent a real and recurring commitment to the network by those who hold its currency.

1.5 Issuance

New coins are created by each block. The reward began at 31.19582664 DCR per block and is multiplied by 100/101 every `SubsidyReductionInterval` = 6144 blocks, about every 21 days, which produces a smooth decline in place of Bitcoin's abrupt halvings. Total issuance converges on approximately 21 million DCR. The first block after genesis additionally created a fixed allocation that was divided between the original developers and an airdrop to early participants (Chapter 11).

Each block's reward is divided between the miner, the voters and the treasury. The division has been changed twice by stakeholder vote, as Table 1.1 shows.

Table 1.1 Division of the block reward

Period	Proof of work	Proof of stake	Treasury
From launch	60%	30%	10%
After DCP-0010 (2022)	10%	80%	10%
After DCP-0012 (2023)	1%	89%	10%

1.6 Governance

The consensus rules of a blockchain are whatever its full nodes enforce, so changing them requires that nodes change in step. Decred makes that coordination explicit and puts the decision to a vote.

A proposed change is specified in a Decred Change Proposal and implemented in the node software, where it lies dormant. Once enough miners and voters have upgraded, the change is put to the tickets as an *agenda*. Voting runs in fixed intervals of `RuleChangeActivationInterval` = 8064 blocks, about four weeks. If, in one interval, the yes votes are at least 75% of the votes that expressed a preference (in the integer arithmetic of Chapter 13), and enough such votes were cast to meet quorum, the change is locked in; it activates exactly one interval later, giving the remaining nodes four weeks to upgrade. The state of every agenda is a function of the chain alone, so all nodes agree on precisely which rules apply to each block. Chapter 13 specifies the mechanism.

Table 1.2 lists the rule changes adopted in this way. Each is described in full in the chapter concerned, and because a node must validate the whole chain from its first block, each remains part of the specification in both its before and after forms.

Table 1.2 Consensus changes adopted by stakeholder vote

Proposal	Agenda	Change
DCP-0001	`sdiffalgorithm`	New ticket price algorithm
DCP-0002, DCP-0003	`lnfeatures`	`OP_SHA256` and relative lock times, in support of payment channels
DCP-0004	`fixlnseqlocks`	Correction to the evaluation of relative lock times
DCP-0005	`headercommitments`	Block headers commit to compact filters for light clients
DCP-0006	`treasury`	Decentralized treasury controlled by stakeholder vote
DCP-0007	`reverttreasurypolicy`	Revised limit on treasury expenditure
DCP-0008	`explicitverupgrades`	New transaction and script versions must be enabled by vote
DCP-0009	`autorevocations`	Automatic revocation of missed and expired tickets
DCP-0010	`changesubsidysplit`	Block reward division changed to 10/80/10
DCP-0011	`blake3pow`	Proof of work changed to BLAKE3; new difficulty algorithm
DCP-0012	`changesubsidysplitr2`	Block reward division changed to 1/89/10
DCP-0013	`maxtreasuryspend`	New limit on treasury expenditure

Treasury spending follows a parallel path. A treasury spend transaction is published in advance and voted on by the tickets over a window of blocks; it is valid in a block only if it gathered the required quorum and at least 60% approval, and only if it keeps total expenditure within a limit set by the consensus rules (Chapter 12). What to fund is debated off-chain in a proposal system described in Chapter 19.

1.7 Design Properties

Several properties of the design are worth stating before the detail.

Transaction identifiers cannot be altered by third parties a transaction's identifier is computed over its inputs' references and its outputs, and excludes the signatures (Chapter 4). Protocols that build chains of unconfirmed transactions, such as payment channels, rely on this.

Signatures are not limited to one scheme scripts can verify ECDSA and Schnorr signatures over secp256k1 and Ed25519 signatures (Chapter 3, Chapter 6).

Scripts are versioned every output carries a script version, so new script languages can be introduced without disturbing existing outputs.

Transactions can expire a transaction may name a height from which it can no longer be mined, so its fate is settled within a known time.

Headers are self-describing each header records the ticket pool size, the number of votes, tickets and revocations in the block, the ticket price, and the outcome of the lottery, which lets header-only clients follow the stake system.

Readers who know Bitcoin's protocol will find a summary of the differences in Appendix C.

1.8 Scope and Structure of this Document

The document specifies the protocol: what is valid, and how nodes communicate. It does not describe how the reference implementation is organized, how it stores the chain on disk, or its remote procedure call interface, none of which another implementation need share.

- **Part I** lays the foundations: the conventions used throughout (Chapter 2), the cryptographic primitives (Chapter 3), the serialization of transactions and blocks (Chapter 4), and keys and addresses (Chapter 5).
- **Part II** specifies the script language (Chapter 6) and the rules for individual transactions (Chapter 7).

- **Part III** specifies consensus: tickets and votes (Chapter 8), proof of work and both difficulty algorithms (Chapter 9), block validation and chain selection (Chapter 10), issuance (Chapter 11), the treasury (Chapter 12), rule changes (Chapter 13), and the parameters of each network (Chapter 14).
- **Part IV** covers the network: the peer-to-peer protocol (Chapter 15), compact filters and light clients (Chapter 16), relay and mining policy (Chapter 17), coin mixing (Chapter 18), and the systems built around the chain (Chapter 19).
- The **appendices** give the complete opcode table (Appendix A), an index of every consensus rule (Appendix B), and a comparison with Bitcoin (Appendix C).

Consensus rules are confined to Parts II and III and to the parts of Chapter 3, Chapter 4 and Chapter 16 on which they depend. An implementation that follows those chapters exactly will agree with the network on every block. The remaining chapters describe conventions that a node must follow to be useful to its peers and its users, but on which the validity of the chain does not depend.

CHAPTER 2

Conventions and Notation

This chapter fixes the vocabulary and notation used in the rest of the document. It is short, but several of its conventions, in particular the treatment of integer arithmetic and of hash byte order, are the source of most consensus failures in independent implementations and are worth reading closely.

2.1 Normative Language

The key words **MUST**, **MUST NOT**, **SHOULD**, **SHOULD NOT** and **MAY** are to be interpreted as described in RFC 2119 [5]. They are set in semibold sans-serif capitals whenever they carry that meaning.

Requirements come in two kinds, and the distinction matters:

- A **consensus rule** determines whether a block, and therefore everything in it, is valid. Every full node must reach the same verdict on every block, so a consensus rule has exactly one correct implementation. Consensus rules are set apart in numbered boxes, like Rule 2.1, and are listed together in Appendix B.
- A **policy rule** governs what a node chooses to relay, store in its memory pool, or include in the blocks it creates. Nodes are free to differ on policy without splitting the network. Policy is described in ordinary prose and in Chapter 17.

RULE 2.1 Example of a consensus rule
A rule box states one requirement completely, using **MUST** or **MUST NOT**. The line at the bottom right names the place in the reference implementation that enforces it, as a path within the dcrd source tree followed by the function.

`dcrd: internal/blockchain/validate.go · checkBlockHeaderSanity`

Besides rule boxes, four kinds of aside appear throughout the text.

NOTE
A note adds informative detail. Nothing in a note is required for consensus.

RATIONALE
A rationale explains why a rule exists or why it takes the form it does.

HISTORY
A history box describes how a rule changed over the life of the chain. Because a node must validate every block since the genesis block, historical behaviour is as binding as current behaviour: where a rule changed at some height or through a vote, both forms are specified.

IMPLEMENTATION PITFALL
A pitfall marks behaviour that is easy to implement almost correctly.

2.2 The Reference Implementation

Decred has no specification separate from its software. The protocol is defined by the behaviour of *dcrd* [6], the full-node implementation maintained by the Decred developers, and by the Decred Change Proposals (DCPs) [7] that document each change to the consensus rules put to a stakeholder vote. This document describes dcrd as of the commit named on the title page. Where this document and dcrd disagree, dcrd is correct and this document is in error.

Source references such as `blockchain/stake/lottery.go · findTicketIdxs` give a path relative to the root of the dcrd repository and one or more identifiers within that file. A DCP is cited as, for example, DCP-0011.

Each network that runs the protocol (the main network, the test network and two kinds of private network) is defined by a set of **network parameters**. A parameter is written in the form `TicketMaturity`, using the name of the corresponding field in dcrd's `chaincfg.Params` structure. The values for every network are tabulated in Chapter 14. Unless a passage says otherwise, concrete numbers in the text are main-network values. Constants that are the same on every network are written in the form `MaxAmount`.

2.3 Integers and Arithmetic

Consensus code is integer code. No rule in this document involves floating point arithmetic, and an implementation **MUST NOT** use floating point to evaluate any formula given here.

Table 2.1 lists the integer types used. All are two's complement.

Table 2.1 Integer types

Type	Width	Range
`uint8` , `byte`	8 bits	0 to $2^{8}-1$
`uint16`	16 bits	0 to $2^{16}-1$
`uint32`	32 bits	0 to $2^{32}-1$
`int32`	32 bits	-2^{31} to $2^{31}-1$
`uint64`	64 bits	0 to $2^{64}-1$
`int64`	64 bits	-2^{63} to $2^{63}-1$
`bigint`	unbounded	Arbitrary-precision signed integer

Three properties of fixed-width arithmetic are relied upon and must be reproduced exactly:

Wrapping Addition, subtraction and multiplication of fixed-width integers wrap modulo 2^{w}, where w is the width, unless a rule explicitly checks for overflow. Where a computation is specified on a fixed-width type, the text names the type.

Truncating division Integer division rounds toward zero, so $-7 \div 2 = -3$, and the remainder takes the sign of the dividend, so $-7 \bmod 2 = -1$. This is the behaviour of Go, C and Rust, and differs from the flooring division of Python and of mathematical convention whenever exactly one operand is negative and the division is inexact.

Arithmetic shift A right shift of a signed integer replicates the sign bit, so $-7 \gg 1 = -4$. A right shift of a `bigint` likewise rounds toward negative infinity.

In formulas, $\lfloor x \rfloor$ denotes the floor function, and $a \div b$ and $a \bmod b$ denote truncating division and its remainder as just defined. For non-negative operands the two notions of division coincide and plain fraction notation with an explicit floor is used.

2.4 Bytes, Byte Order and Hashes

A **byte string** is a finite sequence of bytes. Byte strings are written in hexadecimal, lowest index first, in a fixed-width face: `0100000000` is five bytes, the first of which is `0x01`. The concatenation of byte strings a and b is written $a \parallel b$, and the length of a in bytes is $|a|$.

Unless stated otherwise, multi-byte integers are serialized **little-endian**: least significant byte first. The functions LE_{16}, LE_{32} and LE_{64} map an integer to its 2-, 4- or 8-byte little-endian encoding, and BE_n denotes the big-endian counterparts, which occur in a small number of places that are pointed out individually.

A **hash** is a 32-byte string produced by one of the hash functions of Chapter 3. Hashes are subject to a display convention inherited from Bitcoin that must be kept firmly in mind.

> **IMPLEMENTATION PITFALL**
> When a block hash or transaction hash is shown to people (in block explorers, in RPC results, and in this document wherever it is called a hash *in display form*), its 32 bytes are written in **reverse** order. The hash whose bytes in memory and on the wire are `80d9212bf4ceb066ded2866b39d4ed89e0ab60f335c11df8e7bf85d9c35c8e29` is displayed as `298e5cc3d985bfe7f81dc135f360abe089edd4396b86d2de66b0cef42b21d980`, which is the hash of the main-network genesis block. Serialized structures always carry the bytes in their natural, un-reversed order, which this document calls the hash's *internal byte order*.

When a 32-byte hash is compared with a numeric target, as in the proof-of-work check, it is interpreted as a 256-bit unsigned integer in little-endian byte order. Read as a number, the display form is therefore the big-endian hexadecimal representation, which is why the hashes of blocks mined before the change of proof-of-work function are displayed with leading zeros. Since that change the proof-of-work hash and the block hash are different values (Chapter 9), and block hashes no longer have this appearance.

2.5 Amounts

The unit of account is the **decred**, abbreviated DCR. All amounts in the protocol are integers counting **atoms**, where

$$1\ \mathrm{DCR} = 10^8\ \text{atoms}. \tag{2.1}$$

Amounts are carried in the `int64` type. The largest amount that is valid anywhere in the protocol is `MaxAmount` $= 21 \times 10^6 \times 10^8$ atoms, twenty-one million decred. This is a bound used in range checks on individual amounts. It is not derived from the issuance schedule, whose exact total is computed in Chapter 11.

2.6 Heights, Time and Intervals

The **height** of a block is its distance from the genesis block, which has height 0. Heights are carried on the wire as `uint32` and used in computations as `int64`.

Timestamps are Unix time: whole seconds since 1970-01-01 00:00:00 UTC, ignoring leap seconds. Block header timestamps are `uint32` on the wire.

Many rules operate on fixed-length runs of consecutive blocks. An **interval** or **window** of length n always begins at a height that is a multiple of n unless a rule specifies an offset. The text states, for each such rule, which block is the first to which a newly computed value applies, because this is where off-by-one errors occur.

2.7 Pseudocode

Algorithms are given in a Python-like pseudocode. It is meant to be read, not executed, and follows these conventions:

- Integers are unbounded unless a variable is annotated with a fixed-width type or a conversion such as `uint32(x)` is applied. A conversion to an unsigned type reduces its argument modulo 2^w. A conversion to a signed type reduces modulo 2^w and reinterprets the result as two's complement.
- `a // b` is used only where both operands are non-negative, where floor and truncating division agree. Where an operand may be negative the functions `tdiv(a, b)` and `tmod(a, b)` denote truncating division and its remainder, and `fdiv(a, b)`, for `b > 0`, denotes flooring division, which rounds toward negative infinity.
- `>>` and `<<` are shifts with the semantics given in Section 2.3.
- `x[i]` indexes a sequence from zero, and `x[i:j]` is the subsequence from index `i` up to but not including `j`.
- `a + b` on byte strings is concatenation. `le16`, `le32`, `le64`, `be32` and so on serialize integers as described in Section 2.4.
- `reject(reason)` aborts validation: the object being validated is invalid.

Algorithm 2.1 shows the style.

ALGORITHM 2.1 Example: the sum of a list of amounts, with range checks

```
MAX_AMOUNT = 21_000_000 * 100_000_000

def checked_sum(amounts: list[int64]) -> int64:
    total = 0
    for amount in amounts:
        if amount < 0 or amount > MAX_AMOUNT:
            reject("output value out of range")
        total = total + amount          # cannot overflow int64 here; dcrd
                                        # checks for it all the same
        if total > MAX_AMOUNT:
            reject("total output value out of range")
    return total
```

dcrd: blockchain/standalone/tx.go · CheckTransactionSanity

2.8 Consensus Votes and Rule Activation

Decred's consensus rules have changed many times since launch, and since the first agendas of 2017 each change to the validation rules has been activated by a vote of the stakeholders (Chapter 13); the exceptions are the rules keyed to a proof-of-work majority of block versions (the rejection of superseded block versions and the enforcement of the header stake version, Chapter 10) and the testnet3 difficulty rules fixed at a hard-coded height (Chapter 9). Each change is identified by the DCP that specifies it and by the short **agenda identifier** under which it was voted, such as `blake3pow`. A rule that depends on a vote is described in terms of whether the agenda **is active for a block**, which is a deterministic function of the block's ancestors defined in Chapter 13. The heights at which each agenda became active on the main network are listed there as well, but an implementation must derive activation from the chain itself and not from a table of heights, since the other networks activate at different points.

2.9 Test Vectors

Where a computation can be checked in isolation, test vectors are supplied.

TEST VECTOR BLAKE-256 of the empty string

```
input   (0 bytes)
output  716f6e863f744b9ac22c97ec7b76ea5f5908bc5b2f67c61510bfc4751384ea7a
```

Every vector in this document was produced by running the reference implementation, not transcribed from secondary sources.

CHAPTER 3

Cryptographic Primitives

This chapter specifies every cryptographic primitive on which Decred's consensus rules depend: the hash functions, the elliptic curves, the three signature schemes that scripts can verify, and the Merkle tree used to commit to the transactions of a block. Later chapters use these primitives by name and do not repeat their definitions.

Decred differs from Bitcoin in its choice of primitives more than its structure suggests. Its general-purpose hash is BLAKE-256, not double SHA-256; its proof-of-work hash has been BLAKE-256 and then BLAKE3; and besides ECDSA over secp256k1 its script language can verify Schnorr signatures over secp256k1 (a Decred-specific scheme) and Ed25519 signatures. The definitions below are taken from the reference implementation. Where a primitive is specified by an external standard, the standard is cited and the parameters and any departures from it are stated.

Table 3.1 summarizes the primitives and where they appear.

Table 3.1 Cryptographic primitives and their uses

Primitive	Section	Used for
BLAKE-256 (14 rounds)	Section 3.2	Transaction and block hashes, Merkle trees, signature hashes, the version 1 proof-of-work hash, `OP_BLAKE256`, `OP_HASH160`, `OP_HASH256`, the Schnorr challenge
BLAKE3	Section 3.4	The version 2 proof-of-work hash (DCP-0011)
RIPEMD-160	Section 3.5	`OP_RIPEMD160`, `OP_HASH160`, address hashes
SHA-256	Section 3.5	`OP_SHA256`; HMAC inside deterministic nonce generation (not consensus)
SHA-1	Section 3.5	`OP_SHA1`
secp256k1	Section 3.6	Public keys for ECDSA and EC-Schnorr-DCRv0
ECDSA	Section 3.8	`OP_CHECKSIG`, `OP_CHECKMULTISIG` and their `VERIFY` forms
EC-Schnorr-DCRv0	Section 3.9	`OP_CHECKSIGALT` with signature type 2; treasury spend authorization
Ed25519	Section 3.10	`OP_CHECKSIGALT` with signature type 1
Merkle tree	Section 3.11	Transaction tree roots in the block header; header commitment proofs

3.1 Notation

A **word** is a `uint32`. For words, $a + b$ in the hash function definitions means addition modulo 2^{32}, $a \oplus b$ (`^` in pseudocode) is bitwise exclusive or, and $\mathrm{rotr}(a, n)$ is rotation to the right by n bits. In pseudocode the mask `M32 = 0xffffffff` makes the reduction modulo 2^{32} explicit.

For an integer x with $0 \le x < 2^{256}$, $\mathrm{BE}_{256}(x)$ is its 32-byte big-endian encoding and $\mathrm{LE}_{256}(x)$ its 32-byte little-endian encoding. For a byte string b, $\mathrm{int}_{\mathrm{BE}}(b)$ and $\mathrm{int}_{\mathrm{LE}}(b)$ interpret b as an unsigned integer in big-endian and little-endian order, and $b[i : j]$ is the substring of bytes i up to but not including j, counting from 0. $\|$ is concatenation.

The pseudocode follows Section 2.7 and uses these helpers in addition: `be32(w)`, `le32(w)` and `be64(x)` encode an integer as 4 or 8 bytes; `be32_decode(b)` and `le32_decode(b)` decode 4 bytes; `be256(x)` is $\mathrm{BE}_{256}(x)$;

`int_be(b)` and `int_le(b)` are int_{BE} and int_{LE}; `left_pad(b, 32)` prefixes zero bytes up to 32 bytes; `blake256` is Algorithm 3.2; `hmac_sha256(K, m)` is HMAC (RFC 2104 [8]) with SHA-256; `pow(a, e, m)` is modular exponentiation; and `scalar_mult`, `point_add` and `INFINITY` are elliptic curve scalar multiplication, point addition (complete, handling the identity and doubling) and $\mathcal{O}$.

For elliptic curves, upper-case letters denote points and lower-case letters integers. kG is scalar multiplication, $\mathcal{O}$ is the point at infinity (the group identity), and $P.x$, $P.y$ are the affine coordinates of a point $P \neq \mathcal{O}$, always as fully reduced integers. An integer is **odd** or **even** according to its least significant bit after full reduction.

3.2 BLAKE-256

BLAKE-256 is the 256-bit member of the BLAKE family, a finalist of the NIST SHA-3 competition, in the final version 1.3 of its specification, which uses 14 rounds [9]. Decred uses it wherever Bitcoin uses SHA-256 or double SHA-256. It is not BLAKE2s (a later, different function with 10 rounds and little-endian words), and it is not the 10-round BLAKE-32 of the first round of the competition. BLAKE-256 is an iterated hash with a 512-bit block, a 256-bit chaining value, and a 64-bit counter of message bits; the specification also defines an optional 128-bit salt. Decred always uses a zero salt, and the definition below is written with the salt removed.

RULE 3.1 BLAKE-256
Wherever this document writes BLAKE-256, or H, the function defined by Algorithm 3.1 and Algorithm 3.2 with the constants of Table 3.2, Table 3.3 and Table 3.4 is meant. Its output is always 32 bytes. An implementation **MUST** reproduce it bit for bit, including the counter rule for padding-only blocks.

dcrd: `crypto/blake256/hasher.go · finalize,finalize256`
`crypto/blake256/internal/compress/blocks_generic.go · blocksGeneric`

3.2.1 Constants

The initial chaining value (Table 3.2) is the same as SHA-256's. The sixteen constants $c_0, \ldots, c_{15}$ (Table 3.3) are the first 512 bits of the fractional part of π. Each round uses one of ten permutations $\sigma_0, \ldots, \sigma_9$ of the message word indices (Table 3.4); round r (counting from 0) uses $\sigma_{r \bmod 10}$, so rounds 10 to 13 repeat σ_0 to σ_3.

Table 3.2 BLAKE-256 initial chaining value IV (also the BLAKE3 IV)

IV_0	IV_1	IV_2	IV_3	IV_4	IV_5	IV_6	IV_7
`6a09e667`	`bb67ae85`	`3c6ef372`	`a54ff53a`	`510e527f`	`9b05688c`	`1f83d9ab`	`5be0cd19`

Table 3.3 BLAKE-256 constants c_i

c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7
`243f6a88`	`85a308d3`	`13198a2e`	`03707344`	`a4093822`	`299f31d0`	`082efa98`	`ec4e6c89`
c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
`452821e6`	`38d01377`	`be5466cf`	`34e90c6c`	`c0ac29b7`	`c97c50dd`	`3f84d5b5`	`b5470917`

Table 3.4 BLAKE message permutations σ_j: entry i of row j is $\sigma_j(i)$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
σ_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ_2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ_3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ_4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ_5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ_6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ_8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ_9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

3.2.2 Compression function

The compression function maps a chaining value $h_0, \ldots, h_7$, a 64-byte block and the counter t to a new chaining value. The block is read as sixteen **big-endian** words $m_0, \ldots, m_{15}$. A 16-word state $v_0, \ldots, v_{15}$, viewed as a 4×4 matrix, is initialized from the chaining value, the constants and the counter; each round applies the function G to the four columns and then to the four diagonals of the matrix. In the i-th application of G within round r, with $\sigma = \sigma_{r \bmod 10}$, the message word $m_{\sigma(2i)}$ is combined with the constant $c_{\sigma(2i+1)}$, and $m_{\sigma(2i+1)}$ with $c_{\sigma(2i)}$. In the pseudocode, `IV`, `C` and `SIGMA` are the tables Table 3.2, Table 3.3 and Table 3.4, indexed from 0.

ALGORITHM 3.1 BLAKE-256 compression

```
M32 = 0xffffffff

def rotr32(x, n):
    return ((x >> n) | (x << (32 - n))) & M32

def G(v, m, sigma, i, a, b, c, d):
    x, y = sigma[2 * i], sigma[2 * i + 1]
    v[a] = (v[a] + v[b] + (m[x] ^ C[y])) & M32
    v[d] = rotr32(v[d] ^ v[a], 16)
    v[c] = (v[c] + v[d]) & M32
    v[b] = rotr32(v[b] ^ v[c], 12)
    v[a] = (v[a] + v[b] + (m[y] ^ C[x])) & M32
    v[d] = rotr32(v[d] ^ v[a], 8)
    v[c] = (v[c] + v[d]) & M32
    v[b] = rotr32(v[b] ^ v[c], 7)

def blake256_compress(h: list[uint32], block: bytes, t: uint64) -> list[uint32]:
    m = [be32_decode(block[4 * i : 4 * i + 4]) for i in range(16)]
    t0, t1 = t & M32, t >> 32
    v = h[0:8] + C[0:4] + [t0 ^ C[4], t0 ^ C[5], t1 ^ C[6], t1 ^ C[7]]
    for r in range(14):
        sigma = SIGMA[r % 10]
        G(v, m, sigma, 0, 0, 4,  8, 12)     # columns
        G(v, m, sigma, 1, 1, 5,  9, 13)
        G(v, m, sigma, 2, 2, 6, 10, 14)
        G(v, m, sigma, 3, 3, 7, 11, 15)
        G(v, m, sigma, 4, 0, 5, 10, 15)     # diagonals
        G(v, m, sigma, 5, 1, 6, 11, 12)
        G(v, m, sigma, 6, 2, 7,  8, 13)
        G(v, m, sigma, 7, 3, 4,  9, 14)
    return [h[i] ^ v[i] ^ v[i + 8] for i in range(8)]
```

dcrd: crypto/blake256/internal/compress/blocks_generic.go · g, blocksGeneric

With a non-zero salt $s_0, \ldots, s_3$, the initial $v_8, \ldots, v_{11}$ would be $s_i \oplus c_i$ and the output $h_i \oplus s_{i \bmod 4} \oplus v_i \oplus v_{i+8}$; with the zero salt used by Decred these reduce to the expressions above.

3.2.3 Padding, counter and output

The message is padded with a 1 bit, then zero bits until its length is congruent to 447 modulo 512, then a 1 bit (which distinguishes BLAKE-256 from BLAKE-224, where this bit is 0), and finally the message length in bits as a 64-bit big-endian integer. For messages made of whole bytes this is the byte sequence `80 00 ... 00 01` followed by the length, where the first and last padding bytes merge into the single byte `81` when exactly one padding byte fits.

The counter t passed with each block is the number of **message** bits contained in that block and all preceding blocks. A block that contains no message bits at all, which is the case for the only block of the empty message, for the extra padding block appended when a message's final partial block has more than 55 bytes, and for the padding block of a message whose length is a non-zero multiple of 64, is compressed with $t = 0$.

ALGORITHM 3.2 BLAKE-256

```
def blake256(msg: bytes) -> bytes:
    L = len(msg)
    p = bytearray(msg) + b"\x80"
    while len(p) % 64 != 56:
        p += b"\x00"
    p[-1] |= 0x01                       # 0x80 becomes 0x81 if L % 64 == 55
    p += be64(8 * L)
    h = IV
    for j in range(len(p) // 64):
        if 64 * j < L:
            t = 8 * min(64 * (j + 1), L)    # message bits up to the end of block j
        else:
            t = 0                           # block holds padding only
        h = blake256_compress(h, p[64 * j : 64 * j + 64], t)
    return be32(h[0]) + be32(h[1]) + ... + be32(h[7])
```

dcrd: crypto/blake256/hasher.go · write, finalize, wordsToBytes256

IMPLEMENTATION PITFALL

Three details distinguish BLAKE-256 from the functions it resembles and are the usual causes of wrong output: the message words and the output are **big-endian** (BLAKE2 and BLAKE3 are little-endian); every message word is XORed with a constant before it is added, and the constant index is the **other** half of the σ pair; and a block containing only padding is compressed with counter 0, not with the total bit length. The vectors for 55-, 56- and 64-byte inputs below exercise the three padding cases.

TEST VECTOR BLAKE-256

input	BLAKE-256
(empty)	716f6e863f744b9ac22c97ec7b76ea5f 5908bc5b2f67c61510bfc4751384ea7a
00	0ce8d4ef4dd7cd8d62dfded9d4edb0a7 74ae6a41929a74da23109e8f11139c87
"abc"	1833a9fa7cf4086bd5fda73da32e5a1d 75b4c3f89d5c436369f9d78bb2da5c28
"a" x 55	6e8d7898571228c1106fcec9ef9c5db9 df8a3a2dcd2655a848af596d181bbae4
"a" x 56	ea7a29472a26148914abb8033869be9b dea294fdd2b73ed7a02a7692940f5b9e
"a" x 64	84d7f3bbf2cfc3ee940ddb6d25045c6d 3f756c4b2077a8128e171d5d165be170
00 x 72	d419bad32d504fb7d44d460c42c5593f e544fa4c135dec31e21bd9abdcc22d41
"The quick brown fox jumps over the lazy dog"	7576698ee9cad30173080678e5965916 adbb11cb5245d386bf1ffda1cb26c9d7

Quoted inputs are ASCII without a terminator; `"a" x 55` is 55 bytes of `0x61`. The 1-byte and 72-byte inputs are the examples from the BLAKE specification [9].

3.3 Hash Functions Derived from BLAKE-256

dcrd names its general-purpose hash `chainhash.HashB` (returning bytes) and `chainhash.HashH` (returning a 32-byte hash value); both are a single BLAKE-256. Transaction hashes, block hashes, Merkle tree nodes and signature hashes all use this single application. Table 3.5 lists the other compositions in use.

Table 3.5 Hash functions derived from BLAKE-256

Name	Definition	Used by
$H(x)$	BLAKE-256(x)	Everything not listed below
HASH256(x)	BLAKE-256(BLAKE-256(x))	`OP_HASH256` (the only consensus use); the Base58Check checksum (Chapter 5)
HASH160(x)	RIPEMD-160(BLAKE-256(x))	`OP_HASH160` ; public key hashes and script hashes in addresses (Chapter 5)
Message hash	BLAKE-256(varstr(Decred Signed Message:\n) ‖ varstr(m))	Signed messages (Section 3.8.4); not consensus

RULE 3.2 Derived hash functions

HASH160 **MUST** be computed as RIPEMD-160 of the 32-byte BLAKE-256 digest, giving 20 bytes. HASH256 **MUST** be computed as BLAKE-256 of the 32-byte BLAKE-256 digest. Where a consensus structure is said to be hashed without further qualification, a single BLAKE-256 is meant.

dcrd: chaincfg/chainhash/hashfuncs.go · HashB, HashH
txscript/opcode.go · opcodeHash160, opcodeHash256
txscript/stdaddr/addressv0.go · Hash160

IMPLEMENTATION PITFALL

Decred has no double hashing in its block or transaction identifiers, Merkle trees or signature hashes. An implementation adapted from Bitcoin code must remove the second hash everywhere in consensus except `OP_HASH256` , and must replace SHA-256 by BLAKE-256 inside HASH160.

TEST VECTOR Derived hashes

```
HASH256("")     d8ee5f957b78a961fb729098b4efb56440a14e05e3c55890f5edbc626380aaa6
HASH256("abc") f1a44bb867d5ac7053d5e071d62d4513d5b243420d9863e2718823a161c1344a
HASH160(02ba9cf60e2ea62eb020697b80a34eaaf17cdf7ee6114b3bdbdaac3494066a10fd)
                  2f4b63971d0e28df5c6fa3ace542adb5eee4b2ab
```

The public key is the one used in the ECDSA vectors of Section 3.8.

3.4 BLAKE3

Since the activation of the `blake3pow` agenda (DCP-0011), the proof-of-work hash of a block header is BLAKE3 [10] instead of BLAKE-256. dcrd computes it with the Go library `lukechampine.com/blake3` at version v1.3.0, in the default **hash** mode (unkeyed, no key derivation) with a 32-byte output. The input is the 180-byte serialized block header, the same bytes that are hashed with BLAKE-256 to form the block hash (Chapter 4). Which blocks use which proof-of-work hash, and how the result is compared with the target, is specified in Chapter 9.

RULE 3.3 BLAKE3 proof-of-work hash function

The BLAKE3 hash used by Decred **MUST** be the BLAKE3 hash function of [10] in hash mode with the default 32-byte output, which is the first 32 bytes of the extendable output. For inputs of at most 1024 bytes, including the 180-byte block header, it is fully specified by Algorithm 3.3; longer inputs, which do not occur in consensus, use the tree mode of [10].

dcrd: wire/blockheader.go · PowHashV2 | lukechampine.com/blake3 v1.3.0 · New, Sum

BLAKE3 splits its input into 1024-byte chunks, each made of up to sixteen 64-byte blocks, and combines the chunks in a binary tree. An input of at most 1024 bytes is a single chunk, which is also the root of the tree, so the hash is computed by chaining the compression function over the blocks of that chunk. The compression function shares the G function and rotation amounts of BLAKE-256 but differs in everything else: 7 rounds, little-endian words, no constants mixed into the message, a fixed message permutation applied between rounds, and a state that includes the block length and a set of domain flags.

Table 3.6 BLAKE3 parameters used for single-chunk inputs

Parameter	Value
IV	Identical to the BLAKE-256 IV of Table 3.2
Message permutation `PERM`	2, 6, 3, 10, 7, 0, 4, 13, 1, 11, 12, 5, 9, 14, 15, 8 (new word i is old word `PERM[i]`)
Flags	`CHUNK_START` = 1, `CHUNK_END` = 2, `ROOT` = 8 (`PARENT` = 4 and the keyed and key-derivation flags are not used for single-chunk unkeyed hashing)
Rounds	7
Counter	The chunk index, which is 0 for a single-chunk input

ALGORITHM 3.3 BLAKE3 for inputs of at most 1024 bytes

```
PERM = [2, 6, 3, 10, 7, 0, 4, 13, 1, 11, 12, 5, 9, 14, 15, 8]
CHUNK_START, CHUNK_END, ROOT = 1, 2, 8

def g3(v, a, b, c, d, x, y):
    v[a] = (v[a] + v[b] + x) & M32
    v[d] = rotr32(v[d] ^ v[a], 16)
    v[c] = (v[c] + v[d]) & M32
    v[b] = rotr32(v[b] ^ v[c], 12)
    v[a] = (v[a] + v[b] + y) & M32
    v[d] = rotr32(v[d] ^ v[a], 8)
    v[c] = (v[c] + v[d]) & M32
    v[b] = rotr32(v[b] ^ v[c], 7)

def blake3_compress(cv, m, counter: uint64, block_len, flags) -> list[uint32]:
    v = cv[0:8] + IV[0:4] + [counter & M32, counter >> 32, block_len, flags]
    for r in range(7):
        g3(v, 0, 4,  8, 12, m[0],  m[1]);  g3(v, 1, 5,  9, 13, m[2],  m[3])
        g3(v, 2, 6, 10, 14, m[4],  m[5]);  g3(v, 3, 7, 11, 15, m[6],  m[7])
        g3(v, 0, 5, 10, 15, m[8],  m[9]);  g3(v, 1, 6, 11, 12, m[10], m[11])
        g3(v, 2, 7,  8, 13, m[12], m[13]); g3(v, 3, 4,  9, 14, m[14], m[15])
        m = [m[PERM[i]] for i in range(16)]
    return [v[i] ^ v[i + 8] for i in range(8)] + [v[i + 8] ^ cv[i] for i in range(8)]

def blake3_256(msg: bytes) -> bytes:        # len(msg) <= 1024
    n = max(1, (len(msg) + 63) // 64)       # block count; the empty input has one
    cv = IV
    for j in range(n):
        block = msg[64 * j : 64 * j + 64]
        block_len = len(block)              # 64, except possibly for the last block
        block = block + bytes(64 - block_len)
        m = [le32_decode(block[4 * i : 4 * i + 4]) for i in range(16)]
        flags = 0
        if j == 0:
            flags |= CHUNK_START
        if j == n - 1:
            flags |= CHUNK_END | ROOT
        cv = blake3_compress(cv, m, 0, block_len, flags)[0:8]
    return le32(cv[0]) + le32(cv[1]) + ... + le32(cv[7])
```

dcrd: lukechampine.com/blake3 v1.3.0 · guts.CompressNode, guts.CompressChunk
blake3.go · Hasher.Sum, rootNode

For the 180-byte header the chunk has three blocks of 64, 64 and 52 bytes, compressed with flags 1, 0 and 10 respectively and block lengths 64, 64 and 52.

TEST VECTOR BLAKE3 (32-byte output)

```
(empty)    af1349b9f5f9a1a6a0404dea36dcc9499bcb25c9adc112b7cc9a93cae41f3262
"abc"      6437b3ac38465133ffb63b75273a8db548c558465d79db03fd359c6cd5bd9d85
00 x 180   3419118e1c6537b8de8a2e6a22dcc2734c0d92729ce542484de3e829e6e67f12
```

TEST VECTOR Main-network genesis block header

```
serialized header (180 bytes):
  01000000000000000000000000000000000000000000000000000000000000000000000000
  000000000dc101dfc3c6a2eb10ca0c5374e10d28feb53f7eabcc850511ceadb9
  9174aa6600000000000000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000ffff011b00c2eb0b00000000
  0000000000000000a0d7b85600000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000
BLAKE-256  80d9212bf4ceb066ded2866b39d4ed89e0ab60f335c11df8e7bf85d9c35c8e29
BLAKE3     1baa4b79809fc85fac9333371d37e203f34c345595f5ed5ed435fbc10b09d3d7
```

Both hashes are shown with their bytes in order. The BLAKE-256 value is the genesis block hash, displayed in reversed byte order as in Section 2.4. The BLAKE3 value is only an example of the function: the genesis block predates DCP-0011 and is never checked against it.

3.5 SHA-256, SHA-1 and RIPEMD-160

These three functions are used exactly as standardized and are specified by reference. SHA-256 and SHA-1 are defined in FIPS 180-4 [11]; RIPEMD-160 is defined by its authors [12]. dcrd uses Go's standard library for SHA-256 and SHA-1 and a copy of the `golang.org/x/crypto` RIPEMD-160 code (`crypto/ripemd160`).

In consensus they appear only as script opcodes: `OP_RIPEMD160`, `OP_SHA1`, `OP_SHA256` (which is interpreted as SHA-256 only when the corresponding script flag is in effect), and inside `OP_HASH160`. Their stack semantics and activation are given in Chapter 6. SHA-256 also appears, outside consensus, as the hash of the HMAC used for deterministic nonces (Algorithm 3.6), and SHA-512 inside Ed25519 (Section 3.10).

TEST VECTOR SHA-256, SHA-1 and RIPEMD-160

```
SHA-256("")      e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
SHA-256("abc")   ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
SHA-1("")        da39a3ee5e6b4b0d3255bfef95601890afd80709
SHA-1("abc")     a9993e364706816aba3e25717850c26c9cd0d89d
RIPEMD-160("")   9c1185a5c5e9fc54612808977ee8f548b2258d31
RIPEMD-160("abc")
                 8eb208f7e05d987a9b044a8e98c6b087f15a0bfc
```

3.6 The secp256k1 Curve

ECDSA and EC-Schnorr-DCRv0 both operate on secp256k1 [13], the Koblitz curve $y^2 = x^3 + 7$ over the prime field $\mathbb{F}_p$. Its group of points has prime order n and cofactor 1, so every point other than $\mathcal{O}$ generates the whole group. Table 3.7 lists the domain parameters.

Table 3.7 secp256k1 domain parameters (hexadecimal, big-endian)

Parameter	Value
p	`fffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f` $= 2^{256} - 2^{32} - 977$

Parameter	Value
a, b	$a = 0, b = 7$
$G.x$	`79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798`
$G.y$	`483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8`
n	`fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141`
$\frac{n-1}{2}$	`7fffffffffffffffffffffffffffffff5d576e7357a4501ddfe92f46681b20a0`
$p - n$	`14551231950b75fc4402da1722fc9baee`
Cofactor h	1

A **scalar** is an integer modulo n. A **private key** is a scalar d with $1 \le d \le n - 1$, and its **public key** is the point $Q = dG$.

3.6.1 Public key encodings

Public keys are serialized in the formats of SEC 1 [14] and ANSI X9.62, with coordinates as 32-byte big-endian integers. The first byte identifies the format (Table 3.8). dcrd always produces the compressed format, or the uncompressed format on request, and never produces the hybrid format.

Table 3.8 secp256k1 public key formats and where each is accepted

Format	Prefix	Length	Parse (Rule 3.4)	Strict (Rule 3.5)	Schnorr (Rule 3.10)
Compressed	`02` (y even), `03` (y odd)	33	yes	yes	yes
Uncompressed	`04`	65	yes	yes	no
Hybrid	`06` (y even), `07` (y odd)	65	yes	no	no
Anything else			no	no	no

RULE 3.4 Parsing a secp256k1 public key

A byte string b parses as a secp256k1 public key if and only if one of the following holds; otherwise parsing fails.

1. $|b| = 65$, $b_0 \in \{$ `04` , `06` , `07` $\}$, and with $x = \mathrm{int}_{\mathrm{BE}}(b[1:33])$ and $y = \mathrm{int}_{\mathrm{BE}}(b[33:65])$: $x < p$, $y < p$, if $b_0 =$ `06` then y is even, if $b_0 =$ `07` then y is odd, and $y^2 \equiv x^3 + 7 \pmod p$. The key is (x, y).
2. $|b| = 33$, $b_0 \in \{$ `02` , `03` $\}$, $x = \mathrm{int}_{\mathrm{BE}}(b[1:33]) < p$, and $x^3 + 7$ is a quadratic residue modulo p. The key is (x, y) where y is the square root of $x^3 + 7$ modulo p that is even if $b_0 =$ `02` and odd if $b_0 =$ `03` .

Every other length, and every other prefix byte for the given length, fails. A successfully parsed key is never $\mathcal{O}$, since $\mathcal{O}$ has no encoding.

dcrd: dcrec/secp256k1/pubkey.go · PublicKey.parse, ParsePubKey

Because $p \equiv 3 \pmod 4$, the square root of a residue a is $a^{\frac{p+1}{4}} \bmod p$; $x^3 + 7$ is a residue exactly when the square of that value equals it.

The script engine applies a stricter syntactic test to ECDSA public keys before parsing them. It is part of the consensus rules wherever the script chapter applies it (Chapter 6); it rejects the hybrid format even though the parser accepts it.

RULE 3.5 Strict public key encoding

A public key byte string is **strictly encoded** if and only if it is 33 bytes long with first byte `02` or `03` , or 65 bytes long with first byte `04` . Strict encoding is purely syntactic: it does not check that the coordinates are in range or that the point is on the curve.

dcrd: txscript/engine.go · isStrictPubKeyEncoding | txscript/consensus.go · CheckPubKeyEncoding

Since ECDSA public keys in scripts must pass Rule 3.5 before they are parsed, and EC-Schnorr-DCRv0 keys must be compressed (Rule 3.10), a hybrid key can never validate a signature in consensus even though Rule 3.4 accepts it. A key that is strictly encoded but fails Rule 3.4 (for example, one not on the curve) makes the signature check fail rather than aborting the script (Chapter 6).

TEST VECTOR Public key parsing

```
key                                parse                        strict
02ba9c...10fd (compressed)         ok                           ok
04ba9c...1674 (uncompressed)       ok                           ok
06ba9c...1674 (hybrid, y even)     ok                           rejected
07ba9c...1674 (hybrid, y even)     fails: parity mismatch       rejected
05ba9c...1674                      fails: unsupported format    rejected
04ba9c...1675 (y + 1)              fails: not on curve          ok
02 || BE256(p)                     fails: x >= p                ok
02 || BE256(5)                     fails: x not on curve        ok
first 32 bytes of 02ba9c...        fails: length                rejected
```

The full keys are those of the ECDSA vector in Section 3.8.5.

3.6.2 Scalars from byte strings

Several algorithms below convert a byte string b to a scalar. dcrd does this by taking the first $\min(|b|, 32)$ bytes of b, left-padding them with zero bytes to 32 bytes, and interpreting the result as a big-endian integer z. It reports **overflow** if $z \geq n$ and otherwise yields z. Where an algorithm below reduces (written `scalar_reduce(b)` in pseudocode), the value used is $z \bmod n$; where it rejects on overflow, $z \geq n$ is a failure. The same conversion into a field element reports overflow when $z \geq p$.

3.7 Signature Schemes in Consensus

Scripts verify signatures with two families of opcodes, whose full semantics are given in Chapter 6. This chapter owns the mathematics and the encodings; Table 3.9 shows how they connect.

Table 3.9 Signature schemes reachable from scripts

Opcode family	Type	Scheme	Public key	Signature (excluding the hash type byte)
`OP_CHECKSIG`, `OP_CHECKMULTISIG`		ECDSA / secp256k1	33 or 65 bytes, strict	DER, 8 to 72 bytes
`OP_CHECKSIGALT`	0	none		Always yields false
`OP_CHECKSIGALT`	1	Ed25519	32 bytes	64 bytes
`OP_CHECKSIGALT`	2	EC-Schnorr-DCRv0 / secp256k1	33 bytes, compressed	64 bytes
`OP_CHECKSIGALT`	other	none		Always yields true (reserved for future soft forks)

The signature type numbers are the constants `STEcdsaSecp256k1` = 0, `STEd25519` = 1 and `STSchnorrSecp256k1` = 2 defined in `dcrec/common.go`. ECDSA is never selected through `OP_CHECKSIGALT`: type 0 makes it yield false. Types 1 and 2 are executed by the consensus engine, so both alternative schemes are reachable in consensus. In all cases the message that is signed is the 32-byte signature hash of the spending transaction defined in Chapter 6, and every signature carries one trailing hash type byte that is removed before the signature is parsed. Outside scripts, EC-Schnorr-DCRv0 also authorizes treasury spends (Chapter 12); that 64-byte signature carries no hash type byte.

3.8 ECDSA

ECDSA over secp256k1 is the primary signature scheme. Verification is standard ECDSA [14]; Decred's specifics lie in the signature encoding and in deterministic signing.

3.8.1 Verification

RULE 3.6 ECDSA verification

An ECDSA signature (r, s) is valid for a 32-byte message hash z and a public key Q if and only if Algorithm 3.4 returns true. In consensus, (r, s) comes from Rule 3.7, Q from Rule 3.4, and z is the signature hash.

dcrd: dcrec/secp256k1/ecdsa/signature.go · Signature.Verify

ALGORITHM 3.4 ECDSA verification

```
def ecdsa_verify(r: int, s: int, z: bytes, Q: Point) -> bool:
    if r == 0 or s == 0:                    # cannot happen after DER parsing
        return False
    e = scalar_reduce(z)                    # first 32 bytes, big-endian, mod n
    w = pow(s, n - 2, n)                    # s^-1 mod n
    u1 = e * w % n
    u2 = r * w % n
    X = point_add(scalar_mult(u1, G), scalar_mult(u2, Q))
    if X == INFINITY:
        return False
    return X.x % n == r
```

dcrd: dcrec/secp256k1/ecdsa/signature.go · Signature.Verify

NOTE

dcrd avoids converting X to affine coordinates: with X in Jacobian coordinates (X_J, Y_J, Z_J) it accepts if $rZ_J^2 \equiv X_J$ or, when $r + n < p$, if $(r + n)Z_J^2 \equiv X_J$ modulo p. Since $X.x < p < 2n$, this is exactly the condition $X.x \bmod n = r$.

IMPLEMENTATION PITFALL

u_1 is zero when $e \equiv 0$, making $u_1 G = \mathcal{O}$, and $u_1 G + u_2 Q$ may be $\mathcal{O}$. Point addition must handle the identity and the doubling case; a formula that assumes distinct, non-identity operands produces wrong verdicts on such inputs.

3.8.2 DER signature encoding

ECDSA signatures in scripts are encoded with the ASN.1 Distinguished Encoding Rules (DER, ITU-T X.690 [15]) as a sequence of two integers (Table 3.10). All lengths are single bytes, since no valid signature needs a long-form length.

Table 3.10 DER encoding of an ECDSA signature

Field	Type	Bytes	Description
seq	byte	1	`0x30`, the ASN.1 SEQUENCE identifier.
len	byte	1	Number of remaining bytes; must equal the signature length minus 2.
rtype	byte	1	`0x02`, the ASN.1 INTEGER identifier.
rlen	byte	1	Length ℓ_r of R in bytes.
R	bytes	ℓ_r	r as a minimal big-endian two's complement integer.
stype	byte	1	`0x02`.
slen	byte	1	Length ℓ_s of S in bytes.

Field	Type	Bytes	Description
S	bytes	ℓ_s	s as a minimal big-endian two's complement integer.

"Minimal two's complement" means that the first byte of an integer must not have its high bit set (that would make it negative), and a leading `0x00` byte is allowed only when the following byte has its high bit set. The shortest valid signature is therefore 8 bytes ($\ell_r = \ell_s = 1$) and the longest 72 bytes ($\ell_r = \ell_s = 33$).

RULE 3.7 Parsing a DER signature

Let b be the signature with its hash type byte removed and $L = |b|$. b parses as an ECDSA signature if and only if all of the following hold, where offsets are zero-based, $\ell_r = b_3$, $o_s = 4 + \ell_r$ (the offset of the S type byte), and $\ell_s = b_{o_s+1}$:

1. $8 \le L \le 72$.
2. $b_0 =$ `0x30`.
3. $b_1 = L - 2$.
4. $o_s < L$ and $o_s + 1 < L$ (the S type and length bytes are inside b).
5. $o_s + 2 + \ell_s = L$ (the lengths account for every byte; there is no trailing data).
6. $b_2 =$ `0x02`.
7. $\ell_r \neq 0$.
8. b_4 has its high bit clear (R is not negative).
9. Not ($\ell_r > 1$ and $b_4 =$ `0x00` and b_5 has its high bit clear) (R has no excess leading zero).
10. $b_{o_s} =$ `0x02`.
11. $\ell_s \neq 0$.
12. b_{o_s+2} has its high bit clear (S is not negative).
13. Not ($\ell_s > 1$ and $b_{o_s+2} =$ `0x00` and b_{o_s+3} has its high bit clear) (S has no excess leading zero).
14. With all leading zero bytes stripped from the R field, at most 32 bytes remain, and their big-endian value r satisfies $1 \le r \le n - 1$.
15. With all leading zero bytes stripped from the S field, at most 32 bytes remain, and their big-endian value s satisfies $1 \le s \le n - 1$.

The result is the pair (r, s). The parser does **not** require $s \le (n - 1)/2$; high-S signatures parse and verify.

dcrd: dcrec/secp256k1/ecdsa/signature.go · ParseDERSignature

The script engine additionally applies a **strict encoding** check to every ECDSA signature before parsing it (Chapter 6). It repeats items 1 to 13 of Rule 3.7 and adds the low-S requirement, but it does not check the range of r: a signature that is strictly encoded but has $r = 0$ or $r \ge n$, or $s = 0$, passes the strict check and then fails to parse. The two outcomes differ in script: a strict-encoding failure aborts the script with an error, whereas a parse failure makes the signature check return false.

RULE 3.8 Strict ECDSA signature encoding

A signature (without hash type) is **strictly encoded** if and only if it satisfies items 1 to 13 of Rule 3.7 and, with all leading zero bytes stripped from the S field, at most 32 bytes remain and their value s satisfies $s < n$ and $s \le (n - 1)/2$. In particular a strictly encoded signature has $s \le$ `7fffffffffffffffffffffffffffffff5d576e7357a4501ddfe92f46681b20a0`.

dcrd: txscript/consensus.go · CheckSignatureEncoding

RATIONALE

If (r, s) is a valid signature then so is $(r, n - s)$. Without the low-S rule anyone could change the S value of a signature in a transaction, and thereby the transaction's full hash, without invalidating it. Requiring $s \le (n - 1)/2$ leaves one valid choice.

3.8.3 Signing

Signing is not part of consensus, but dcrd's signatures are deterministic and implementations that wish to reproduce them, or the test vectors, must follow Algorithm 3.5. The nonce is generated per RFC 6979 [16] with HMAC-SHA256 (Algorithm 3.6), and s is normalized to the lower half of the range as required by Rule 3.8.

ALGORITHM 3.5 Deterministic ECDSA signing

```
def ecdsa_sign(d: int, z: bytes) -> tuple[int, int, int]:
    for iteration in range(0, 2**32):
        k = nonce_rfc6979(be256(d), z, b"", iteration)
        X = scalar_mult(k, G)
        r = X.x % n
        if r == 0:
            continue
        e = scalar_reduce(z)
        s = pow(k, n - 2, n) * (e + d * r) % n
        if s == 0:
            continue
        recid = (2 if X.x >= n else 0) | (X.y & 1)   # public key recovery code
        if s > (n - 1) // 2:
            s = n - s
            recid ^= 1                               # -k gives the opposite y parity
        return r, s, recid
```

dcrd: dcrec/secp256k1/ecdsa/signature.go · signRFC6979, sign

A signature is serialized in DER (Table 3.10) by encoding r and s as 32-byte big-endian values, prefixing a zero byte, and stripping leading zero bytes while more than one byte remains and the next byte has its high bit clear.

ALGORITHM 3.6 RFC 6979 nonce generation as implemented

```
def nonce_rfc6979(key: bytes, z: bytes, extra: bytes, iteration: uint32) -> int:
    key = left_pad(key[0:32], 32)       # truncate to 32 bytes or left-pad with zeros
    z = left_pad(z[0:32], 32)
    data = key + z
    if len(extra) == 32:                # extra data is used only if exactly 32 bytes
        data = data + extra
    V = b"\x01" * 32
    K = b"\x00" * 32
    K = hmac_sha256(K, V + b"\x00" + data)
    V = hmac_sha256(K, V)
    K = hmac_sha256(K, V + b"\x01" + data)
    V = hmac_sha256(K, V)
    generated = 0
    while True:
        V = hmac_sha256(K, V)
        k = int_be(V)
        if 1 <= k <= n - 1:
            generated += 1
            if generated > iteration:
                return k
        K = hmac_sha256(K, V + b"\x00")
        V = hmac_sha256(K, V)
```

dcrd: dcrec/secp256k1/nonce.go · NonceRFC6979

The function also accepts an optional 16-byte "version" string that is appended after the extra data (with 32 zero bytes standing in for absent extra data); no signature scheme in this chapter uses it.

IMPLEMENTATION PITFALL
RFC 6979 feeds $\text{bits2octets}(z) = \text{int}_{\text{BE}}(z) \bmod n$ into the HMAC. dcrd feeds z itself, unreduced. The two differ only when $\text{int}_{\text{BE}}(z) \geq n$, which happens with probability about 2^{-128} for a hash output, but an implementation that reduces z produces different nonces, and therefore different signatures, for such messages.

3.8.4 Compact recoverable signatures and signed messages

A **compact signature** is a 65-byte encoding of an ECDSA signature from which the signer's public key can be recovered. It is used for signed messages (the `signmessage` interface of wallets and the `verifymessage` interface of wallets and dcrd; Chapter 5 gives the message format) and is not part of consensus.

Table 3.11 Compact signature

Field	Type	Bytes	Description
`code`	`byte`	1	$27 + \text{recid} + 4c$, where recid is the recovery code of Algorithm 3.5 and $c = 1$ if the signer's address uses the compressed public key, else 0. Valid values are 27 to 34.
`r`	`bytes`	32	$\text{BE}_{256}(r)$.
`s`	`bytes`	32	$\text{BE}_{256}(s)$, with $s \leq (n-1)/2$ as produced by signing.

Bit 0 of the recovery code is the parity of $X.y$ for the nonce point X (after the low-S adjustment), and bit 1 is set when $X.x \geq n$, so that $r = X.x - n$. Recovery from a compact signature and a 32-byte hash z proceeds as follows, failing at any step that cannot be completed:

1. Require the length to be 65 and the first byte c to satisfy $27 \leq c \leq 34$. Let $c' = c - 27$; the original key was compressed if bit 2 of c' is set, and $\text{recid} = c' \bmod 4$.
2. Parse r and s from the next two 32-byte fields; require $1 \leq r \leq n-1$ and $1 \leq s \leq n-1$ (rejecting on overflow).
3. Let $x = r$. If bit 1 of recid is set, require $r + n < p$ and let $x = r + n$.
4. Find y with $y^2 \equiv x^3 + 7 \pmod p$ whose parity equals bit 0 of recid; fail if none exists. Let $X = (x, y)$.
5. Let e be z reduced as in Section 3.6.2, $w = r^{-1} \bmod n$, $u_1 = -ew \bmod n$, $u_2 = sw \bmod n$, and $Q = u_1 G + u_2 X$. Fail if $Q = \mathcal{O}$.
6. The recovered key is Q, to be serialized compressed if compressed is set and uncompressed otherwise.

A signed message m (a UTF-8 string) is verified by computing the message hash

$$z = \text{BLAKE-256}(\text{varstr}(\texttt{"Decred Signed Message:\textbackslash n"}) \parallel \text{varstr}(m)), \tag{3.1}$$

where varstr is the variable-length string encoding of Chapter 4 (a variable-length integer byte count followed by the bytes), recovering Q from the compact signature and z, and comparing the version 0 pay-to-public-key-hash address of HASH160 of Q's serialization with the claimed address (Chapter 5). Unlike Bitcoin, the message hash is a single BLAKE-256, not double SHA-256.

3.8.5 Test vectors

TEST VECTOR ECDSA key, nonce and signature

```
d   cbe35df915b1e955c4956bba366d95256a7ee187c9d5267a8f0c69afae669c0e
      (= BLAKE-256("Decred crypto chapter test key"))
z   5a029fb117f154fc26caa93ca613772e2795e5776e2fd4208abbf6386be52dc7
      (= BLAKE-256("Decred crypto chapter test message"))
Q compressed  02ba9cf60e2ea62eb020697b80a34eaaf17cdf7ee6114b3bdbdaac3494066a10fd
Q uncompr.    04ba9cf60e2ea62eb020697b80a34eaaf17cdf7ee6114b3bdbdaac3494066a10fd
                5b94e113908bde6f4dc99d83494af73fdf7db95194617ff0418056a978481674
k (iter. 0)   83d68d3a073a926fdd7f485df64f27e829bc7c07334b795f5d36b161f5139ab1
X.x           c766d1c56552499bca21cc0f329b3ca2ecd96b501a60e5797576f33331c6ea4e
X.y parity    even
s (raw)       a8184510d564a0d715fdff73d704cb78356d3631ed1579eba2ddef42acd1722a
```

```
r            c766d1c56552499bca21cc0f329b3ca2ecd96b501a60e5797576f33331c6ea4e
s            57e7baef2a9b5f28ea02008c28fb34868541a6b4c23326501cf46f4a2364cf17
DER  3045022100c766d1c56552499bca21cc0f329b3ca2ecd96b501a60e5797576f33331c6ea4e
     022057e7baef2a9b5f28ea02008c28fb34868541a6b4c23326501cf46f4a2364cf17
compact (compressed key, recid 1: code 27 + 1 + 4 = 0x20)
     20c766d1c56552499bca21cc0f329b3ca2ecd96b501a60e5797576f33331c6ea4e
       57e7baef2a9b5f28ea02008c28fb34868541a6b4c23326501cf46f4a2364cf17
```

Here the raw s exceeded $(n-1)/2$ and was negated, which flipped the recovery code's parity bit from 0 to 1.

TEST VECTOR Signed message

```
message "hello"; key d as above
z = BLAKE-256(17 "Decred Signed Message:\n" 05 "hello")
  = 776fea952d41c5269b91e9710afcd91103ad41a06e814f8ecba72f49044fdfe6
compact signature
  208900fad91d879b2643d8b2046f033ca389247e3a9fe499d6d7908cb72d14712f
    07e9ce1734285c6f9a79ef3617ce8453f38d9a104c93cd14b4d7c7070dc236af
```

TEST VECTOR DER parsing and strict encoding

The base signature is the DER signature above, with `R = 00c766...ea4e` and `S = 57e7...cf17`.

Modification	Parse (Rule 3.7)	Strict (Rule 3.8)
None	ok	ok
S replaced by $n - s$ (`022100a818...722a`, 72 bytes)	ok	high S
Extra `00` before R (`02220000c766...`)	excess padding (9)	excess padding
First byte of R field `80` instead of `00`	negative R (8)	negative R
R = `00` (`020100`)	R is 0 (14)	ok
R = `00` $\|\|$ $\mathrm{BE}_{256}(n)$	R >= n (14)	ok
S = `00` (`020100`)	S is 0 (15)	ok
S length 0 (`0200`)	zero S length (11)	zero S length
First byte `31`	sequence id (2)	sequence id
Length byte `46` instead of `45`	bad length (3)	bad length
One trailing byte appended	bad length (3)	bad length
R type byte `03`	R integer id (6)	R integer id
`3006020101020101` ($r = s = 1$)	ok	ok
`30060201010201` (7 bytes)	too short (1)	too short

Numbers in parentheses are the items of Rule 3.7 that fail.

3.9 EC-Schnorr-DCRv0

EC-Schnorr-DCRv0 is a Schnorr signature scheme [17] over secp256k1 defined by Decred. A signature is the pair (r, s) where r is the x coordinate of the nonce point R, whose y coordinate is required to be even, and $s = k - ed \bmod n$ with challenge $e = \text{BLAKE-256}(\mathrm{BE}_{256}(r) \,\|\, m)$. Verification checks $sG + eQ = R$. It is not compatible with BIP 340 [18] Schnorr signatures, which use a different challenge hash, the opposite sign convention and x-only public keys. The scheme is described in the README of the `schnorr` package, which this section follows; where the README and the code differ in detail, the code is authoritative.

3.9.1 Encoding

Table 3.12 EC-Schnorr-DCRv0 signature

Field	Type	Bytes	Description
r	bytes	32	$\mathrm{BE}_{256}(r)$, the x coordinate of R, with $r < p$.
s	bytes	32	$\mathrm{BE}_{256}(s)$, with $s < n$.

RULE 3.9 Parsing an EC-Schnorr-DCRv0 signature

A byte string b parses as an EC-Schnorr-DCRv0 signature if and only if $|b| = 64$, $r = \mathrm{int}_{\mathrm{BE}}(b[0:32]) < p$ and $s = \mathrm{int}_{\mathrm{BE}}(b[32:64]) < n$. No other condition is checked: $r = 0$, $s = 0$, and an r that is not the x coordinate of any curve point all parse (and then fail or succeed in verification on their merits).

dcrd: dcrec/secp256k1/schnorr/signature.go · ParseSignature

RULE 3.10 Parsing an EC-Schnorr-DCRv0 public key

A byte string parses as an EC-Schnorr-DCRv0 public key if and only if it is exactly 33 bytes long, its first byte is `02` or `03`, and it parses as a compressed key under Rule 3.4. Uncompressed and hybrid keys are rejected.

dcrd: dcrec/secp256k1/schnorr/pubkey.go · ParsePubKey

3.9.2 Verification

RULE 3.11 EC-Schnorr-DCRv0 verification

A parsed signature (r, s) is valid for a message m and a parsed public key Q if and only if Algorithm 3.7 returns true.

dcrd: dcrec/secp256k1/schnorr/signature.go · schnorrVerify, Signature.Verify

ALGORITHM 3.7 EC-Schnorr-DCRv0 verification

```
def schnorr_verify(r: int, s: int, m: bytes, Q: Point) -> bool:
    if len(m) != 32:
        return False
    if not on_curve(Q):                # always true for a parsed key
        return False
    e = int_be(blake256(be256(r) + m))  # r is always encoded as 32 bytes
    if e >= n:                         # no reduction: overflow is a failure
        return False
    R = point_add(scalar_mult(s, G), scalar_mult(e, Q))
    if R == INFINITY:
        return False
    if R.y % 2 == 1:
        return False
    return R.x == r
```

dcrd: dcrec/secp256k1/schnorr/signature.go · schnorrVerify

IMPLEMENTATION PITFALL

The challenge e is not reduced modulo n: a hash value $e \geq n$ makes the signature invalid. The comparison $R.x = r$ is over the field, not modulo n. Because $s = 0$ and $e = 0$ are both possible inputs, point arithmetic must handle $\mathcal{O}$ correctly.

3.9.3 Signing

Signing is deterministic. The nonce comes from Algorithm 3.6 with 32 bytes of extra data that separate it from ECDSA nonces for the same key and message:

`extra` = BLAKE-256(`"EC-Schnorr-DCRv0"`) = `0b75f97b60e8a5762876c004829ee9b926fa6f0d2eeaec3a4fd1446a768331cb`

ALGORITHM 3.8 **EC-Schnorr-DCRv0 signing**

```
def schnorr_sign(d: int, m: bytes) -> tuple[int, int]:
    if len(m) != 32 or d == 0:
        fail()
    for iteration in range(0, 2**32):
        k = nonce_rfc6979(be256(d), m, EXTRA_DCRV0, iteration)
        R = scalar_mult(k, G)
        if R.y % 2 == 1:
            k = n - k                          # -k gives a nonce point with even y
        r = R.x
        e = int_be(blake256(be256(r) + m))
        if e >= n:
            continue
        s = (k - e * d) % n
        return r, s
```

dcrd: `dcrec/secp256k1/schnorr/signature.go` · `Sign`, `schnorrSign`

TEST VECTOR EC-Schnorr-DCRv0

```
d, z, Q as in the ECDSA vector (Q compressed = 02ba9c...10fd)
k (iter. 0)  18de4f626d03fb5ce2b99e35c1a647001de4a09b17b10575f7f14069885fa651
R.y parity   even (k not negated)
r            fc941c3662ebb36c6ac57a1399e5d1c914ebba55e5632e450f5069d9c6023954
e            f574323b5a096cf37a2fc5c63dc0ab696204ae903083a9e89835a6918dc67ba1
s            908e6a2805b90ade93188d193394e10d3c1e178522a316237618b3d4d40fa229
signature    fc941c3662ebb36c6ac57a1399e5d1c914ebba55e5632e450f5069d9c6023954
             908e6a2805b90ade93188d193394e10d3c1e178522a316237618b3d4d40fa229
```

Parsing edge cases: 64 zero bytes parse ($r = s = 0$) but cannot verify; replacing s by $\mathrm{BE}_{256}(n)$ fails to parse ($s \geq n$); replacing r by $\mathrm{BE}_{256}(p)$ fails to parse ($r \geq p$); the 65-byte uncompressed form of Q is rejected as a public key.

3.10 Ed25519

Ed25519 [19] is EdDSA over the twisted Edwards curve edwards25519, as standardized in RFC 8032 [20]. dcrd's implementation, the `dcrec/edwards` package (module `github.com/decred/dcrd/dcrec/edwards/v2` at version v2.0.4, which dcrd's `go.mod` takes as a published module rather than from its own source tree; the two copies differ only in their tests), wraps the Go library `github.com/agl/ed25519` (pinned at version `v0.0.0-20170116200512-5312a6153412`), adding its own decoding and range checks. Signatures produced by dcrd from a 32-byte secret are exactly RFC 8032 Ed25519 signatures, and honestly generated RFC 8032 signatures on a 32-byte message verify. The differences lie in the treatment of unusual encodings and of $S = 0$, and are specified below.

Table 3.13 edwards25519 parameters

Parameter	Value
Field prime q	$2^{255} - 19$
Curve	$-x^2 + y^2 = 1 + dx^2y^2$
d	$-121665/121666 \bmod q$ = `52036cee2b6ffe738cc740797779e89800700a4d4141d8ab75eb4dca135978a3`
Group order ℓ	$2^{252} + 27742317777372353535851937790883648493$ = `1000000000000000000000000000000014def9dea2f79cd65812631a5cf5d3ed`
Cofactor	8

Parameter	Value
Base point B	$B.y = 4/5 \bmod q$ = 6666666666666666666666666666666666666666666666666666666666666658, $B.x$ even = 216936d3cd6e53fec0a4e231fdd6dc5c692cc7609525a7b2c9562d608f25d51a; encoded 5866666666666666666666666666666666666666666666666666666666666666

3.10.1 Point encoding and decoding

A point (x, y) is encoded as 32 bytes: $\mathrm{LE}_{256}(y)$ with the most significant bit of the last byte set to the least significant bit of x (the "sign" of x). The encoding produced by the implementation always has $y < q$ (it is **canonical**).

Decoding, as performed by the library and relied upon by every rule below, is more permissive than RFC 8032:

ALGORITHM 3.9 Ed25519 point decoding as implemented

```
def ed_decode(b: bytes) -> Point:      # len(b) == 32
    sign = b[31] >> 7
    y = (int_le(b) & (2**255 - 1)) % q     # y >= q is NOT rejected; it is reduced
    u = (y * y - 1) % q
    v = (d * y * y + 1) % q
    x = sqrt_mod_q(u * inverse(v, q))       # fail if u/v is not a square mod q
    if (x & 1) != sign:
        x = (q - x) % q                     # x = 0 stays 0 even if sign == 1
    return (x, y)
```

dcrd: dcrec/edwards/primitives.go · encodedBytesToBigIntPoint
github.com/agl/ed25519/edwards25519 · ExtendedGroupElement.FromBytes

A square root of a modulo q, when it exists, is $a^{(q+3)/8}$ or that value times $\sqrt{-1} = 2^{(q-1)/4} \bmod q$.

3.10.2 Parsing and verification

RULE 3.12 Parsing an Ed25519 public key

A byte string b parses as an Ed25519 public key if and only if $|b| = 32$, Algorithm 3.9 succeeds on b giving $A = (x, y)$, and it is not the case that $x = 0$ and the sign bit (bit 7 of $b[31]$) is 1. (dcrd represents that case internally with $x = q$ and rejects it because $x \geq q$.) A non-canonical y encoding (bits 0 to 254 encoding a value $\geq q$) is accepted and reduced.

dcrd: dcrec/edwards/pubkey.go · ParsePubKey

RULE 3.13 Parsing an Ed25519 signature

A byte string b parses as an Ed25519 signature if and only if $|b| = 64$, Algorithm 3.9 succeeds on $R_{\mathrm{enc}} = b[0:32]$, and $S = \mathrm{int}_{\mathrm{LE}}(b[32:64])$ satisfies $0 < S < \ell$. The signature retains the 32 bytes R_{enc} exactly as given.

dcrd: dcrec/edwards/signature.go · parseSig, ParseSignature

RULE 3.14 Ed25519 verification

A parsed signature (R_{enc}, S) is valid for a message m and a parsed public key A if and only if

$$\mathrm{encode}(SB - kA) = R_{\mathrm{enc}}, \quad k = \mathrm{int}_{\mathrm{LE}}(\text{SHA-512}(R_{\mathrm{enc}} \parallel A_{\mathrm{enc}} \parallel m)) \bmod \ell, \tag{3.2}$$

where A_{enc} is the **canonical** encoding of A (Section 3.10.1), not necessarily the bytes from which A was parsed, encode is the canonical encoding, and the comparison is of all 32 bytes. In consensus m is the 32-byte signature hash.

dcrd: dcrec/edwards/ecdsa.go · Verify | github.com/agl/ed25519 · Verify

The equation is the cofactorless one: it is not multiplied by 8. Since the left side is canonical, a non-canonical R_{enc} never verifies. No condition is placed on the order of A or R: small-order and mixed-order points are accepted if the equation holds.

Table 3.14 Ed25519 as implemented compared with RFC 8032

Aspect	RFC 8032	dcrd
Public key with $y \geq q$	Decoding fails	y reduced modulo q; accepted if the reduced value decodes, and the canonical re-encoding is hashed
Public key with $x = 0$, sign bit 1	Decoding fails	Rejected
R with $y \geq q$	Decoding fails	Parses if the reduced value decodes; never verifies
R with $x = 0$, sign bit 1	Decoding fails	Parses, never verifies
$S = 0$	Allowed	Rejected
$S \geq \ell$	Rejected	Rejected
Verification equation	$[8]SB = [8]R + [8]kA$ (the unmultiplied form is permitted)	$SB - kA$ re-encoded and compared with R_{enc} (cofactorless)
Message	Arbitrary	The 32-byte signature hash (pure Ed25519, no context or prehash)

Signing from a 32-byte secret s_{seed} follows RFC 8032 exactly: $h = \text{SHA-512}(s_{\text{seed}})$, the scalar a is $\text{int}_{\text{LE}}(h[0:32])$ with bits 0, 1, 2 and 255 cleared and bit 254 set, $A = aB$, $r = \text{int}_{\text{LE}}(\text{SHA-512}(h[32:64] \parallel m)) \bmod \ell$, $R = rB$, and $S = (r + ka) \bmod \ell$ with k as in Rule 3.14. (The package also has a mode for keys given as a bare scalar, which signs with an RFC 6979 nonce; dcrd's transaction signing code in `txscript/sign` does not use it. Its signatures are verified by the same rules.)

TEST VECTOR Ed25519

```
seed  232b277c698f23760d3bb922ffd92c47f80db87382f567de705c1c96108a6818
        (= BLAKE-256("Decred crypto chapter ed25519 seed"))
A     4fb3bab83ba936b65d6e5aeefe6433aae325b78121f18401aaadda5090a7465e
m     5a029fb117f154fc26caa93ca613772e2795e5776e2fd4208abbf6386be52dc7
sig   86e6c6ce81e30f209948be1bba9e1313ca78355b428206af9341dea5cc57a8c9
      ee1c6b29a2c9a472591759f35ea53e1700c6d773722bcb79a0493d273c541d0b
```

Parsing edge cases: the public key `01` followed by 31 zero bytes (the identity) parses; the same with the last byte `80` is rejected; the non-canonical encoding `eeff...ff7f` of $y = q + 1$ parses and re-encodes as `01` followed by 31 zero bytes; `02` followed by 31 zero bytes ($y = 2$) is not on the curve. As R_{enc}, the identity with the sign bit set and the non-canonical encoding of $y = q + 1$ both parse (and cannot verify); a signature with $S = 0$ or $S = \ell$ is rejected.

3.11 Merkle Trees

A Merkle tree [21] commits to an ordered list of 32-byte **leaves** with a single 32-byte **root**. Each interior node is the BLAKE-256 hash of the concatenation of its two children. When a level has an odd number of nodes, its last node is paired with itself. The block header commits to the transactions of a block with Merkle roots whose leaves are transaction hashes; which hashes are used, and how the regular and stake tree roots are combined under DCP-0005, is specified in Chapter 4 and Chapter 10. This section defines the generic tree.

RULE 3.15 Merkle root
The Merkle root of a list of leaves **MUST** be computed by Algorithm 3.10. In particular, the root of the empty list is 32 zero bytes, the root of a single leaf is that leaf itself (it is not hashed), and at each level with an odd number of nodes greater than one the last node is duplicated before pairing.

dcrd: blockchain/standalone/merkle.go · CalcMerkleRoot, CalcMerkleRootInPlace

ALGORITHM 3.10 Merkle root

```
def merkle_root(leaves: list[bytes]) -> bytes:
    if len(leaves) == 0:
        return bytes(32)
    level = list(leaves)
    while len(level) > 1:
        if len(level) % 2 == 1:
            level.append(level[-1])
        level = [blake256(level[2 * i] + level[2 * i + 1])
                 for i in range(len(level) // 2)]
    return level[0]
```

dcrd: blockchain/standalone/merkle.go · CalcMerkleRootInPlace

IMPLEMENTATION PITFALL
Because the last node of an odd level is duplicated, the lists $[L_0, L_1, L_2]$ and $[L_0, L_1, L_2, L_2]$ have the same root, and in general appending copies of trailing leaves can preserve the root. A Merkle root therefore does not by itself determine the number of leaves. The rule that makes this harmless for blocks (no duplicate transactions) is part of Chapter 10.

TEST VECTOR Merkle roots
Leaves $L_i = \text{BLAKE-256}(i)$ for the single byte i:

```
L0  0ce8d4ef4dd7cd8d62dfded9d4edb0a774ae6a41929a74da23109e8f11139c87
L1  4a6c419a1e25c85327115c4ace586decddfe2990ed8f3d4d801871158338501d
L2  49af37ab5270015fe25276ea5a3bb159d852943df23919522a202205fb7d175c
L3  b706d561742ad3671703c247eb927ee8a386369c79644131cdeb2c5c26bf6c5d
L4  4c6eb9e38415034f4c93d3304d10bef38bf0ad420eefd0f72f940f11c5857786

root([])       0000000000000000000000000000000000000000000000000000000000000000
root([L0])     0ce8d4ef4dd7cd8d62dfded9d4edb0a774ae6a41929a74da23109e8f11139c87
root([L0..L1]) ad204769bc6f9a916299fe1af8d54828b7403e2dbe65c8888e16e9fa1deae0c5
root([L0..L2]) 0135cd60725091a8795118c7ed50b8bf8a272707938a9fa7b24f0399bbe4a048
root([L0..L3]) f71e10d28e0d14e884e043114c2e30cfbed5ef375faf4fd9790b122a0e574e0b
root([L0..L4]) 1a3f528d95b03c38767aaf2daab6ee3a97c3040adfffbb67ecb357d418b34517
```

`root([L0..L2])` $= H(H(L_0 \parallel L_1) \parallel H(L_2 \parallel L_2))$, and it equals the root of `[L0, L1, L2, L2]`.

3.11.1 Inclusion proofs

An **inclusion proof** for the leaf at zero-based index i is the list of sibling hashes on the path from that leaf to the root, ordered from the leaf level upward. Together with the leaf and its index it suffices to recompute the root. dcrd generates such proofs for the header commitments of DCP-0005, which it serves to light clients (Chapter 16); they are not used in block validation. The proof for a tree of $N \geq 1$ leaves has $\lceil \log_2 N \rceil$ entries, and none for a single leaf.

ALGORITHM 3.11 Inclusion proof generation

```
def inclusion_proof(leaves: list[bytes], index: uint32) -> list[bytes] | None:
    if len(leaves) == 0 or index >= len(leaves):
        return None
    level = list(leaves)
    proof = []
```

```
    while len(level) > 1:
        if len(level) % 2 == 1:
            level.append(level[-1])
        sibling = index ^ 1                 # left if index is odd, else right
        proof.append(level[sibling])
        level = [blake256(level[2 * j] + level[2 * j + 1])
                 for j in range(len(level) // 2)]
        index = index >> 1
    return proof
```

dcrd: blockchain/standalone/inclusionproof.go · GenerateInclusionProof

ALGORITHM 3.12 **Inclusion proof verification**

```
def verify_inclusion(root: bytes, leaf: bytes, index: uint32,
                     proof: list[bytes]) -> bool:
    if len(proof) > 32:
        return False
    if index > 2**len(proof) - 1:           # largest index this proof length covers
        return False
    h = leaf
    for sibling in proof:
        if index & 1 == 1:
            h = blake256(sibling + h)
        else:
            h = blake256(h + sibling)
        index = index >> 1
    return h == root
```

dcrd: blockchain/standalone/inclusionproof.go · VerifyInclusionProof

Verification does not learn the number of leaves. A proof for the last leaf of an odd level contains a copy of that leaf as its own sibling.

TEST VECTOR Inclusion proofs for the five leaves above

```
index 2:
  [0] b706d561742ad3671703c247eb927ee8a386369c79644131cdeb2c5c26bf6c5d
  [1] ad204769bc6f9a916299fe1af8d54828b7403e2dbe65c8888e16e9fa1deae0c5
  [2] 92ad3df5fcd26cbbd6ea00363893dd6f201516f3760bdfa9f9359945369beacd
index 4:
  [0] 4c6eb9e38415034f4c93d3304d10bef38bf0ad420eefd0f72f940f11c5857786
  [1] 72b0e5fb43e19eb8b640aec409532c2cd8a583cfb5ada680f8664a6b53c36558
  [2] f71e10d28e0d14e884e043114c2e30cfbed5ef375faf4fd9790b122a0e574e0b
```

For index 2 the entries are L_3, $H(L_0 \parallel L_1)$ and the hash of the duplicated right subtree $H(H(L_4 \parallel L_4) \parallel H(L_4 \parallel L_4))$; for index 4 they are L_4 itself (its own duplicate), $H(L_4 \parallel L_4)$ (the duplicate of its parent), and the root of $L_0, \ldots, L_3$. Both verify against `root([L0..L4])`. The index 4 proof presented with index 8 is rejected, since 8 exceeds $2^3 - 1$.

3.12 Other Primitives

No other cryptographic primitive is evaluated in block validation outside the chapters that own it:

- The pseudorandom number generator that selects winning tickets is built on BLAKE-256 and is specified in Chapter 8.
- SipHash and Golomb-coded sets, used by the version 2 compact filters committed to by header commitments, are specified in Chapter 16.

- Base58Check (whose checksum is the first four bytes of HASH256 of the payload) and hierarchical deterministic keys are specified in Chapter 5; neither is consensus-critical.
- The signature hash that each signature commits to is specified in Chapter 6.

CHAPTER 4

Data Encoding and Structures

Every object that the consensus rules talk about, a transaction, a block header, a block, is exchanged and hashed as a byte string. This chapter specifies those byte strings exactly: the primitive encodings they are built from, the layout of transactions in each of their three serialization types, the 180-byte block header, the block, and the identifiers (hashes) derived from each of them. It also specifies what the two Merkle-root fields of the header contain, before and after the change made by DCP-0005.

The generic Merkle tree construction and the hash functions themselves belong to Chapter 3. Scripts carried inside transactions are opaque byte strings here and are interpreted in Chapter 6. The rules that decide whether a decoded transaction or block is *valid* are in Chapter 7 and Chapter 10. The framing of these structures inside peer-to-peer messages is in Chapter 15.

4.1 Decoding Principles

All structures are concatenations of fields with no padding, no alignment and no field tags. Integers are little-endian (Section 2.4) unless stated otherwise. A structure carries no overall length prefix: its extent is determined by decoding it field by field from its first byte.

dcrd computes every identifier and every size from the *decoded* structure, by serializing it again, never from the bytes it received. The encoding is designed so that this makes no difference: the decoder accepts only canonical encodings, so every byte string that decodes successfully re-encodes to exactly the same byte string, and two different accepted byte strings never decode to the same structure. An implementation may therefore hash either the received bytes or a re-serialization, *provided* it rejects everything that dcrd's decoder rejects. The rules in this chapter that use the word "undecodable" list every such rejection.

> **RULE 4.1 Decodability**
> A block **MUST** be decodable according to Section 4.6, and every transaction it contains **MUST** be decodable according to Section 4.3.7. A byte string that is undecodable under any rule of this chapter is not a block (or transaction) at all, and no block can be built on it. In particular, a structure that ends before all of its fields have been read is undecodable.
>
> `dcrd: wire/msgblock.go · MsgBlock.BtcDecode | wire/msgtx.go · MsgTx.BtcDecode`

The serialization used on the wire is the same as the one dcrd uses for storage and for hashing. dcrd's encoder and decoder accept a protocol version argument; the only effect of that argument on any structure in this chapter is the transaction-count bound of Rule 4.10, which never decides the validity of a block (see the note after that rule).

4.2 Primitive Encodings

4.2.1 Fixed-Width Integers

Fixed-width integers occupy exactly their width in bytes and are serialized least significant byte first. Signed integers are serialized as the two's complement bit pattern of the same width, so an `int64` field is written

as `le64(uint64(x))` and read back as `int64(read_le64())`. The decoder places no range restriction on any integer field: every bit pattern decodes, and range checks (for example on amounts) are validation rules stated elsewhere.

Table 4.1 Fixed-width field types

Type	Bytes	Encoding
`uint8`	1	The byte itself.
`uint16`	2	`le16(x)`
`uint32`	4	`le32(x)`
`int32`	4	`le32(uint32(x))`
`uint64`	8	`le64(x)`
`int64`	8	`le64(uint64(x))`
`[n]byte`	n	The n bytes in order, unaltered. Used for hashes ($n = 32$), the header's final state ($n = 6$) and extra data ($n = 32$).

4.2.2 Variable-Length Integers

Counts and lengths are encoded as **variable-length integers** (*varints*), the same scheme as Bitcoin's "CompactSize". The first byte, the **discriminant**, either is the value itself or announces a 2-, 4- or 8-byte little-endian value that follows. The scheme can represent every `uint64`.

Table 4.2 Variable-length integer encoding

Value v	Bytes	Discriminant	Encoding
$0 \le v \le 252$	1	v	`u8(v)`
$253 \le v \le 2^{16} - 1$	3	`0xfd`	`0xfd || le16(v)`
$2^{16} \le v \le 2^{32} - 1$	5	`0xfe`	`0xfe || le32(v)`
$2^{32} \le v \le 2^{64} - 1$	9	`0xff`	`0xff || le64(v)`

The same value could be written in more than one of these forms, for example 5 as `05` or as `fd0500`. Only the shortest form is accepted.

RULE 4.2 Canonical variable-length integers

A varint is undecodable if its value could have been encoded in fewer bytes. Precisely, after reading the discriminant d and the value v that follows it:

- if $d = 0$xfd and $v < 0$xfd (that is, $v \le 252$), reject;
- if $d = 0$xfe and $v < 0$x10000, reject;
- if $d = 0$xff and $v < 0$x100000000, reject.

A discriminant below `0xfd` is always canonical. A varint whose following 2, 4 or 8 bytes are not all present is undecodable (Rule 4.1).

dcrd: wire/common.go · ReadVarInt

ALGORITHM 4.1 Variable-length integer encoding and decoding

```
def encode_varint(v: uint64) -> bytes:
    if v < 0xfd:
        return u8(v)
    if v <= 0xffff:
        return u8(0xfd) + le16(v)
    if v <= 0xffffffff:
        return u8(0xfe) + le32(v)
    return u8(0xff) + le64(v)
```

```
def decode_varint(r) -> uint64:
    # r.read_*() reject("truncated") when too few bytes remain
    d = r.read_u8()
    if d == 0xff:
        v = r.read_le64()
        if v < 0x100000000:
            reject("non-canonical varint")
    elif d == 0xfe:
        v = r.read_le32()
        if v < 0x10000:
            reject("non-canonical varint")
    elif d == 0xfd:
        v = r.read_le16()
        if v < 0xfd:
            reject("non-canonical varint")
    else:
        v = d
    return v

def varint_size(v: uint64) -> int:
    return 1 if v < 0xfd else 3 if v <= 0xffff else 5 if v <= 0xffffffff else 9
```

dcrd: wire/common.go · WriteVarInt
wire/common.go · ReadVarInt
wire/common.go · VarIntSerializeSize

> **NOTE**
> DCP-0005 adopts the same encoding, with the same canonical requirement, for the item count of compact block filters (Chapter 16).

4.2.3 Variable-Length Byte Strings

A **variable-length byte string** (*varbytes*) is a varint length n followed by n bytes. Scripts are the only varbytes in consensus structures. Each use site imposes a maximum on n, checked *after* the varint has been decoded and *before* the bytes are read; a length above the maximum makes the structure undecodable even if the input would contain that many bytes. The maxima for scripts are given in Rule 4.7.

A **variable-length string** has the same encoding with the bytes interpreted as text. Strings occur only in peer-to-peer messages (Chapter 15); dcrd bounds their length by `MaxMessagePayload` and, for some fields, additionally requires printable ASCII (`0x20` to `0x7e`).

4.2.4 Hashes

A hash is serialized as its 32 bytes in the order the hash function produced them, with no length prefix. All hashes in consensus structures are BLAKE-256 digests (Chapter 3), written $\mathrm{BLAKE256}(x)$ below, with the single exception of the proof-of-work hash of Section 4.4.2. When a hash is shown to people it is displayed with its bytes reversed, as described in Section 2.4. Every byte string in this chapter is in serialized (internal) order unless it is explicitly labelled as display form.

4.3 Transactions

A Decred transaction is split into two parts. The **prefix** holds everything that determines what the transaction does: the outpoints it spends, the outputs it creates, its lock time and its expiry. The **witness** holds, for each input, the signature script that authorizes the spend together with three fields of fraud-proof data (the amount, block height and block index of the output being spent). The transaction's identifier is the hash of the prefix alone, so altering a signature script, which third parties can sometimes do without invalidating it, cannot change the identifier that later transactions use to refer to it.

A transaction can be serialized in three ways: in full (prefix followed by witness), as the prefix alone, or as the witness alone. Which one is used is recorded in the transaction's first four bytes.

Table 4.3 In-memory transaction structure

Field	Type	Meaning
`version`	`uint16`	Transaction version (Table 4.5).
`ser_type`	`uint16`	Serialization type the transaction was or will be encoded with (Table 4.4).
`inputs`	list of TxIn	Each with prefix fields and witness fields (Table 4.8, Table 4.9).
`outputs`	list of TxOut	Table 4.11.
`lock_time`	`uint32`	Earliest height or time at which the transaction may be included (Chapter 7).
`expiry`	`uint32`	Height from which the transaction may no longer be included; `0` means no expiry (Chapter 7).

4.3.1 The Version Word and Serialization Types

Every serialized transaction begins with a 32-bit little-endian **version word**. Its low 16 bits are the transaction version; its high 16 bits are the serialization type:

$$\text{word} = \text{version} + 2^{16} \cdot \text{ser_type}. \tag{4.1}$$

Table 4.4 Transaction serialization types

`ser_type`	dcrd name	Body after the version word	Use
`0`	`TxSerializeFull`	prefix body, then witness body	The normal encoding: in blocks, in `tx` messages, in storage.
`1`	`TxSerializeNoWitness`	prefix body only	Input to the transaction hash (Section 4.3.8).
`2`	`TxSerializeOnlyWitness`	witness body only	Input to the witness hash (Section 4.3.8).

RULE 4.3 Serialization type
A transaction whose serialization type (the high 16 bits of the version word) is not `0`, `1` or `2` is undecodable.

dcrd: wire/msgtx.go · MsgTx.BtcDecode

The version (low 16 bits) is not restricted by the decoder. The versions with a defined meaning are listed below; which versions are permitted in which context is a validation rule (Chapter 7, Chapter 8, Chapter 12).

Table 4.5 Transaction version constants

Constant	Value	Meaning
`TxVersion`	1	Initial transaction version.
`TxVersionSeqLock`	2	Enables relative lock times through the sequence field.
`TxVersionAutoRevocations`	2	Version of revocations under DCP-0009.
`TxVersionTreasury`	3	Treasury transactions (DCP-0006).

4.3.2 Outpoints

An **outpoint** identifies one output of an earlier transaction. Unlike Bitcoin's, a Decred outpoint also names the **transaction tree** in which the referenced transaction appears, because a block carries regular and stake transactions in two separate lists (Section 4.6).

Table 4.6 Outpoint (37 bytes)

Field	Type	Bytes	Description
`Hash`	`[32]byte`	32	Transaction hash (Section 4.3.8) of the transaction that created the output.
`Index`	`uint32`	4	Zero-based index of the output within that transaction's outputs.

Field	Type	Bytes	Description
Tree	int8	1	Tree of the transaction that created the output: 0 regular, 1 stake (Table 4.7).

Table 4.7 Transaction tree identifiers

Constant	Value	Meaning
TxTreeRegular	0	The regular transaction tree: ordinary transfers and the coinbase.
TxTreeStake	1	The stake transaction tree: tickets, votes, revocations and treasury transactions.
TxTreeUnknown	−1	In-memory marker for "not in a block"; never serialized.

RULE 4.4 Outpoint tree byte

The tree byte is read as an unsigned byte and reinterpreted as int8. If the result is negative, that is, if the byte is 0x80 or greater, the transaction is undecodable.

dcrd: wire/msgtx.go · ReadOutPoint

Tree values 2 to 127 decode successfully. dcrd identifies unspent outputs by the full triple (hash, index, tree), and every output it records has tree 0 or 1, according to the type of the transaction that created it (Chapter 7). An outpoint with any other tree value therefore never refers to an existing output, and an input carrying one is rejected by the ordinary missing-input rule, or, for an input that is required to be null, by the null-outpoint requirement (which includes tree 0), not by the decoder.

A **null outpoint** has the all-zero hash, index MaxPrevOutIndex = 0xffffffff and tree 0. It marks inputs that spend nothing, such as those of the coinbase, the stakebase input of a vote, the treasurybase and treasury spends; which of these require a null outpoint, and which components are checked, is specified with each transaction type.

4.3.3 Transaction Inputs

Each input has fields in the prefix and fields in the witness. The two halves are paired by position: the i-th witness entry belongs to the i-th prefix entry.

Table 4.8 Transaction input, prefix part (41 bytes)

Field	Type	Bytes	Description
PreviousOutPoint	Outpoint	37	The output being spent (Table 4.6).
Sequence	uint32	4	Sequence number. MaxTxInSequenceNum = 0xffffffff is the default. For transactions of version 2 or higher it can encode a relative lock time (Table 4.10; semantics in Chapter 7).

Table 4.9 Transaction input, witness part (16 bytes plus the script)

Field	Type	Bytes	Description
ValueIn	int64	8	Amount in atoms of the output being spent, committed by the spender. NullValueIn = −1 is the conventional "not given" value.
BlockHeight	uint32	4	Height of the block containing the output being spent. NullBlockHeight = 0 is the null value.
BlockIndex	uint32	4	Zero-based position, within its tree, of the transaction that created the output being spent. NullBlockIndex = 0xffffffff is the null value.
SignatureScript	varbytes	var	Script that satisfies the spent output's public key script (Chapter 6).

dcrd calls the three numeric witness fields **fraud-proof** data: they repeat, inside the spending transaction, the amount and the location of the output being spent. For inputs that spend an existing output, consensus requires them to match the output exactly; for inputs that spend nothing, specific transaction types require the null values. Both requirements are specified in Chapter 7. Note that "null" for BlockHeight is 0, a valid height, while "null" for BlockIndex is 0xffffffff.

Table 4.10 Sequence-field constants (semantics in Chapter 7)

Constant	Value	Meaning
`MaxTxInSequenceNum`	`0xffffffff`	Maximum and default sequence number.
`SequenceLockTimeDisabled`	`1 << 31`	If set, the sequence number is not a relative lock time.
`SequenceLockTimeIsSeconds`	`1 << 22`	If set, the relative lock time counts units of 512 seconds; otherwise blocks.
`SequenceLockTimeMask`	`0x0000ffff`	Mask that extracts the relative lock time value.
`SequenceLockTimeGranularity`	`9`	Seconds-based lock times are shifted right by this amount ($2^9 = 512$).

4.3.4 Transaction Outputs

Table 4.11 Transaction output (10 bytes plus the script)

Field	Type	Bytes	Description
`Value`	`int64`	8	Amount in atoms (Section 4.7).
`Version`	`uint16`	2	Script version of `PkScript`. `DefaultPkScriptVersion` $= 0$ (Chapter 6).
`PkScript`	`varbytes`	var	Public key script: the conditions for spending this output (Chapter 6).

4.3.5 Lock Time and Expiry

The prefix ends with two `uint32` fields. `LockTime` is interpreted as a block height or a Unix time depending on its magnitude, and `Expiry` is a block height, with `NoExpiryValue` $= 0$ meaning that the transaction never expires. Their semantics are given in Chapter 7. A witness-only serialization does not carry them.

4.3.6 Serialized Layouts

The **prefix body** and **witness body** are defined as follows; each serialization type is the version word followed by one or both bodies (Table 4.4).

Table 4.12 Prefix body

Field	Type	Bytes	Description
`TxInCount`	`varint`	1–9	Number of inputs n.
`TxIn[i]`	`TxIn prefix`	$41n$	Table 4.8, for $i = 0, \ldots, n-1$.
`TxOutCount`	`varint`	1–9	Number of outputs m.
`TxOut[j]`	`TxOut`	var	Table 4.11, for $j = 0, \ldots, m-1$.
`LockTime`	`uint32`	4	
`Expiry`	`uint32`	4	

Table 4.13 Witness body

Field	Type	Bytes	Description
`WitnessCount`	`varint`	1–9	Number of witness entries. In a full serialization it equals the input count n (Rule 4.6).
`TxIn[i]`	`TxIn witness`	var	Table 4.9, in input order.

ALGORITHM 4.2 **Transaction serialization**

```
FULL, NO_WITNESS, ONLY_WITNESS = 0, 1, 2

def encode_varbytes(b: bytes) -> bytes:
    return encode_varint(len(b)) + b
```

```python
def prefix_body(tx) -> bytes:
    out = encode_varint(len(tx.inputs))
    for ti in tx.inputs:
        out += ti.prev.hash + le32(ti.prev.index) + u8(ti.prev.tree) + le32(ti.sequence)
    out += encode_varint(len(tx.outputs))
    for to in tx.outputs:
        out += le64(uint64(to.value)) + le16(to.version) + encode_varbytes(to.pk_script)
    return out + le32(tx.lock_time) + le32(tx.expiry)

def witness_body(tx) -> bytes:
    out = encode_varint(len(tx.inputs))
    for ti in tx.inputs:
        out += (le64(uint64(ti.value_in)) + le32(ti.block_height)
                + le32(ti.block_index) + encode_varbytes(ti.sig_script))
    return out

def serialize_tx(tx, ser_type: int) -> bytes:
    out = le32(tx.version | (ser_type << 16))
    if ser_type == FULL:
        return out + prefix_body(tx) + witness_body(tx)
    if ser_type == NO_WITNESS:
        return out + prefix_body(tx)
    if ser_type == ONLY_WITNESS:
        return out + witness_body(tx)
```

dcrd: wire/msgtx.go · MsgTx.BtcEncode
wire/msgtx.go · encodePrefix
wire/msgtx.go · encodeWitness

The serialized size of a transaction in each type follows directly. With n inputs, m outputs, signature scripts s_i and public key scripts p_j, and $V(x)$ the varint size of x:

$$\text{size}_{\text{prefix}} = 4 + V(n) + 41n + V(m) + \sum_j \big(10 + V(|p_j|) + |p_j|\big) + 8, \tag{4.2}$$

$$\text{size}_{\text{witness}} = 4 + V(n) + \sum_i \big(16 + V(|s_i|) + |s_i|\big), \tag{4.3}$$

$$\text{size}_{\text{full}} = \text{size}_{\text{prefix}} + \text{size}_{\text{witness}} - 4. \tag{4.4}$$

The size that the transaction-size rule of Chapter 7 compares with `MaxTxSize` (393216 bytes on the main network) is the size in the serialization type the transaction was decoded with.

4.3.7 Decoding

Decoding reads the version word, rejects an unknown serialization type, and then reads the bodies that the type calls for. Fields that the chosen type does not carry are set to zero or empty. In particular a prefix-only transaction has every input's `ValueIn`, `BlockHeight` and `BlockIndex` equal to 0 (not to the null values) and empty signature scripts, and a witness-only transaction has all-zero outpoints, zero sequence numbers, no outputs, and zero lock time and expiry.

The decoder bounds every count and length before allocating storage for it. The bounds are derived from `MaxMessagePayload` $= 2^{25} = 33554432$ bytes, the largest peer-to-peer message payload: a count is rejected when even the smallest possible elements could not fit into one message.

Table 4.14 Transaction decoding bounds

Bound	Value	Derivation
Inputs (prefix or witness count)	780336	$\lfloor 33554432/43 \rfloor + 1$, where $43 = 32 + 4 + 1 + 1 + 4 + 1$ is dcrd's `minTxInPayload` (outpoint, a one-byte script-length varint and the sequence, plus one byte).

Bound	Value	Derivation
Outputs	3728271	$\lfloor 33554432/9 \rfloor + 1$, where $9 = 8 + 1$ is `minTxOutPayload` (value and a one-byte script-length varint).
Signature script length	33554432	`MaxMessagePayload`.
Public key script length	33554432	`MaxMessagePayload`.

RULE 4.5 Transaction element counts

A transaction is undecodable if its input count (in the prefix body, or in the witness body of a witness-only serialization) exceeds 780336, or if its output count exceeds 3728271. Each count is compared after it has been decoded as a varint and before any element is read.

dcrd: wire/msgtx.go · MsgTx.decodePrefix | wire/msgtx.go · MsgTx.decodeWitness

RULE 4.6 Witness count of a full serialization

In a transaction of serialization type `0` (full), the witness count **MUST** equal the number of inputs read from the prefix body; otherwise the transaction is undecodable.

dcrd: wire/msgtx.go · MsgTx.decodeWitness

RULE 4.7 Script lengths

A signature script or public key script whose length varint exceeds 33554432 makes the transaction undecodable. The comparison is made before the script bytes are read.

dcrd: wire/msgtx.go · readScript | wire/msgtx.go · readTxInWitness | wire/msgtx.go · readTxOut

ALGORITHM 4.3 Transaction decoding: constants and helpers

```
MAX_MESSAGE_PAYLOAD = 33554432
MAX_TX_IN  = MAX_MESSAGE_PAYLOAD // 43 + 1      # 780336
MAX_TX_OUT = MAX_MESSAGE_PAYLOAD // 9 + 1       # 3728271

def decode_varbytes(r, max_len: int) -> bytes:
    n = decode_varint(r)
    if n > max_len:
        reject("script too long")
    return r.read(n)                            # reject("truncated") if short

def read_witness_fields(r, ti):
    ti.value_in = int64(r.read_le64())
    ti.block_height = r.read_le32()
    ti.block_index = r.read_le32()
    ti.sig_script = decode_varbytes(r, MAX_MESSAGE_PAYLOAD)
```

dcrd: wire/msgtx.go · readScript | wire/msgtx.go · readTxInWitness

ALGORITHM 4.4 Transaction decoding

```
def decode_tx(r) -> Tx:
    word = r.read_le32()
    tx = Tx(version=word & 0xffff, ser_type=word >> 16,
            inputs=[], outputs=[], lock_time=0, expiry=0)
    if tx.ser_type not in (FULL, NO_WITNESS, ONLY_WITNESS):
        reject("unknown serialization type")

    if tx.ser_type in (FULL, NO_WITNESS):
        n = decode_varint(r)
        if n > MAX_TX_IN:
            reject("too many inputs")
```

```
        for _ in range(n):
            ti = TxIn()                          # all witness fields zero / empty
            ti.prev.hash = r.read(32)
            ti.prev.index = r.read_le32()
            tree = r.read_u8()
            if tree >= 0x80:
                reject("negative transaction tree")
            ti.prev.tree = tree
            ti.sequence = r.read_le32()
            tx.inputs.append(ti)
        m = decode_varint(r)
        if m > MAX_TX_OUT:
            reject("too many outputs")
        for _ in range(m):
            to = TxOut()
            to.value = int64(r.read_le64())
            to.version = r.read_le16()
            to.pk_script = decode_varbytes(r, MAX_MESSAGE_PAYLOAD)
            tx.outputs.append(to)
        tx.lock_time = r.read_le32()
        tx.expiry = r.read_le32()

    if tx.ser_type == FULL:
        k = decode_varint(r)
        if k != len(tx.inputs):
            reject("witness count differs from input count")
        for ti in tx.inputs:
            read_witness_fields(r, ti)
    elif tx.ser_type == ONLY_WITNESS:
        k = decode_varint(r)
        if k > MAX_TX_IN:
            reject("too many inputs")
        for _ in range(k):
            ti = TxIn()                          # all prefix fields zero
            read_witness_fields(r, ti)
            tx.inputs.append(ti)
    return tx
```

dcrd: wire/msgtx.go · MsgTx.BtcDecode | wire/msgtx.go · ReadOutPoint | wire/msgtx.go · readTxOut

NOTE

None of the bounds in Table 4.14 can be reached by a transaction inside a block that respects the block size limit of Chapter 10 (at most `MaxBlockPayload` = 1310720 bytes): an input prefix alone occupies 41 bytes and an output at least 11. They matter for robustness, not for which blocks are valid, but an implementation that enforces tighter bounds at decoding time must make sure they too can never bind on a block within the size limit.

4.3.8 Transaction Identifiers

Three hashes are defined for every transaction. Each is computed from the decoded transaction, independently of the serialization type it was received in.

ALGORITHM 4.5 Transaction hashes

```
def tx_hash(tx) -> bytes:                # "transaction hash", "txid"
    return blake256(serialize_tx(tx, NO_WITNESS))

def tx_witness_hash(tx) -> bytes:
    return blake256(serialize_tx(tx, ONLY_WITNESS))
```

```
def tx_full_hash(tx) -> bytes:
    return blake256(tx_hash(tx) + tx_witness_hash(tx))
```

dcrd: wire/msgtx.go · MsgTx.TxHash
wire/msgtx.go · MsgTx.TxHashWitness
wire/msgtx.go · MsgTx.TxHashFull

Table 4.15 Where each transaction hash is used

Hash	Used for
Transaction hash (`tx_hash`)	The transaction's identity: the `Hash` field of outpoints that spend its outputs, the identity of tickets in the ticket pool and lottery, duplicate-transaction checks, display to users. It does not commit to any witness field.
Witness hash (`tx_witness_hash`)	Only as an input to the full hash.
Full hash (`tx_full_hash`)	The leaves of the transaction-tree Merkle trees (Section 4.5), so that the block header commits to the witnesses as well as the prefixes.

IMPLEMENTATION PITFALL

- The transaction hash is the hash of the *prefix serialization*, which begins with the version word carrying serialization type `1`. It is therefore **not** the hash of any substring of the full serialization, whose version word carries type `0`. For a version-1 transaction the prefix serialization begins `01000100`, the full serialization `01000000`.
- Likewise the witness serialization begins with type `2` in its version word and repeats the input count.
- The full hash is the hash of the 64-byte concatenation of the two other hashes, **not** the hash of the full serialization.
- Signature hashing (Chapter 6) defines its own serializations. Its prefix component coincides with the transaction hash for the signature hash type `SigHashAll` without `SigHashAnyOneCanPay` (dcrd reuses the cached transaction hash in that case); its witness component is not the witness hash defined here.

RATIONALE

Including the serialization type in the hashed version word guarantees that the prefix and witness hashes of a transaction differ even when the two bodies would otherwise be identical byte strings (code comment in `wire/msgtx.go · MsgTx.TxHashFull`). Excluding the witness from the identifier makes the identifier immune to signature-script malleability.

4.3.9 Transactions in Blocks

Each transaction in a block is decoded with the serialization type given in its own version word. Blocks produced by dcrd contain only full (type `0`) transactions, and dcrd's mempool policy refuses any other type, but the block format and the consensus code do not themselves name the serialization type.

IMPLEMENTATION PITFALL

dcrd has no consensus rule that names the serialization type of a transaction in a block. A block containing a type-`1` or type-`2` transaction is decoded as specified above (Rule 4.1 does not reject it), its size (header `Size`, Rule 4.11) counts the transaction's actual, shorter encoding, its Merkle leaf is the full hash of the transaction *as decoded* (with zeroed witness or zeroed prefix fields), and the transaction is then subject to every ordinary validation rule using those decoded values. A header can therefore match such a block in `Size` and in its Merkle roots. The block is nevertheless always invalid, for the following reasons:

- A type-`2` transaction has no outputs and fails the requirement that every transaction have at least one output (Chapter 7).
- A type-`1` transaction has `ValueIn` $= 0$, `BlockHeight` $= 0$, `BlockIndex` $= 0$ and an empty signature script on every input, and it has at least one input. An input that spends an existing output fails the fraud-proof rule (`ValueIn` must equal the output's amount, and outputs of amount 0 cannot be spent at all). The null input of a coinbase, treasurybase or treasury spend fails the null-fraud-proof rule (`BlockIndex`

must be `0xffffffff`). A vote's stakebase input fails the same rule, so the transaction is not recognized as a vote and is then rejected for its null outpoint; in any case its ticket input is an input of the first kind.

Running dcrd's validation on blocks whose transactions were re-encoded in type `1` produced, respectively, a fraud-proof amount error (regular transaction and ticket), a non-null coinbase fraud proof error (coinbase) and a null-previous-output error (vote); type `2` produced the no-outputs error. Because the Merkle leaf is computed from the decoded fields, a full-type transaction with the same field values has the same leaf and is invalid for the same reason, so the header itself commits to invalid data. An implementation may therefore reject a block containing a non-full transaction once the header's commitments to it have been checked, but must not validate the transaction as though the missing fields were present.

4.4 The Block Header

The block header is a fixed 180-byte structure (`MaxBlockHeaderPayload` $= 84 + 3 \cdot 32 = 180$). Every field has a fixed width, so the header has exactly one serialization. Besides the usual chain link, Merkle root, difficulty, time and nonce, it carries a summary of the block's stake activity, which allows several stake rules to be checked from headers alone.

Table 4.16 Block header (180 bytes). The offset of each field is given first.

Field	Type	Bytes	Description
`Version`	`int32`	4	Offset 0. Block version (Chapter 13); not the protocol version.
`PrevBlock`	`[32]byte`	32	Offset 4. Block hash (Section 4.4.1) of the parent block; all zero in the genesis block. This is always the BLAKE-256 block hash, never the proof-of-work hash.
`MerkleRoot`	`[32]byte`	32	Offset 36. Commitment to the transactions: the regular-tree Merkle root, or the combined root after DCP-0005 (Section 4.5).
`StakeRoot`	`[32]byte`	32	Offset 68. The stake-tree Merkle root, or the header commitment root after DCP-0005 (Section 4.5).
`VoteBits`	`uint16`	2	Offset 100. Bit 0 (`0x0001`) set means the regular transaction tree of the parent block is approved by this block's votes; its consistency with the votes is specified in Chapter 8 and its effect in Chapter 10. The remaining bits are described in Chapter 13.
`FinalState`	`[6]byte`	6	Offset 102. Final state of the ticket-lottery pseudorandom generator (Chapter 8).
`Voters`	`uint16`	2	Offset 108. Number of votes in the block.
`FreshStake`	`uint8`	1	Offset 110. Number of ticket purchases in the block.
`Revocations`	`uint8`	1	Offset 111. Number of revocations in the block.
`PoolSize`	`uint32`	4	Offset 112. Size of the live ticket pool (Chapter 8).
`Bits`	`uint32`	4	Offset 116. Proof-of-work target in compact form (Chapter 9).
`SBits`	`int64`	8	Offset 120. Stake difficulty: the ticket price in atoms (Chapter 9).
`Height`	`uint32`	4	Offset 128. Height of this block.
`Size`	`uint32`	4	Offset 132. Length in bytes of the serialized block (Rule 4.11).
`Timestamp`	`uint32`	4	Offset 136. Block time, Unix seconds.
`Nonce`	`uint32`	4	Offset 140. Proof-of-work nonce.
`ExtraData`	`[32]byte`	32	Offset 144. Additional nonce space for miners; no consensus meaning beyond being hashed.
`StakeVersion`	`uint32`	4	Offset 176. Stake version, determined by the versions of recent votes (Chapter 13).

ALGORITHM 4.6 Block header serialization

```
def serialize_header(h) -> bytes:           # always exactly 180 bytes
    return (le32(uint32(h.version)) + h.prev_block + h.merkle_root + h.stake_root
            + le16(h.vote_bits) + h.final_state + le16(h.voters)
```

```
            + u8(h.fresh_stake) + u8(h.revocations) + le32(h.pool_size)
            + le32(h.bits) + le64(uint64(h.sbits)) + le32(h.height)
            + le32(h.size) + le32(h.timestamp) + le32(h.nonce)
            + h.extra_data + le32(h.stake_version))
```

dcrd: wire/blockheader.go · writeBlockHeader

The decoder reads the 18 fields in this order and imposes no condition on any of them; a header is undecodable only if fewer than 180 bytes are available.

> **NOTE**
> `Nonce` and `ExtraData` together form 36 contiguous bytes (offsets 140 to 175) that the consensus rules treat as an uninterpreted blob. DCP-0011 describes them as a single 36-byte extra-data field and recommends a layout for miners and pools; dcrd's split into a 4-byte nonce and 32 further bytes is a representation convenience only.

> **NOTE**
> The timestamp is an unsigned 32-bit count of seconds, so it can represent times up to 2106-02-07. The header has no other time field.

4.4.1 The Block Hash

The **block hash**, which identifies a block and is the value stored in the next block's `PrevBlock`, is

$$\text{BlockHash} = \text{BLAKE256}(\text{serialize_header}(h)). \tag{4.5}$$

It commits to every header field, and through `MerkleRoot` (and, before DCP-0005, `StakeRoot`) to every transaction. The block hash has always been computed this way and is unaffected by DCP-0011.

4.4.2 The Proof-of-Work Hash

The **proof-of-work hash** is the value compared with the target. Its input is the same 180-byte header serialization; only the hash function differs:

- if the `blake3pow` agenda (DCP-0011) is not active for the block, the proof-of-work hash is the block hash, $\text{BLAKE256}(\text{serialize_header}(h))$;
- if it is active, the proof-of-work hash is $\text{BLAKE3}(\text{serialize_header}(h))$ with a 32-byte output.

The hash functions are specified in Chapter 3 and the target check in Chapter 9. After DCP-0011 the proof-of-work hash and the block hash of a block are different values, and only the block hash is used to identify blocks.

4.5 Transaction Trees and Merkle Roots

A block holds two ordered lists of transactions: the **regular transaction tree** and the **stake transaction tree**. ("Tree" is Decred's name for these lists; each is a flat sequence, and the word refers to the Merkle tree built over it.) Which transactions belong in which list is specified in Chapter 10 and Chapter 8.

The Merkle root of a list of transactions is the Merkle root, as defined in Chapter 3, of the list of their **full hashes** (Algorithm 4.5) in block order. That construction pairs adjacent hashes with BLAKE-256, duplicating the last hash of any level with an odd number of entries; the root of an empty list is 32 zero bytes, and the root of a one-element list is that element.

ALGORITHM 4.7 **Transaction tree roots and the combined root**

```
def tree_root(txs) -> bytes:
    # merkle_root() is the generic construction of the cryptography chapter;
    # merkle_root([]) is 32 zero bytes.
```

```
    return merkle_root([tx_full_hash(tx) for tx in txs])

def combined_root(regular_txs, stake_txs) -> bytes:
    # Equal to merkle_root([tree_root(regular_txs), tree_root(stake_txs)]).
    return blake256(tree_root(regular_txs) + tree_root(stake_txs))
```

dcrd: blockchain/standalone/merkle.go · CalcTxTreeMerkleRoot
blockchain/standalone/merkle.go · CalcCombinedTxTreeMerkleRoot

What the header's two root fields must contain depends on whether the `headercommitments` agenda (DCP-0005) is active for the block, as determined from its parent by the rules of Chapter 13.

RULE 4.8 Merkle roots before header commitments

If the `headercommitments` agenda is not active for the block, the header's `MerkleRoot` **MUST** equal `tree_root` of the regular transaction tree and its `StakeRoot` **MUST** equal `tree_root` of the stake transaction tree.

dcrd: internal/blockchain/validate.go · checkMerkleRoots
internal/blockchain/validate.go · checkMerkleRootsContext

RULE 4.9 Merkle roots after header commitments

If the `headercommitments` agenda is active for the block, the header's `MerkleRoot` **MUST** equal `combined_root` of the regular and stake transaction trees, that is, $\mathrm{BLAKE256}(R \parallel S)$ where R and S are the regular and stake tree roots. The `StakeRoot` field then holds the **header commitment root**, which **MUST** equal the version 1 commitment root of DCP-0005, which is the hash of the block's version 2 compact filter (the BLAKE-256 hash of the serialized filter data, with an all-zero hash for an empty filter). The filter and its hash are specified in Chapter 16 and the enforcement of this field in Chapter 10.

dcrd: internal/blockchain/validate.go · checkMerkleRoots
internal/blockchain/validate.go · checkConnectBlock

HISTORY

At launch the header committed to the two trees separately, in `MerkleRoot` and `StakeRoot`. DCP-0005 merged both commitments into `MerkleRoot` and repurposed `StakeRoot` as the root of a new Merkle tree of *header commitments*, whose only leaf so far is the hash of the block's compact filter, so that light clients can verify filters from headers. On the main network the agenda became active at block 431488; an implementation must nevertheless determine activation from the chain (Chapter 13).

NOTE

dcrd checks the Merkle roots twice. Before the full chain context of a block is available it determines the agenda state "positionally", from the known activation block of each network; where that is not possible (on networks without a recorded activation) it provisionally accepts a header that matches *either* variant. The contextual check made when the block is connected then requires the variant selected by the agenda state. The net consensus effect is exactly the two rules above.

IMPLEMENTATION PITFALL

The Merkle leaves are **full** hashes, which commit to the witnesses. Outpoints refer to transactions by their **transaction hash**, which does not. Using the transaction hash as the leaf yields a different root (Section 4.9.4 shows both for a real block).

4.6 Block Serialization

Table 4.17 Block

Field	Type	Bytes	Description
`Header`	`BlockHeader`	180	Table 4.16.
`TxCount`	`varint`	1–9	Number of regular transactions.
`Transactions`	`Tx[]`	var	Regular transaction tree, each transaction serialized in its own serialization type (normally full).
`STxCount`	`varint`	1–9	Number of stake transactions.
`STransactions`	`Tx[]`	var	Stake transaction tree, likewise.

ALGORITHM 4.8 **Block serialization**

```
def serialize_block(b) -> bytes:
    out = serialize_header(b.header) + encode_varint(len(b.regular))
    for tx in b.regular:
        out += serialize_tx(tx, tx.ser_type)
    out += encode_varint(len(b.stake))
    for tx in b.stake:
        out += serialize_tx(tx, tx.ser_type)
    return out
```

dcrd: wire/msgblock.go · MsgBlock.BtcEncode

Decoding reads the header, then each count followed by that many transactions decoded with Algorithm 4.4. The encoding of a block is exactly the bytes so consumed. The decoders of blocks and transactions do not themselves examine what follows the structure. When a block or a transaction is carried in a peer-to-peer message, dcrd rejects the message unless its payload consists of exactly those bytes (a payload with unconsumed trailing bytes is an error); that requirement, and the per-message size limits, are part of Chapter 15. Outside that framing, trailing bytes are simply not part of the block: dcrd's `submitblock` RPC, for example, decodes the block from the front of the submitted data and does not reject anything after it. Such bytes count neither towards the block's size nor towards its hash, although dcrd keeps the submitted buffer as the block's serialization: it is written to the block database with the trailing bytes and returned unchanged by `getblock` in raw form, while peers are sent a re-encoding without them.

RULE 4.10 **Block transaction counts**

A block is undecodable if either its regular or its stake transaction count exceeds $\lfloor \lfloor P/15 \rfloor /2 \rfloor + 1$, where 15 is the size of the smallest possible full transaction (`minTxPayload` : version word, three one-byte varints, lock time, expiry) and P is `MaxBlockPayloadV3` $= 1000000$ when decoding at peer-to-peer protocol version 3 or lower (giving 33334), and `MaxBlockPayload` $= 1310720$ at higher protocol versions (giving 43691). dcrd decodes blocks from storage and from the `submitblock` RPC at protocol version 0, and therefore with the bound 33334.

dcrd: wire/msgblock.go · MsgBlock.BtcDecode | wire/msgblock.go · MaxTxPerTxTree

NOTE

This bound depends on a transport parameter, but it never decides whether a block is valid. Every block must have a serialized size of at most 1310720 bytes (Chapter 10), and every transaction must have at least one input and one output (Chapter 7), which makes it at least 66 bytes long even in prefix-only form (see Table 4.18). Such a block holds at most $\lfloor 1310720/66 \rfloor = 19859$ transactions in total, well below 33334. Put the other way round, a block that exceeded the count within 1310720 bytes would have to contain a transaction shorter than 40 bytes, which cannot have both an input and an output. An implementation may therefore omit the bound, or apply either value, without affecting consensus.

4.6.1 Block Size

The **size** of a block is the length in bytes of `serialize_block` applied to the decoded block: header, both counts and every transaction in the serialization type it was decoded with. Since decoding is canonical this is also the number of bytes the block occupied on the wire.

> **RULE 4.11 Header size field**
> The header's `Size` field **MUST** equal the size of the block as just defined.
> `dcrd: internal/blockchain/validate.go · checkBlockDataSanity`

The limits on the size, the context-free bound of `MaxBlockPayload` = 1310720 bytes and the network-specific maximum `MaximumBlockSizes` (393216 bytes on the main network), are block validation rules specified in Chapter 10.

Table 4.18 Minimum sizes and fixed sizes

Structure	Bytes	Composition
Block header	180	Fixed.
Outpoint	37	32 + 4 + 1.
Input, prefix part	41	Outpoint + 4.
Input, witness part	≥ 17	8 + 4 + 4 + script-length varint + script; 17 with an empty script.
Output	≥ 11	8 + 2 + script-length varint + script; 11 with an empty script.
Full transaction, no inputs or outputs	15	4 + 1 + 1 + 4 + 4 + 1. Decodable but never valid.
Full transaction, 1 input, 1 output, empty scripts	84	Smallest full transaction that has an input and an output.
Prefix-only transaction, 1 input, 1 output	66	Same transaction in serialization type 1.
Witness-only transaction, 1 input	22	Same transaction in serialization type 2.
Main-network genesis block	300	Header, one 118-byte coinbase, empty stake tree.

4.6.2 The Genesis Block

Each network's genesis block is a fixed constant (Chapter 14). It is valid by definition, and none of its fields are validated. Two of its fields are inconsistent with the rules above, and an implementation must use the constant as given rather than recompute it:

- its header `Size` is 0 on every network, although the main-network genesis block serializes to 300 bytes;
- on the test network (version 3) its `MerkleRoot` is the *transaction hash* of the genesis coinbase instead of its full hash (a mistake recorded in a comment in `chaincfg/testnetparams.go · TestNet3Params`); on the other networks it is the full hash, as Rule 4.8 would require.

Changing either value would change the genesis hash and therefore every block of that network.

4.7 Amounts

Amounts are integers counting **atoms** and are carried in `int64` fields: `TxOut.Value`, `TxIn.ValueIn` and the header's `SBits`. On the wire they are eight little-endian bytes of the two's complement representation, and every bit pattern decodes. The range checks, which make negative output values, values above `MaxAmount` and negative input values other than `NullValueIn` invalid, are transaction rules (Chapter 7).

Table 4.19 Amount constants

Constant	Value	Meaning
`AtomsPerCoin`	10^8	Atoms per DCR.
`AtomsPerCent`	10^6	Atoms per hundredth of a DCR.
`MaxAmount`	$21 \cdot 10^6 \cdot 10^8 = 2100000000000000$	Largest amount permitted in any single value or sum (Chapter 7).
`NullValueIn`	-1	Conventional "not given" value of `ValueIn`.

4.8 Summary of Constants and Decoding Limits

Table 4.20 Encoding constants

Constant	Value	Meaning
`MaxVarIntPayload`	9	Longest varint encoding.
`HashSize`	32	Bytes in a hash.
`MaxBlockHeaderPayload`	180	Bytes in a block header (exact).
`MaxMessagePayload`	33554432	Largest peer-to-peer payload; basis of the transaction decoding bounds.
`MaxBlockPayloadV3`	1000000	Largest block message at protocol version 3 or lower.
`MaxBlockPayload`	1310720	Largest block message at later protocol versions; also the context-free block size limit (Chapter 10).
`MaxPrevOutIndex`	`0xffffffff`	Largest outpoint index; the index of a null outpoint.
`MaxTxInSequenceNum`	`0xffffffff`	Largest and default sequence number.
`NullValueIn`	−1	Null `ValueIn`.
`NullBlockHeight`	0	Null `BlockHeight`.
`NullBlockIndex`	`0xffffffff`	Null `BlockIndex`.
`NoExpiryValue`	0	`Expiry` value meaning “never expires”.
`DefaultPkScriptVersion`	0	Default output script version.
`TxTreeRegular`	0	Regular tree identifier.
`TxTreeStake`	1	Stake tree identifier.

Table 4.21 Every condition under which a consensus structure is undecodable

Structure	Condition	Rule
Any	Input ends before a field is complete.	Rule 4.1
Varint	Non-canonical (value fits a shorter form).	Rule 4.2
Transaction	Serialization type not 0, 1 or 2.	Rule 4.3
Transaction	Outpoint tree byte $\geq$ `0x80`.	Rule 4.4
Transaction	Input count $>$ 780336 or output count $>$ 3728271.	Rule 4.5
Transaction	Full serialization whose witness count differs from its input count.	Rule 4.6
Transaction	Script length $>$ 33554432.	Rule 4.7
Block	Regular or stake transaction count above 33334 (protocol version $\leq$ 3, and storage) or 43691 (later versions).	Rule 4.10

4.9 Test Vectors

All vectors in this section were produced by running dcrd’s `wire`, `chaincfg` and `blockchain/standalone` packages at the reference commit. Byte strings are in serialized order; hashes are given in both serialized (“internal”) and display form.

4.9.1 Variable-Length Integers

TEST VECTOR Varint encodings at every boundary

```
value                       encoding
0                           00
252          (0xfc)         fc
253          (0xfd)         fdfd00
255          (0xff)         fdff00
256          (0x100)        fd0001
```

```
65535         (0xffff)          fdffff
65536         (0x10000)         fe00000100
4294967295    (0xffffffff)      feffffffff
4294967296    (0x100000000)     ff0000000001000000
2^64 - 1                        ffffffffffffffffff
```

TEST VECTOR Varint decoding: rejected and accepted inputs

```
input                 result
fd0000                reject: non-canonical (0 < 0xfd)
fdfc00                reject: non-canonical (252 < 0xfd)
fdfd00                253
fe00000000            reject: non-canonical (0 < 0x10000)
fefffe0000            reject: non-canonical (65535 < 0x10000)
fe00000100            65536
ff0000000000000000    reject: non-canonical (0 < 0x100000000)
ffffffffff00000000    reject: non-canonical (4294967295 < 0x100000000)
ff0000000001000000    4294967296
fd00                  reject: truncated
fe000001              reject: truncated
(empty)               reject: truncated
```

4.9.2 The Main-Network Genesis Block

The main-network genesis header serializes to the following 180 bytes:

TEST VECTOR Main-network genesis block header

```
01000000
0000000000000000000000000000000000000000000000000000000000000000
0dc101dfc3c6a2eb10ca0c5374e10d28feb53f7eabcc850511ceadb99174aa66
0000000000000000000000000000000000000000000000000000000000000000
0000 0000000000000 0000 00 00 00000000 ffff011b 00c2eb0b00000000
00000000 00000000 a0d7b856 00000000
0000000000000000000000000000000000000000000000000000000000000000
00000000

block hash (internal):
  80d9212bf4ceb066ded2866b39d4ed89e0ab60f335c11df8e7bf85d9c35c8e29
block hash (display):
  298e5cc3d985bfe7f81dc135f360abe089edd4396b86d2de66b0cef42b21d980
BLAKE3 of the header (internal):
  1baa4b79809fc85fac9333371d37e203f34c345595f5ed5ed435fbc10b09d3d7
```

The lines hold, in order: `Version`; `PrevBlock`; `MerkleRoot`; `StakeRoot`; `VoteBits` to `SBits`; `Height`, `Size`, `Timestamp` and `Nonce`; `ExtraData`; `StakeVersion` (see Table 4.22). The BLAKE3 value only illustrates Section 4.4.2; the genesis block is not subject to a proof-of-work check.

Table 4.22 Main-network genesis header, field by field

Offset	Bytes	Field	Serialized bytes	Value
0	4	Version	`01000000`	1
4	32	PrevBlock	`0000000000000000000000000000000000000000000000000000000000000000`	none
36	32	MerkleRoot	`0dc101dfc3c6a2eb10ca0c5374e10d28feb53f7eabcc850511ceadb99174aa66`	Full hash of the coinbase
68	32	StakeRoot	`0000000000000000000000000000000000000000000000000000000000000000`	Empty stake tree

Offset	Bytes	Field	Serialized bytes	Value
100	2	VoteBits	`0000`	0
102	6	FinalState	`000000000000`	zero
108	2	Voters	`0000`	0
110	1	FreshStake	`00`	0
111	1	Revocations	`00`	0
112	4	PoolSize	`00000000`	0
116	4	Bits	`ffff011b`	`0x1b01ffff`
120	8	SBits	`00c2eb0b00000000`	200000000 (2 DCR)
128	4	Height	`00000000`	0
132	4	Size	`00000000`	0 (see Section 4.6.2)
136	4	Timestamp	`a0d7b856`	1454954400 (2016-02-08 18:00:00 UTC)
140	4	Nonce	`00000000`	0
144	32	ExtraData	`0000000000000000000000000000000000000000 0000000000000000000000000000`	zero
176	4	StakeVersion	`00000000`	0

The genesis block has one regular transaction, the coinbase, and an empty stake tree. Its `MerkleRoot` is `tree_root` of a one-element list, which is the coinbase's full hash, and its `StakeRoot` is `tree_root` of the empty list, which is zero: the genesis header conforms to Rule 4.8.

TEST VECTOR Main-network genesis coinbase in all three serialization types

```
full serialization (118 bytes):
  01000000                         version 1, type 0
  01                               1 input
  0000000000000000000000000000000000000000000000000000000000000000
  ffffffff 00                      index 0xffffffff, tree 0
  ffffffff                         sequence
  01                               1 output
  0000000000000000 0000            value 0, script version 0
  20                               script length 32
  801679e98561ada96caec2949a5d41c4cab3851eb740d951c10ecbcf265c1fd9
  00000000 00000000                lock time, expiry
  01                               1 witness
  ffffffffffffffff                 ValueIn = -1 (null)
  00000000 ffffffff                BlockHeight, BlockIndex (null)
  02 0000                          signature script 0000

prefix serialization (98 bytes): the first 98 bytes above,
  with version word 01000100 in place of 01000000
witness serialization (24 bytes):
  01000200 01 ffffffffffffffff 00000000 ffffffff 02 0000

transaction hash, internal and display:
  2c0ad603d44a16698ac951fa22aab5e7b30293fa1d0ac72560cdfcc9eabcdfe7
  e7dfbceac9fccd6025c70a1dfa9302b3e7b5aa22fa51c98a69164ad403d60a2c
witness hash, internal and display:
  6c6a06b87b148ada9f3950d0cc275a4bad21622116bd1de16909f5eff1874eeb
  eb4e87f1eff50969e11dbd16216221ad4b5a27ccd050399fda8a147bb8066a6c
full hash, internal and display:
  0dc101dfc3c6a2eb10ca0c5374e10d28feb53f7eabcc850511ceadb99174aa66
  66aa7491b9adce110585ccab7e3fb5fe280de174530cca10eba2c6c3df01c10d
BLAKE-256 of the full serialization (not an identifier), internal:
  33c951d47eef1f372f3ac45f8da9eb3f7982121b8db4a30ab12c28ccb3a51c00
```

TEST VECTOR Main-network genesis block (300 bytes)
The 180-byte header above, then `01`, the 118-byte full coinbase, and `00` (no stake transactions). Its size is $180 + 1 + 118 + 1 = 300$ bytes; the header's `Size` field nevertheless holds 0.

4.9.3 A Regular Transaction

The following one-input, two-output pay-to-public-key-hash transaction is taken from dcrd's test data (`dcrutil/txsort/testdata/tx150043-14.hex`).

TEST VECTOR Transaction `aa4c208c…ff6684`, full serialization (252 bytes)

```
01000000                                                 version 1, type 0
01                                                       1 input; outpoint:
64084d5f9b7fcedd3d816b7e1ba154123310dd892c09b53c8bf811804efab903
00000000 00                                              index 0, tree 0
ffffffff                                                 sequence
02                                                       2 outputs
52deda0a00000000 0000                                    182115922, ver. 0
19 76a9146ab21ce59fedbfe724cd979d1409397c68105c0d88ac    25-byte script
52deda0a00000000 0000                                    182115922, ver. 0
19 76a9145529ab44931e1ad083df53fa97f4922a80f234b688ac    25-byte script
00000000                                                 lock time 0
00000000                                                 expiry 0 (none)
01                                                       1 witness
781fb61500000000                                         ValueIn 364257144
0d4a0200                                                 BlockHeight 150029
04000000                                                 BlockIndex 4
6b                                                       107-byte script:
483045022100c6ae9a0587e64b6761a55e81321eee6dd3550e9c9a69c058d773
4f0ade1ce00702200650ecc1eacc938bedae55a7f2d0e73bee548dcce77352ef
f77b090b8087b6de012102d30f38730d4d4feb7f7fec0d2c81b550ff702d03c8
191cd6b98e28f9da9e2c9c
```

The fee, input amount minus output amounts, is $364257144 - 2 \cdot 182115922 = 25300$ atoms.

TEST VECTOR The same transaction, prefix and witness serializations

```
prefix serialization (127 bytes):
  01000100 01
  64084d5f9b7fcedd3d816b7e1ba154123310dd892c09b53c8bf811804efab903
  00000000 00 ffffffff
  02
  52deda0a00000000 0000
  19 76a9146ab21ce59fedbfe724cd979d1409397c68105c0d88ac
  52deda0a00000000 0000
  19 76a9145529ab44931e1ad083df53fa97f4922a80f234b688ac
  00000000 00000000

witness serialization (129 bytes):
  01000200 01 781fb61500000000 0d4a0200 04000000 6b
  483045022100c6ae9a0587e64b6761a55e81321eee6dd3550e9c9a69c058d773
  4f0ade1ce00702200650ecc1eacc938bedae55a7f2d0e73bee548dcce77352ef
  f77b090b8087b6de012102d30f38730d4d4feb7f7fec0d2c81b550ff702d03c8
  191cd6b98e28f9da9e2c9c

transaction hash, internal and display:
  8466ff8d3e1ca16c4f3bd6dc651c8bd9da9c0066d51dd1546957a9858c204caa
  aa4c208c85a9576954d11dd566009cdad98b1c65dcd63b4f6ca11c3e8dff6684
witness hash, internal and display:
  3f2d5c43f31cb6d0379c5522bcedb8552d4da1bd1b1b9f4efa7288dc9789f7bf
  bff78997dc8872fa4e9f1b1bbda14d2d55b8edbc22559c37d0b61cf3435c2d3f
```

```
full hash, internal and display:
  9a40e02e84c432d82dbdce8f62ab6bfce9ba3b28cbd4554529a29efc946736d2
  d2366794fc9ea2294555d4cb283bbae9fc6bab628fcebd2dd832c4842ee0409a
```

Decoding the 127-byte prefix serialization yields a transaction with serialization type 1, one input and two outputs, whose transaction hash is the same as above. Its full hash, computed from the zeroed witness fields, is (display) `d2de7ea99bdc25cb96c3b0bcbc1ea8cc267d8b24e8f1ee673ca637077e890f28`, which differs from the full hash of the complete transaction.

4.9.4 Merkle Roots of a Block After DCP-0005

Main-network block 432100 (from dcrd's test data, `internal/rpcserver/testdata/block432100.bz2`) is later than the activation of `headercommitments` at block 431488. It has 2 regular and 5 stake transactions.

TEST VECTOR Block 432100: roots and size

```
block hash (display):
  000000000000000023455b4328635d8e014dbeea99c6140aa715836cc7e55981
regular tree root (display):
  703983b9e834362a3cd3b4e397b76465b85fde1a87325415ec7527d471f24ed8
stake tree root (display):
  8ef384279548ac3965c0f972dcbc39992ffbc8563b4adba80e932887abe38102
combined root = BLAKE-256(regular root || stake root), internal:
  0bc8a255edde9901ecc4cdb93e4e573cb38ae91e84495ecddc0c93c019351d5d
header bytes 36..67 (MerkleRoot), equal to the combined root:
  0bc8a255edde9901ecc4cdb93e4e573cb38ae91e84495ecddc0c93c019351d5d
header bytes 68..99 (StakeRoot), the commitment root:
  7731998be0a78e955f6fb98d2f35479905c3279f6257beab42a51d556cef55b9
regular root computed with transaction-hash leaves (wrong), display:
  cccb5c46855861a22b010df67933286e9225f5cb55a5a8833d9a80fae27c1e1d
header bytes 132..135 (Size):
  de0a0000 = 2782 = length of the serialized block
```

The combined root is the BLAKE-256 hash of the 64 bytes formed by the regular and stake tree roots in internal byte order (the reverse of the display forms shown).

4.9.5 Decoding Failures

Each of the following modifications of the 118-byte genesis coinbase was passed to dcrd's transaction decoder.

TEST VECTOR Transaction decoding failures

modification	result
byte 2 (low byte of type) := 03	reject: unsupported serialization type
byte 41 (outpoint tree) := 80	reject: negative transaction tree (-128)
byte 41 (outpoint tree) := 7f	decodes (tree 127)
byte 98 (witness count) := 02	reject: witness count 2, prefix count 1
byte 98 (witness count) := 00	reject: witness count 0, prefix count 1
last byte removed	reject: truncated
byte 2 := 01	decodes as a 98-byte prefix-only transaction; the remaining 20 bytes are not part of it

The last case shows why a decoder must stop at the end of the structure: in a block, those 20 bytes would be read as the start of the next transaction or count.

CHAPTER 5

Keys and Addresses

An **address** is a short, checksummed text string that tells a payer how to construct the output script that pays a particular recipient. Decred addresses, private-key strings and hierarchical deterministic (HD) extended keys all use the same building blocks: a 2- or 4-byte prefix that identifies the network and the kind of object, a payload, a checksum built from BLAKE-256, and the Base58 text encoding.

> **NOTE**
> **Nothing in this chapter is a consensus rule.** Blocks and transactions never contain addresses; consensus code sees only output scripts (Chapter 6). An address is a human-facing encoding of a script, and two nodes that disagree about addresses still agree about every block. The chapter must nevertheless be followed exactly by any software that exchanges addresses, keys or signed messages with other Decred software: a wrong prefix or template sends funds to a script nobody can spend. Where the keywords **MUST** and **SHOULD** appear below they express interoperability requirements, not validity of blocks. There are therefore no rule boxes in this chapter.

The reference behaviour is that of dcrd's packages `txscript/stdaddr` (addresses and the scripts they produce), `dcrutil` (private keys, message verification, amounts), `hdkeychain` (HD keys), `chaincfg` (per-network prefixes) and the external module `github.com/decred/base58` v1.0.6, which dcrd pins in its `go.mod`. Where wallet behaviour is described, it is that of dcrwallet [22] and is labelled as a convention.

Throughout, BLAKE256 is the hash function of Chapter 3 and

$$\mathrm{HASH160}(x) = \mathrm{RIPEMD160}(\mathrm{BLAKE256}(x)). \tag{5.1}$$

This differs from Bitcoin's HASH160, which uses SHA-256 in place of BLAKE-256, so Decred's key hashes, addresses and HD fingerprints differ from their Bitcoin counterparts for the same key.

5.1 Overview

Table 5.1 lists the encoded objects defined in this chapter. All are Base58 strings over the same alphabet; they differ in prefix width, payload and checksum.

Table 5.1 Text-encoded objects

Object	Prefix	Payload	Checksum (4 bytes)	Chars
P2PK address (Section 5.4.1)	2	33	$\mathrm{BLAKE256}^2$	53
P2PKH or P2SH address	2	20	$\mathrm{BLAKE256}^2$	35
WIF private key (Section 5.6)	2	33	BLAKE256 (single)	53
Extended key (Section 5.8)	4	74	$\mathrm{BLAKE256}^2$	111 [a]

[a] 112 on regnet, whose extended-key version bytes begin with `0xea`. All lengths are exact for every payload under the prefixes of Table 5.2 (verified by encoding all-zero and all-`0xff` payloads).

5.2 Base58 and Base58Check

5.2.1 The alphabet

Base58 represents a byte string as a number in base 58, written with the 58-character alphabet

```
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
```

in which the character at (zero-based) position i has value i. The digits `0`, the capitals `I` and `O`, and the lowercase `l` are absent to avoid visual confusion. This is the same alphabet Bitcoin uses [23].

5.2.2 Encoding and decoding

The encoding treats the input as a big-endian unsigned integer and writes it in base 58, most significant digit first. Because leading zero bytes do not change the integer, each leading `0x00` byte is written as one leading `1` (the digit for zero) so that the encoding is reversible.

ALGORITHM 5.1 **Base58 encoding and decoding**

```
ALPHABET = "123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz"

def base58_encode(b: bytes) -> str:
    n = int_from_be(b)                  # 0 for the empty string
    digits = ""
    while n > 0:
        digits = ALPHABET[n % 58] + digits
        n = n // 58
    zeros = count_leading(b, 0x00)      # number of leading zero bytes
    return "1" * zeros + digits

def base58_decode(s: str) -> bytes:
    if s == "":
        return b""
    zeros = count_leading(s, "1")       # number of leading '1' characters
    n = 0
    for c in s[zeros:]:
        if c not in ALPHABET:
            return b""                  # invalid character: empty result, no error
        n = n * 58 + ALPHABET.index(c)
    body = be_minimal(n)                # big-endian, no leading zeros; b"" if n == 0
    return b"\x00" * zeros + body
```

dcrd: github.com/decred/base58 · Encode, Decode

IMPLEMENTATION PITFALL
The decoder reports an invalid character by returning an empty byte string, not an error. Every caller in dcrd then rejects the result because its length is wrong, so the observable effect is a “malformed” or “invalid format” error rather than an “invalid character” error.

TEST VECTOR Base58

Bytes (hex)	Base58	Bytes (hex)	Base58
(empty)	(empty)	`61`	`2g`
`00`	`1`	`626262`	`a3gV`
`0000`	`11`	`00010203`	`1Ldp`
`000001`	`112`	`ffffffff`	`7YXq9G`
`0000ffff`	`11LUv`	`3132333435393837363 0`	`3mJr7AoUXx2Wqd`

Decoding `z` yields `39`; decoding `21` yields `3a`; decoding `0OI1` yields the empty string (all four characters are outside the alphabet).

5.2.3 Base58Check

Base58Check adds a type prefix and a checksum. Decred's variant differs from Bitcoin's in two ways that make the encodings incompatible: the prefix (called the **version** or **network identifier**) is **two** bytes, and the checksum is the first four bytes of a **double BLAKE-256**, not of a double SHA-256.

ALGORITHM 5.2 **Base58Check**

```
def checksum(b: bytes) -> bytes:
    return BLAKE256(BLAKE256(b))[0:4]

def check_encode(payload: bytes, version: bytes) -> str:     # len(version) == 2
    b = version + payload
    return base58_encode(b + checksum(b))

def check_decode(s: str) -> (bytes, bytes):
    d = base58_decode(s)
    if len(d) < 6:
        error("invalid format")      # also the result of any invalid character
    if d[len(d)-4:] != checksum(d[0:len(d)-4]):
        error("checksum")
    return d[2:len(d)-4], d[0:2]     # (payload, version)
```

dcrd: `github.com/decred/base58` · `CheckEncode`, `CheckDecode`

TEST VECTOR Base58Check

Item	Value
Version	`073f` (main-network P2PKH)
Payload	`0102030405`
BLAKE256(version ‖ payload)	`1d6b06142d26b2ad114a8ddafc2a140a8156dbe167d74aab8ba677273b5fe666`
BLAKE256 of that	`a5867abf3983ffa4aded66332f4d689decd0ab28a8b1076f47c10b2872984ff7`
Checksum	`a5867abf`
Encoded	`2oDcdouVBocLC4W`
Last character altered	`2oDcdouVBocLC4a` ⟶ checksum error
`1111`	decodes to 4 zero bytes ⟶ invalid format
`Ds0abc`	`0` is not in the alphabet ⟶ invalid format

5.3 Network Prefixes

Each network has its own set of prefixes, so a string encoded for one network fails to decode on another. The prefixes are chosen so that the first one or two characters of the text form are constant and identify the network and type: on the main network, addresses begin with `D`, P2PKH addresses with `Ds`, and so on. The first character is recorded in the network parameters as `NetworkAddressPrefix` but is informative only; decoding relies solely on the decoded prefix bytes.

Table 5.2 Prefixes per network. Each cell gives the prefix bytes in hexadecimal (in serialization order) and the fixed leading characters of the encoded string.

Object	Parameter	mainnet	testnet3	simnet	regnet
First letter	`NetworkAddressPrefix`	`D`	`T`	`S`	`R`

Object	Parameter	mainnet	testnet3	simnet	regnet
P2PK (all suites)	PubKeyAddrID	1386 Dk	28f7 Tk	276f Sk	25e5 Rk
P2PKH ECDSA	PubKeyHashAddrID	073f Ds	0f21 Ts	0e91 Ss	0e00 Rs
P2PKH Ed25519	PKHEdwardsAddrID	071f De	0f01 Te	0e71 Se	0de0 Re
P2PKH Schnorr	PKHSchnorrAddrID	0701 DS	0ee3 TS	0e53 SS	0dc2 RS
P2SH	ScriptHashAddrID	071a Dc	0efc Tc	0e6c Sc	0ddb Rc
WIF private key	PrivateKeyID	22de Pm	230e Pt	2307 Ps	22fe Pr
Extended private key	HDPrivateKeyID	02fda4e8 dprv	04358397 tprv	0420b903 sprv	eab40448 rprv
Extended public key	HDPublicKeyID	02fda926 dpub	043587d1 tpub	0420bd3d spub	eab4f987 rpub
BIP44 coin type	SLIP0044CoinType	42	1	1	1
Legacy coin type	LegacyCoinType	20	11	115	1

All five address prefixes of a network are distinct, so the decoded prefix determines the address type unambiguously. The coin types are explained in Section 5.8.4.

5.4 Address Types

An address is meaningful only together with a **script version**, the 16-bit version number carried with every transaction output (Chapter 4). Every address type defined at present is for script version 0, and every script an address produces is a version 0 script. A request for an address of any other script version fails with `ErrUnsupportedScriptVersion`.

Table 5.3 lists the seven address types. The names follow the form *pay-to-X-signature-suite*: what the output script commits to (a public key, the hash of a public key, or the hash of a script) and, for the first two, which of the three signature suites of Chapter 3 must be used to spend it. In the script templates, `H` is a 20-byte HASH160, `K33` a 33-byte compressed secp256k1 public key and `K32` a 32-byte Ed25519 public key.

Table 5.3 Version 0 address types and their payment scripts

Address type	Prefix	Payload	Payment script (hex template)
P2PK, ECDSA secp256k1	PubKeyAddrID	33	`21` `K33` `ac`
P2PK, Ed25519	PubKeyAddrID	33	`20` `K32` `51` `be`
P2PK, Schnorr secp256k1	PubKeyAddrID	33	`21` `K33` `52` `be`
P2PKH, ECDSA secp256k1	PubKeyHashAddrID	20	`76` `a9` `14` `H` `88` `ac`
P2PKH, Ed25519	PKHEdwardsAddrID	20	`76` `a9` `14` `H` `88` `51` `be`
P2PKH, Schnorr secp256k1	PKHSchnorrAddrID	20	`76` `a9` `14` `H` `88` `52` `be`
P2SH	ScriptHashAddrID	20	`a9` `14` `H` `87`

In opcode form (Chapter 6 defines the opcodes): `21` is `OP_DATA_33`, `20` is `OP_DATA_32`, `14` is `OP_DATA_20`, `51` and `52` are `OP_1` and `OP_2` (which push the signature-type numbers 1 for Ed25519 and 2 for Schnorr), `76` is `OP_DUP`, `a9` is `OP_HASH160`, `88` is `OP_EQUALVERIFY`, `87` is `OP_EQUAL`, `ac` is `OP_CHECKSIG` and `be` is `OP_CHECKSIGALT`. The scripts are therefore:

```
P2PK-ECDSA      <K33> OP_CHECKSIG
P2PK-Ed25519    <K32> OP_1 OP_CHECKSIGALT
P2PK-Schnorr    <K33> OP_2 OP_CHECKSIGALT
P2PKH-ECDSA     OP_DUP OP_HASH160 <H> OP_EQUALVERIFY OP_CHECKSIG
P2PKH-Ed25519   OP_DUP OP_HASH160 <H> OP_EQUALVERIFY OP_1 OP_CHECKSIGALT
P2PKH-Schnorr   OP_DUP OP_HASH160 <H> OP_EQUALVERIFY OP_2 OP_CHECKSIGALT
P2SH            OP_HASH160 <H> OP_EQUAL
```

Script lengths are 35, 35, 36, 25, 26, 26 and 23 bytes respectively.

5.4.1 Pay-to-pubkey addresses

All three P2PK types share the prefix `PubKeyAddrID` and a 33-byte payload made of a **type byte** followed by 32 bytes of key material. The low seven bits of the type byte hold the signature type of Chapter 3 (0 ECDSA secp256k1, 1 Ed25519, 2 Schnorr secp256k1). For the two secp256k1 types the key material is the 32-byte big-endian X coordinate, and the high bit (`0x80`) of the type byte is set exactly when the Y coordinate is odd, that is, when the compressed serialization starts with `0x03`. For Ed25519 the key material is the 32-byte encoded public key unchanged and the high bit is zero.

Table 5.4 P2PK payload type byte

Type byte	Suite	Reconstructed public key
`0x00`	ECDSA secp256k1	`02` ‖ X
`0x80`	ECDSA secp256k1	`03` ‖ X
`0x01`	Ed25519	the 32 bytes as given
`0x02`	Schnorr secp256k1	`02` ‖ X
`0x82`	Schnorr secp256k1	`03` ‖ X

A P2PK address can only be created for a valid public key. For the secp256k1 types the key **MUST** be a point on the curve and is always used in its 33-byte **compressed** serialization: an uncompressed (`0x04`) or hybrid (`0x06`, `0x07`) key is refused with `ErrInvalidPubKeyFormat`, even though the script language would accept a P2PK-ECDSA script containing a 65-byte uncompressed key. Ed25519 keys **MUST** parse as valid Ed25519 points. dcrd's test is decompression plus a range check on the recovered x: it accepts every 32-byte string whose low 255 bits, read little-endian as y, and whose top bit, as the sign of x, decode to a point on the curve. It does not require $y < p$ (so non-canonical encodings such as `ee ff … ff 7f`, which is $y = p + 1$, are accepted), it accepts points of small order, including the identity `01 00 … 00`, and it rejects an encoding with $x = 0$ and the sign bit set. The address and its payment script keep the 32 bytes exactly as given.

> **IMPLEMENTATION PITFALL**
> When an output script of the form `<65-byte uncompressed key> OP_CHECKSIG` is converted to an address (Section 5.4.5), the result is the P2PK-ECDSA address of the **compressed** form of that key, and the payment script of that address is the 35-byte compressed script, not the original. Address-to-script and script-to-address are not inverses for such outputs.

> **IMPLEMENTATION PITFALL**
> The decoder masks the high bit off the type byte before testing for Ed25519, so a payload beginning `0x81` decodes as the same Ed25519 address as `0x01`, and re-encodes with `0x01`. Implementations that compare address strings for equality should compare re-encoded strings. Any type value other than 0, 1 or 2 after masking is unsupported.

5.4.2 Pay-to-pubkey-hash addresses

A P2PKH address carries the 20-byte HASH160 of a public key, and the payment script requires the spender to reveal a public key that hashes to it and a valid signature by that key. The signature suite is fixed by the address prefix, not by the payload, and selects the script template of Table 5.3.

- **ECDSA secp256k1**: the hash is normally of the 33-byte compressed key. For historical reasons the script also accepts a key in 65-byte uncompressed form if the hash was computed over that form; wallets **SHOULD** always hash the compressed form.
- **Schnorr secp256k1**: the hash **MUST** be of the 33-byte compressed key. The consensus rules for `OP_CHECKSIGALT` with signature type 2 reject uncompressed keys, so an output whose hash commits to an uncompressed key is unspendable.
- **Ed25519**: the hash is of the 32-byte encoded Ed25519 public key.

Every P2PK address has a corresponding P2PKH address of the same suite, obtained by hashing the (compressed) public key and using the P2PKH prefix of that suite.

5.4.3 Pay-to-script-hash addresses

A P2SH address carries the 20-byte HASH160 of a **redeem script**; the payment script requires the spender to reveal a script with that hash and to satisfy it. The execution rules for P2SH, including the restrictions on the redeem script, are in Chapter 6.

TEST VECTOR Addresses for a fixed key

The secp256k1 private key and the Ed25519 private scalar are both the 32 bytes `01 02 03 … 1f 20`, read big-endian. The P2SH redeem script is the P2PK-ECDSA payment script of the secp256k1 key.

Item	Value
Private key	`0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20`
secp256k1 public key (compressed)	`0284bf7562262bbd6940085748f3be6afa52ae317155181ece31b66351ccffa4b0`
HASH160 of it	`b47b61b2aeec0f3900616e6b032246e1a11e09b9`
Ed25519 public key	`80334024b705b5fd76b1bce1b26d96234ab7b2d989987895fa72c43b3e5c5f85`
HASH160 of it	`6ede19217e38d198dcab3778cb5b8cd451478242`
Redeem script	`210284bf7562262bbd6940085748f3be6afa52ae317155181ece31b66351cc ffa4b0ac`
HASH160 of it	`d7b0b267d1462b015611b5f78912214883384571`

Type	Network	Address
P2PK ECDSA	mainnet	`DkM3V4UE8ofRtdYsHfRUKn4we8zQtZoy4nLoRpKvE2bhm4jSWNCHZ`
	testnet3	`TkKmZPK7sjUFaKwZhFL2CbQE4BxAVnnLGLF6rDhdtB9VZbgL6sQyQ`
P2PK Ed25519	mainnet	`DkM5PoUuyB2y79g7DjNFoZ9dTQEgYsBP5atQ5USjPetm8vSomnEgj`
	testnet3	`TkKoU8Koi6qnnr4odKGogNUusTCSA69kH8nhVspT3oSYwTPa2Qx9E`
P2PK Schnorr	mainnet	`DkM7NYph2sNHF5pWT5EdZmyaqgpe817NkXv5a2pDbpb1mqii1LupZ`
	testnet3	`TkKqSsfamoB6vnDCrf9BSbJsFjnPjE5jx5pNzSBwFy8oaNfZn7kpv`
P2PKH ECDSA	mainnet	`DshRCvfb21VQvaXZYhBivFS7VAurvcKi32t`
	testnet3	`TshURuo6QoYX2wCvN5ot4pTP5GsnVK4bmxd`
P2PKH Ed25519	mainnet	`DeiFpNi2n7zvqzsxPP2G6QjWS89h8A4vTwZ`
	testnet3	`TeiK3MqYAv42xMZKCmeREykn2E7cgjKnRy2`
P2PKH Schnorr	mainnet	`DSkTosw79zJgbsevBBB5ABfEi9dtSZ9MLCN`
	testnet3	`TSkX2s4cYnMniELGzZoEJkgWJFbp1E7F8u4`
P2SH	mainnet	`Dcs841mixKtD7siWRMBz824w7wj7M9kgcqg`
	testnet3	`TcsBGzuEM7wKEEPsEjp9Gb6Ci3h2uriPgBA`

The decoded payloads are, in order: `0084bf7562262bbd6940085748f3be6afa52ae317155181ece31b66351ccffa4b0`, `0180334024b705b5fd76b1bce1b26d96234ab7b2d989987895fa72c43b3e5c5f85`, `0284bf7562262bbd6940085748f3be6afa52ae317155181ece31b66351ccffa4b0`, and the three HASH160 values above. The payment scripts are:

Item	Value
P2PK ECDSA	`210284bf7562262bbd6940085748f3be6afa52ae317155181ece31b66351ccffa4b0ac`
P2PK Ed25519	`2080334024b705b5fd76b1bce1b26d96234ab7b2d989987895fa72c43b3e5c5f8551be`
P2PK Schnorr	`210284bf7562262bbd6940085748f3be6afa52ae317155181ece31b66351ccffa4b052be`
P2PKH ECDSA	`76a914b47b61b2aeec0f3900616e6b032246e1a11e09b988ac`
P2PKH Ed25519	`76a9146ede19217e38d198dcab3778cb5b8cd4514782428851be`
P2PKH Schnorr	`76a914b47b61b2aeec0f3900616e6b032246e1a11e09b98852be`

Item	Value
P2SH	a914d7b0b267d1462b015611b5f7891221488338457187

TEST VECTOR The same key on the private networks

Type	Network	Address
P2PK ECDSA	simnet	SkLUPG3sF7D7gM6NaTdJ8XB98ddQnYk3JX4uksdpmha4ZDvN1V5Bw
	regnet	Rk41wzTtp8e9WRD9vZmnGrbsEaSFmdsCtP1UQQfKUF388HW4Hn1BP
P2PKH ECDSA	simnet	Sskoa8AAvZZnsAnqtZPvAzQBneEgez7tfS8
	regnet	RsQo7MQx9cQbpawdL1ZcxgDiiP6Ksoe5TNC
P2PKH Ed25519	simnet	SemeBaCcgg5Jnb9EjFETM9hajbUWrRVxTNf
	regnet	ReRdioTPuiv7k1J2AhQA8qX7fLLA5H3J7Sq
P2PKH Schnorr	simnet	SSorB5Rh4YP4YTvCX3PGQvdK1cxiApYcq4v
	regnet	RSTqiJgUHbDsVt4yxVYyCcSqwMpMPnBJMUm
P2SH	simnet	ScvWRDGJrsxb4TynmDQBNm31RR3w5TFJrGT
	regnet	RcaVxSX65voQ1t8aCfZtASrYM9uaJQ9DsGk

5.4.4 Stake and treasury scripts

The stake system (Chapter 8) and the treasury (Chapter 12) use output scripts that consist of a one-byte **stake opcode tag** followed by an ordinary P2PKH or P2SH script, plus one special `OP_RETURN` commitment format. Only two address types can appear in them: **P2PKH ECDSA secp256k1** and **P2SH**. Ed25519 and Schnorr P2PKH addresses and all P2PK addresses cannot be used for staking or for treasury payouts. Table 5.5 gives the scripts derived from an address of either kind; `S` denotes the address's ordinary payment script from Table 5.3 (25 bytes for P2PKH, 23 for P2SH).

Table 5.5 Stake-tagged scripts derived from an address

Purpose	Script	Where it is valid
Voting rights	`ba` ‖ `S` (`OP_SSTX`)	Output 0 of a ticket purchase
Reward commitment	`6a 1e` ‖ commitment (`OP_RETURN`)	Odd outputs of a ticket purchase (Table 5.6)
Stake change	`bd` ‖ `S` (`OP_SSTXCHANGE`)	Change outputs of a ticket purchase or a treasury add
Vote payment	`bb` ‖ `S` (`OP_SSGEN`)	Payment outputs of a vote, to a committed address
Revocation payment	`bc` ‖ `S` (`OP_SSRTX`)	Payment outputs of a revocation, to a committed address
Treasury payment	`c3` ‖ `S` (`OP_TGEN`)	Payment outputs of a treasury spend

The **reward commitment** is a 32-byte null-data script: `OP_RETURN`, `OP_DATA_30`, then the 30 bytes of Table 5.6. It names the address that a later vote or revocation of the ticket must pay, the amount that address contributed, and optional fee limits. Which scripts are required where, and how the amounts and limits are enforced, are consensus matters specified in Chapter 8; this section gives only the encoding.

Table 5.6 Reward commitment payload (30 bytes, after `6a 1e`)

Field	Type	Bytes	Description
`hash`	`[20]byte`	20	HASH160 from a P2PKH-ECDSA or P2SH address.
`amount`	`uint64 LE`	8	Contributed amount in atoms in the low 63 bits. The most significant bit (`0x80` in the last byte) is 1 if `hash` is a script hash (P2SH) and 0 if it is a public-key hash (P2PKH-ECDSA).

Field	Type	Bytes	Description
`limits`	`uint16 LE`	2	Fee limits. Low byte: vote fee limit; high byte: revocation fee limit. Each byte is 0 for "no fee allowed" (the payment must be exact, Chapter 8) or `0x40 | e` for a fee limit of 2^e atoms. This is what the encoder produces; how consensus reads every bit of the field is in Chapter 8.

When a wallet asks for a fee limit of $L > 0$ atoms, the exponent e is the smallest integer with $2^e \geq L$, so limits that are not powers of two are rounded up. A requested limit of 0 produces a zero byte, which permits no fee at all. dcrd computes e as $\lceil \log_2 L \rceil$ in double-precision floating point; this agrees with the integer definition for every $L \leq 2^{49}$ (about 5.6 million DCR). The smallest disagreement is $L = 2^{49} + 1$, whose floating-point logarithm is exactly 49, so dcrd writes `0x71` where the integer definition gives `0x72`; larger limits just above a power of two can disagree in the same way. dcrd does not reject a negative limit, and the byte it writes for one is not meaningful.

Reading a commitment back into an address takes bytes 2 to 21 of the script as the hash and the most significant bit of byte 29 as the P2SH flag, and builds a P2SH or P2PKH-ECDSA address on the network in use.

TEST VECTOR Stake scripts for the P2PKH-ECDSA and P2SH addresses above

Commitment (a) is for 1 DCR (10^8 atoms) with no vote fee limit and a revocation fee limit of 2^{24}; commitment (b) is for 1 DCR with limits 1 and $2^{24} + 1$.

Item	Value
Voting rights, P2PKH	`ba76a914b47b61b2aeec0f3900616e6b032246e1a11e09b988ac`
Voting rights, P2SH	`baa914d7b0b267d1462b015611b5f7891221488338457187`
Commitment (a), P2PKH	`6a1eb47b61b2aeec0f3900616e6b032246e1a11e09b900e1f505000000000058`
Commitment (a), P2SH	`6a1ed7b0b267d1462b015611b5f7891221488338457100e1f505000000800058`
Commitment (b), P2PKH	`6a1eb47b61b2aeec0f3900616e6b032246e1a11e09b900e1f505000000004059`
Stake change, P2PKH	`bd76a914b47b61b2aeec0f3900616e6b032246e1a11e09b988ac`
Vote payment, P2PKH	`bb76a914b47b61b2aeec0f3900616e6b032246e1a11e09b988ac`
Revocation payment, P2PKH	`bc76a914b47b61b2aeec0f3900616e6b032246e1a11e09b988ac`
Treasury payment, P2PKH	`c376a914b47b61b2aeec0f3900616e6b032246e1a11e09b988ac`
Treasury payment, P2SH	`c3a914d7b0b267d1462b015611b5f7891221488338457187`

The limit bytes for requested limits 1, 2, 3, 4, 2^{24} and $2^{24} + 1$ are `0x40`, `0x41`, `0x42`, `0x42`, `0x58` and `0x59`.

5.4.5 From scripts back to addresses

Software that displays transactions maps each version 0 output script back to zero or more addresses. The script classes are those of Chapter 6; Table 5.7 gives the address each class yields. A script whose version is not 0 yields no address.

Table 5.7 Addresses extracted from version 0 script classes

Script class	Address(es)
P2PKH ECDSA, Ed25519, Schnorr	The P2PKH address of that suite
P2SH	The P2SH address
P2PK ECDSA (compressed or uncompressed key)	The P2PK-ECDSA address of the compressed key, if the key parses
P2PK Ed25519, P2PK Schnorr	The P2PK address of that suite, if the key parses
Multisignature (`OP_CHECKMULTISIG`)	One P2PK-ECDSA address per public key that parses, in script order
Stake-tagged (`OP_SSTX`, `OP_SSGEN`, `OP_SSRTX`, `OP_SSTXCHANGE`, `OP_TGEN`) over P2PKH or P2SH	The inner P2PKH-ECDSA or P2SH address

Script class	Address(es)
Null data (including reward commitments), treasury add, non-standard	None

5.5 Decoding Addresses

Decoding first applies a shape test (length 35 or 53, Base58 alphabet only) that selects the version 0 decoder, then Base58Check, then dispatches on the two prefix bytes. An address for a different network fails with `ErrUnsupportedAddress`, because its prefix matches none of the expected values. dcrd also rejects strings longer than 54 characters with `ErrMalformedAddress` before Base58 decoding; the shape test makes that check unreachable through `DecodeAddress` (it applies only when the version 0 decoder `DecodeAddressV0` is called directly, which also skips the shape test).

ALGORITHM 5.3 **Decoding a version 0 address**

```
def decode_address(s: str, net) -> Address:
    # Shape test that selects the version 0 decoder.
    if len(s) not in (35, 53) or any(c not in ALPHABET for c in s):
        error(ErrUnsupportedAddress)
    payload, ver = check_decode(s)   # checksum failure -> ErrBadAddressChecksum,
                                     # any other failure -> ErrMalformedAddress
    if ver == net.ScriptHashAddrID:  return P2SH(hash20(payload))
    if ver == net.PubKeyHashAddrID:  return P2PKH_ECDSA(hash20(payload))
    if ver == net.PKHSchnorrAddrID:  return P2PKH_SCHNORR(hash20(payload))
    if ver == net.PKHEdwardsAddrID:  return P2PKH_ED25519(hash20(payload))
    if ver == net.PubKeyAddrID:
        if len(payload) < 1:
            error(ErrMalformedAddressData)
        sig_type = payload[0] & 0x7f
        if sig_type in (0, 2):                          # ECDSA or Schnorr secp256k1
            if len(payload) != 33:
                error(ErrMalformedAddressData)
            prefix = 0x03 if payload[0] & 0x80 else 0x02
            key = bytes([prefix]) + payload[1:33]
            require_valid_secp256k1_point(key)          # else ErrInvalidPubKey
            return P2PK_ECDSA(key) if sig_type == 0 else P2PK_SCHNORR(key)
        if sig_type == 1:                               # Ed25519
            if len(payload) - 1 != 32:
                error(ErrMalformedAddressData)
            require_valid_ed25519_point(payload[1:33])  # else ErrInvalidPubKey
            return P2PK_ED25519(payload[1:33])
    error(ErrUnsupportedAddress)     # includes every prefix of another network

def hash20(payload):
    if len(payload) != 20:
        error(ErrInvalidHashLen)
    return payload
```

dcrd: txscript/stdaddr/address.go · DecodeAddress
txscript/stdaddr/addressv0.go · DecodeAddressV0

TEST VECTOR Address decoding failures (main network)

Item	Value
`DshRCvfb21VQvaXZYhBivFS7VAurvcKi32a`	last character altered: `ErrBadAddressChecksum`
`DshRCvfb21VQvaXZYhBivFS7VAurvcKi32t` on testnet3	`ErrUnsupportedAddress`
`DsQxuVRvS4eaJ42dhQEsCXauMWjvopWgrV` (34 characters)	`ErrUnsupportedAddress`
`DkRS8aabvzJAVJDLR4BQdg1xBaxMZdyDTVRXhvHYLTpYdsNzbinHP` (type byte `0x83`)	`ErrUnsupportedAddress`

Item	Value
`DkRND64bRG7Rg8w7c1QTFTEgCXTHNnPWwfcD9FfuSQ8dzZcKTd9Hm` (type byte `0x81`)	accepted; re-encodes as `DkM5PoUu…omnEgj` (P2PK Ed25519 above)

5.6 Private Keys: Wallet Import Format

The **Wallet Import Format** (WIF) encodes a single private key together with its signature suite and network. It is used to move individual keys between wallets. Unlike Bitcoin's WIF it carries an explicit signature-type byte and no "compressed" flag: secp256k1 public keys derived from a WIF key are always compressed.

Table 5.8 WIF serialization (39 bytes, then Base58)

Field	Type	Bytes	Description
`net`	`[2]byte`	2	`PrivateKeyID` of the network.
`type`	`uint8`	1	Signature type: 0 ECDSA secp256k1, 1 Ed25519, 2 Schnorr secp256k1.
`key`	`[32]byte`	32	Private scalar, 32-byte big-endian.
`checksum`	`[4]byte`	4	First 4 bytes of BLAKE256 (a **single** hash) of the preceding 35 bytes.

IMPLEMENTATION PITFALL
The WIF checksum is a single BLAKE-256, unlike the double BLAKE-256 of addresses and extended keys, so WIF is **not** Base58Check as defined in Section 5.2.3. A WIF string with a double-hash checksum is rejected. (The dcrd source comment describing the checksum as "double SHA256" is stale; the code uses one BLAKE-256.)

For secp256k1 the 32 bytes are reduced modulo the group order n when the key is used; the public key is the compressed serialization of $d \cdot G$. For Ed25519 the 32 bytes are the private scalar d, read big-endian (not the 32-byte seed of RFC 8032 [20]); it **MUST** satisfy $0 < d \leq \ell$, where ℓ is the order of the Ed25519 base point, and the public key is the 32-byte encoding of $d \cdot B$.

ALGORITHM 5.4 Decoding a WIF string

```
def decode_wif(s: str, net) -> (int, bytes):
    if len(s) > 54:
        error("malformed private key")
    d = base58_decode(s)
    if len(d) != 39:
        error("malformed private key")
    if BLAKE256(d[0:35])[0:4] != d[35:39]:
        error("checksum mismatch")
    if d[0:2] != net.PrivateKeyID:
        error("wrong network")
    sig_type, key = d[2], d[3:35]
    if sig_type in (0, 2):
        return sig_type, key           # no range check: 0 or >= n is not rejected
    if sig_type == 1:
        if not (0 < int_from_be(key) <= ED25519_ORDER):
            error("invalid scalar")
        return 1, key
    return 0, b""                      # unknown type: no error, empty key (see below)
```

dcrd: dcrutil/wif.go · DecodeWIF

IMPLEMENTATION PITFALL
dcrd's decoder does not reject an unknown type byte: it returns an ECDSA-typed result with an empty key. It also accepts a secp256k1 key of zero (deriving a meaningless public key `02 00…00`) or one that is not below n.

Implementations **SHOULD** reject all three. Encoders **MUST** write the key as exactly 32 bytes, left-padded with zeros; the dcrd encoder copies whatever key bytes it is given, and an HD private key derived by the Decred method can be shorter than 32 bytes (Section 5.8.2).

TEST VECTOR WIF for the private key `01 02 … 1f 20`

Network	Type	WIF
mainnet	0 ECDSA	`PmQdGsGYheiVkbzcb54ZXJZW2FvaPdoqb4A3oGuzndZjLXfrCYEYL`
mainnet	1 Ed25519	`PmQfDcSn9gZvRq8SAmy99JWpd2qh1MTYRvwgsPA9UXZPLvANp1UPz`
mainnet	2 Schnorr	`PmQhAMd1biRM74GFkUsimJU9Dokod57FGojKwVQJARZ3MJf1hzZqG`
testnet3	0 ECDSA	`PtWTYK1qt46UMNqFTchWPWv3QCnDhhtrUsMx48idLCXuW3ZQuqcg9`
testnet3	1 Ed25519	`PtWVV4C5L5wu2by53Kc61WsMzyhLKRYZKk9b8Exn26XZWS42kL26f`
testnet3	2 Schnorr	`PtWXRoNJn7oKhq6td2WfdWpgbkcSw9CGAcwECMCvhzXDWpYakremR`

The decoded bytes of the first are `22de000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20d8225f95`; the checksum `d8225f95` is the first four bytes of the BLAKE-256 of the preceding 35 bytes. The public keys are those of the address vector above.

5.7 Signed Messages

A **signed message** proves control of a P2PKH-ECDSA address without making a transaction. dcrwallet produces them and dcrd verifies them through its `verifymessage` RPC. Only P2PKH-ECDSA secp256k1 addresses are supported; a verifier **MUST** reject any other address type.

The signed digest is

$$h = \mathrm{BLAKE256}(\mathrm{varstr}(\mathrm{MAGIC}) \parallel \mathrm{varstr}(m)), \tag{5.2}$$

where m is the message as bytes (UTF-8), MAGIC is the 23 ASCII bytes `Decred Signed Message:` followed by a line feed (`0x0a`), and $\mathrm{varstr}(x)$ is the length of x as a variable-length integer (Chapter 4) followed by x. The hash is a single BLAKE-256.

The signature is a 65-byte **compact recoverable** ECDSA signature: a header byte, then r and s as 32-byte big-endian integers. The header byte is $27 + \mathrm{rec} + 4c$, where c is 1 if the address hashes the compressed public key (always the case for dcrwallet) and $\mathrm{rec} \in \{0, 1, 2, 3\}$ is the recovery code: bit 0 is the parity of the Y coordinate of the point R that satisfies $s \cdot R = e \cdot G + r \cdot Q$ (with e the digest reduced modulo n and Q the public key), and bit 1 is set if the X coordinate of R is at least n, so that r is that coordinate minus n. dcrd's signer outputs only $s \le n/2$; when it negates s to achieve this it also flips bit 0, because R is then the negation of the nonce point. The 65 bytes are exchanged in standard Base64 with padding (RFC 4648). Signers use deterministic nonces (RFC 6979 [16], as in Chapter 3), so a given key and message always produce the same signature.

ALGORITHM 5.5 **Verifying a signed message**

```
MAGIC = b"Decred Signed Message:\n"

def message_hash(m: bytes) -> bytes:
    return BLAKE256(varint(len(MAGIC)) + MAGIC + varint(len(m)) + m)

def verify_message(address: str, sig_b64: str, m: bytes, net) -> bool:
    if not is_P2PKH_ECDSA(decode_address(address, net)):
        error("not a pay-to-pubkey-hash address")
    sig = base64_decode(sig_b64)                          # failure -> error
    Q, compressed = recover_compact(sig, message_hash(m))   # failure -> False (RPC)
```

```
    key = ser_compressed(Q) if compressed else ser_uncompressed(Q)
    return encode_P2PKH_ECDSA(HASH160(key), net) == address
```

dcrd: internal/rpcserver/rpcserver.go · handleVerifyMessage | dcrutil/util.go · VerifyMessage

Public-key recovery inverts the ECDSA verification equation (SEC 1 [14], section 4.1.6) using the recovery code to select the nonce point:

ALGORITHM 5.6 Compact signature public-key recovery

```
def recover_compact(sig: bytes, h: bytes):    # N: group order, P: field prime
    if len(sig) != 65:                  fail()
    code = sig[0]
    if code < 27 or code > 34:          fail()
    code = code - 27
    compressed = (code & 4) != 0
    r, s = int_from_be(sig[1:33]), int_from_be(sig[33:65])
    if r >= N or r == 0 or s >= N or s == 0:
        fail()
    x = r
    if code & 2:
        if r + N >= P:                  fail()
        x = r + N
    Rpt = point_with_x(x, odd_y=(code & 1) != 0)   # fail if x is not on the curve
    e = int_from_be(h) % N
    w = inverse(r, N)
    Q = ((-e * w) % N) * G + ((s * w) % N) * Rpt
    if Q is infinity:                   fail()
    return Q, compressed
```

dcrd: dcrec/secp256k1/ecdsa/signature.go · RecoverCompact

The recovery does not require $s \leq n/2$; high-s signatures verify.

TEST VECTOR Signed message

Item	Value
Address	`DshRCvfb21VQvaXZYhBivFS7VAurvcKi32t` (private key `01 02 … 1f 20`)
Message	`Decred` (6 bytes)
Preimage	17446563726564205369676e6564204d6573736167653a0a06446563726564
Digest h	e24e7c60221c11a8176e2e7d912cb947e7f68891ed0de3ebe71fa34298041198
Signature	1f43c089ab3b869b04bd6c709418df8be7c9d938bc5e853999c719f8694fc2f5a246cd5e8dd8f23e39422b6238b7da784beb69a28e090cf2e523ccca6f1f901cc2
Base64	`H0PAias7hpsEvWxwlBjfi+fJ2Ti8XoU5mccZ+GlPwvWiRs1ejdjyPjlCK2I4t9p4S+tpoo4JDPLlI8zKbx+QHMI=`

The header byte `0x1f` = 31 = 27 + 0 + 4: compressed key, recovery code 0.

5.8 Hierarchical Deterministic Keys

Hierarchical deterministic keys derive an unbounded tree of key pairs from a single seed, so that a wallet can be backed up once. Each node of the tree is an **extended key**: a private or public key together with a 32-byte **chain code**. Decred uses the BIP32 construction [24] with the changes listed in Table 5.9. Of these, only the stripping of leading zeros changes which keys are derived; the others change only the serialized form.

Table 5.9 Differences between Decred HD keys and BIP32

Aspect	Decred behaviour
Fingerprint	First 4 bytes of HASH160 of the parent's compressed public key, with Decred's BLAKE-256-based HASH160.
Checksum	First 4 bytes of $\mathrm{BLAKE256}(\mathrm{BLAKE256}(\cdot))$ of the 78-byte serialization.
Version bytes	Per network, Table 5.2.
Private child key	Leading zero bytes of the derived private key are stripped, which changes the input to every **hardened** child of that key (Section 5.8.2).
Master key	Unchanged: HMAC-SHA512 keyed with the ASCII string `Bitcoin seed`.

Because the master key and the derivation function are otherwise BIP32's, the master private key and chain code for a given seed equal BIP32's, and so do the keys of any path that never meets a stripped key. The serialized strings always differ (version, fingerprint and checksum).

5.8.1 Master key generation

ALGORITHM 5.7 **Master key from seed**

```
def new_master(seed: bytes, net) -> ExtKey:
    if len(seed) < 16 or len(seed) > 64:          # 128 to 512 bits; 32 recommended
        error("invalid seed length")
    I = HMAC_SHA512(key=b"Bitcoin seed", data=seed)
    IL, IR = I[0:32], I[32:64]
    k = int_from_be(IL)
    if k >= N or k == 0:
        error("unusable seed")                    # choose another seed
    return ExtKey(private=True, key=IL,           # 32 bytes, never stripped
                  chain_code=IR, depth=0, parent_fp=b"\x00" * 4, child=0)
```

dcrd: hdkeychain/extendedkey.go · NewMaster

5.8.2 Child key derivation

Child indices are 32-bit unsigned integers. Indices from 0 to $2^{31}-1$ are **normal**; indices from 2^{31} (`HardenedKeyStart` $= 2^{31} =$ `0x80000000`) to $2^{32}-1$ are **hardened** and are written with a prime, so $0'$ is index 2^{31}. A hardened child can only be derived from an extended private key; a normal child of a public key can be derived from the public key alone, and equals the public key of the corresponding private child.

To describe Decred's variation exactly, an extended private key is taken to hold its private key as a byte string `key` of 1 to 32 bytes whose big-endian value is the private scalar. A master key, and any key parsed from its serialized form, holds exactly 32 bytes. A private key produced by Decred child derivation holds the 32-byte big-endian value **with its leading zero bytes removed**. The parent's bytes enter the HMAC input of a hardened child left-justified, followed by zero padding, so a stripped key produces a different child from the standard BIP32 one.

ALGORITHM 5.8 **Child key derivation (Decred variant)**

```
def ckd(parent: ExtKey, i: uint32, strict: bool = False) -> ExtKey:
    hardened = i >= 0x80000000
    if hardened and not parent.private:
        error("cannot derive a hardened key from a public key")
    if hardened:
        # 0x00, then the parent key bytes LEFT-justified in 32 bytes, then i.
        # Identical to BIP32's 0x00 || ser256(k) only when len(parent.key) == 32.
        data = b"\x00" + parent.key + b"\x00" * (32 - len(parent.key)) + be32(i)
    else:
```

```
        data = pubkey(parent) + be32(i)           # 33-byte compressed key
    I = HMAC_SHA512(key=parent.chain_code, data=data)
    IL, IR = I[0:32], I[32:64]
    t = int_from_be(IL)
    if t >= N or t == 0:
        error("invalid child")                    # caller skips to index i + 1
    if parent.private:
        k = (t + int_from_be(parent.key)) % N     # k == 0 is not checked
        key = be256(k)                            # 32 bytes
        if not strict:
            key = strip_leading_zero_bytes(key)   # Decred variant (Child)
    else:
        key = ser_compressed(t * G + parse_point(parent.key))
    return ExtKey(private=parent.private, key=key, chain_code=IR,
                  depth=parent.depth + 1, child=i,
                  parent_fp=HASH160(pubkey(parent))[0:4])

def pubkey(k: ExtKey) -> bytes:
    if k.private:
        return ser_compressed((int_from_be(k.key) % N) * G)
    return k.key
```

dcrd: hdkeychain/extendedkey.go · child | hdkeychain/extendedkey.go · Child, ChildBIP32Std

dcrd exposes the Decred variant as `Child` and a strictly conforming variant (`strict = True`, never stripping) as `ChildBIP32Std`. The flag affects only the key that is returned: `ChildBIP32Std` applied to a parent that `Child` has already stripped still places the stripped bytes left-justified in the HMAC input, exactly as `Child` does. **Decred wallets use `Child`**, and an implementation that must reproduce existing wallets' addresses **MUST** implement the stripping and the left-justified layout exactly as above.

IMPLEMENTATION PITFALL
The result depends on how the parent was obtained, not only on its value. The same private key yields different hardened children when it was just derived by `Child` (stripped) and when it was parsed from a serialized extended key (32 bytes). For example, deriving $1'$ from the vector-4 key $m/0'$ below gives one result if $m/0'$ was derived in memory and the strict BIP32 result if $m/0'$ was first serialized and parsed. The serialized form always carries the full 32 bytes. Normal children are unaffected, because they depend only on the public key.

IMPLEMENTATION PITFALL
In the wallet path of Section 5.8.4, the dcrwallet source examined for this document (its development branch as of November 2026) derives the purpose key $m/44'$, both coin-type keys and account 0 in memory with `Child` when the wallet is created or restored, but derives every later account, and the candidate accounts that account discovery scans during a restore, from the coin-type key after it has been serialized, stored and parsed. Hence the coin-type key depends on whether $m/44'$ begins with a zero byte, account 0 depends on whether the coin-type key does (each with probability about $\frac{1}{256}$), and accounts 1 and above are derived from the full 32-byte coin-type key, as in strict BIP32. A wallet that must restore dcrwallet accounts reproduces this exactly.

RATIONALE
The stripping is a legacy behaviour, apparently dating from the original Decred port of the BIP32 code. The dcrd source keeps it because changing it "would break derivation for a lot of Decred wallets that rely on this behavior".

Further details of dcrd's behaviour, relevant only in edge cases:

- A child index whose `IL` is not a valid scalar is reported as invalid and wallets move to the next index; the probability is below 2^{-127}.
- dcrd does not check that a private child key is non-zero or that a public child point is not the point at infinity, which BIP32 requires; both have negligible probability.

- The depth is kept as a 16-bit number in memory but serialized modulo 256.

5.8.3 Serialization

Table 5.10 Extended key serialization (78 bytes, then a 4-byte checksum, then Base58)

Field	Type	Bytes	Description
version	[4]byte	4	HDPrivateKeyID for a private key, HDPublicKeyID for a public key.
depth	uint8	1	0 for the master key, 1 for its children, and so on.
parent_fp	[4]byte	4	Parent fingerprint; zero for the master key.
child	uint32 BE	4	Index of this key in its parent; 0 for the master key.
chain_code	[32]byte	32	Chain code.
key_data	[33]byte	33	Private: 0x00 followed by the private key left-padded with zeros to 32 bytes. Public: the 33-byte compressed public key.
checksum	[4]byte	4	$\mathrm{BLAKE256}(\mathrm{BLAKE256}(\cdot))[0:4]$ of the preceding 78 bytes.

The 82 bytes are Base58-encoded as a whole; there is no separate Base58Check prefix. Parsing reverses this:

ALGORITHM 5.9 **Parsing a serialized extended key**

```
def parse_ext_key(s: str, net) -> ExtKey:
    if len(s) > 113:                        # 82 * 137 // 100 + 1
        error("invalid key length")
    d = base58_decode(s)
    if len(d) != 82:
        error("invalid key length")
    if BLAKE256(BLAKE256(d[0:78]))[0:4] != d[78:82]:
        error("bad checksum")
    if d[0:4] not in (net.HDPrivateKeyID, net.HDPublicKeyID):
        error("wrong network")
    depth, parent_fp = d[4], d[5:9]
    child, chain_code, key_data = be32_to_int(d[9:13]), d[13:45], d[45:78]
    if key_data[0] == 0x00:                 # private, regardless of the version bytes
        key = key_data[1:33]                # 32 bytes, leading zeros kept
        if int_from_be(key) >= N or int_from_be(key) == 0:
            error("unusable seed")
        return ExtKey(private=True, key=key, ...)
    require_valid_compressed_secp256k1_point(key_data)
    return ExtKey(private=False, key=key_data, ...)
```

dcrd: hdkeychain/extendedkey.go · NewKeyFromString

IMPLEMENTATION PITFALL
dcrd decides whether a parsed key is private from the first byte of the key data, not from the version bytes, and accepts either of the network's two versions with either kind of key; re-serializing such a key writes the version that matches its kind, not the one it was parsed with. It also does not check that a depth-0 key has a zero fingerprint and child number. Strict implementations that add these checks will reject strings that dcrd accepts; no conforming encoder produces such strings.

5.8.4 Wallet account structure (convention)

The layout of the key tree is a wallet convention, not part of dcrd. dcrwallet follows BIP43 [25] and BIP44 [26]:

$$m/44'/\text{coin_type}'/\text{account}'/\ \text{branch}\ /\ \text{index} \tag{5.3}$$

- **Coin type** is SLIP0044CoinType from the SLIP-0044 registry [27]: 42 on the main network and 1 ("all test networks") elsewhere. Older wallets used the coin types now kept as LegacyCoinType (20, 11, 115 and 1 for mainnet, testnet3, simnet and regnet). dcrwallet stores the keys of both coin types, starts on

the legacy one for backward compatibility, and switches to the SLIP-0044 coin type when no legacy address or account has been used: immediately for a newly generated seed, and after address discovery for a restored seed. Wallet restoration therefore **SHOULD** scan both coin types.

- **Account** numbers run from 0 (the default account) to $2^{31} - 2$; dcrwallet reserves $2^{31} - 1$ for imported keys, which are not HD-derived.
- **Branch** 0 is the external branch (receiving addresses) and branch 1 the internal branch (change). Both are normal children, so an account's extended public key suffices to derive all of its addresses.
- **Index** runs from 0 upward. dcrwallet's default gap limit, the number of consecutive unused addresses after which address discovery stops, is 20.

A wallet address is the P2PKH-ECDSA address of the compressed public key at the leaf. dcrwallet's mnemonic seed (33 words from the PGP word list: 32 encoding a 256-bit seed and one checksum word) is outside the scope of this document.

5.8.5 Test vectors

TEST VECTOR HD chain from the BIP32 test vector 1 seed (main network)
Seed `000102030405060708090a0b0c0d0e0f` (the seed of BIP32 test vector 1). No private key on the path $m/0'/1/2'$ begins with a zero byte, so its keys and chain codes are those of standard BIP32; the fingerprints, checksums and strings are Decred's.

Item	Value
m private	`e8f32e723decf4051aefac8e2c93c9c5b214313817cdb01a1494b917c8436b35`
m chain code	`873dff81c02f525623fd1fe5167eac3a55a049de3d314bb42ee227ffed37d508`
m public	`0339a36013301597daef41fbe593a02cc513d0b55527ec2df1050e2e8ff49c85c2`
m serialized (private)	`02fda4e8000000000000000000873dff81c02f525623fd1fe5167eac3a55a049de3d314bb42ee227ffed37d50800e8f32e723decf4051aefac8e2c93c9c5b214313817cdb01a1494b917c8436b3530117b28`
m dprv	`dprv3hCznBesA6jBtmoyVFPfyMSZ1qYZ3WdjdebquvkEfmRfxC9VFEFi2YDaJqHnx7uGe75eGSa3Mn3oHK11hBW7KZUrPxwbCPBmuCi1nwm182s`
m dpub	`dpubZ9169KDAEUnyoBhjjmT2VaEodr6pUTDoqCEAeqgbfr2JfkB88BbK77jbTYbcYXb2FVz7DKBdW4P618yd51MwF8DjKVopSbS7Lkgi6bowX5w`
Fingerprint of m	`bc495588` (first 4 bytes of `bc495588ca4ffd88706347cce9791fc800e13535`)
$m/0'$ dprv	`dprv3kUQDBztdyjKuwnaL3hfKYpT7W6X2huYH5d61YSWFBebSYwEBHAXJkCpQ7rvMAxPzKqxVCGLvBqWvGxXjAyMJsV1XwKkfnQCM9KctC8k8bk`
$m/0'$ dpub	`dpubZCGVaKZBiMo7pMgLaZm1qmchjWenTeVcUdFQkTNsFGFEA6xs4EW8PKiqYqP7HBAitt9Hw16VQkQ1tjsZQSHNWFc6bEK6bLqrbco24FzBTY4`
$m/0'/1$ dprv	`dprv3nRtCZ5VAoHW4RUwQgRafSNRPUDFrmsgyY71A5eoZceVfuyL9SbZe2rcbwDW2UwpkEniE4urffgbypegscNchPajWzy9QS4cRxF8QYXsZtq`
$m/0'/1$ dpub	`dpubZEDyZgdnFBMHxqNhfCUwBfAg1UmXHiTmB5jKtzbAZhF8PTzy2PwAicNdkg1CmW6TARxQeUbgC7nAQenJts4YoG3KMiqcjsjgeMvwLc43w6C`
$m/0'/1/2'$ dprv	`dprv3pYtkZK168vgrU38gXkUSjHQ2LGpEUzQ9fXrR8fGUR59YviSnm6U82XjQYhpJEUPnVcC9bguJBQU5xVM4VFcDHu9BgScGPA6mQMH4bn5Cth`
$m/0'/1/2'$ dpub	`dpubZGLz7gsJAWzUksvtw3opxx5eeLq5fRaUMDABA3bdUVfnGUk5fiS5Cc3kZGTjWtYr3jrEavQQnAF6jv2WCpZtFX4uFgifXqev6ED1TM9rTCB`
$m/44'/42'/0'/0/0$ private	`062030a1065be57e24b89fad7413b576b5d06a2c11faff137b9c437649d62e08`
$m/44'/42'/0'/0/0$ public	`03976beeebefe06a527bdd1ebbe8e9077f388df42aa29b14348bff6ff7da8dabc1`
$m/44'/42'/0'/0/0$ address	`DsVxTiRYVKtUQogbS6prKdQvFb4dUwy7KyM` (all steps derived in memory with `Child`)

Deriving $m/0'/1$ from the neutered $m/0'$ gives the same dpub as neutering the private $m/0'/1$.

TEST VECTOR Leading-zero stripping (BIP32 test vector 4 seed, main network)
Seed `3ddd5602285899a946114506157c7997e5444528f3003f6134712147db19b678`. The private key of $m/0'$ is `00d948e9261e41362a688b916f297121ba6bfb2274a3575ac0e456551dfd7f7e`, which begins with a zero byte; `Child` holds it as the 31 bytes after that zero. The chain code of $m/0'$ is `cdc0f06456a14876c898790e0b3b1a41`

`c531170aec69da44ff7b7265bfe7743b`. Both methods serialize $m/0'$ identically: `dprv3jkqwzsd88yXqZJdSuLnfKqGPYzCVffataF79tcuPhdudeU3d1uUpvwn5Btmbai3BoVYhzbsEQ7tcH1XtEJ1BhzNSAtZfqv8BtSpdq5FEpj`. The HMAC inputs and results for $m/0'/1'$ differ:

Item	Value
`Child` input	`00d948e9261e41362a688b916f297121ba6bfb2274a3575ac0e456551dfd7f7e0080000001`
`Child` private key	`bf33f8c13baf010213cd34f488297f7729d560871286b2e360b25bc9fecc96f4`
`Child` dprv	`dprv3mhK5vAEzovJTfnqwfzGTamym6yPPGEggq5fpQHrnXGyb22Doqew7Co7mCvYHe6nTCewnggesjM4LVJj8RFJ92ANtogs3NY7ev4YxSwf6ez`
Strict input	`0000d948e9261e41362a688b916f297121ba6bfb2274a3575ac0e456551dfd7f7e80000001`
Strict private key	`3a2086edd7d9df86c3487a5905a1712a9aa664bce8cc268141e07549eaa8661d`
Strict dprv	`dprv3mhK5vAEzovJUV5RycwKGbth8MYtoYL1aJTxN1T8PLmH6da6aMU8nYKQiC5Zj7mC7pH4G8xPxGZVR51FG9Ssptzx1c39A1JxADS6zDrsLPr`

Parsing the serialized $m/0'$ and then applying `Child(1')` yields the strict dprv. The normal child $m/0'/1$ is the same under both methods.

5.9 Amounts

All protocol amounts are integers of atoms, 10^8 atoms to the coin (Chapter 2). User interfaces and the JSON-RPC interface express amounts in coins as decimal numbers. dcrd's conversion helpers, which are not consensus code, behave as follows:

- Coins to atoms multiplies the double-precision value by 10^8, adds 0.5 (subtracts 0.5 if the product is negative), both in double precision, and truncates toward zero. NaN and infinities are rejected; the range is not checked, and an out-of-range value converts to an unspecified integer.
- Formatting divides the amount by the unit's number of atoms in double precision and writes the result with the shortest decimal representation that round-trips that double (with no decimal places for the Atom unit), followed by a space and the unit name of Table 5.11. For example, 44433322211100 atoms formats as `444333.222111 DCR`, `444.333222111 kDCR` or `44433322211100 Atom`.

Table 5.11 Amount units

Unit	Atoms	Name
Megacoin	10^{14}	`MDCR`
Kilocoin	10^{11}	`kDCR`
Coin	10^8	`DCR`
Millicoin	10^5	`mDCR`
Microcoin	10^2	`µDCR`
Atom	1	`Atom`

Implementations that handle amounts **SHOULD** convert to integer atoms at the boundary and never compute with floating-point coin values.

5.10 Bech32

dcrd contains a `bech32` package implementing the Bech32 encoding of BIP173 [28] (the original Bech32 checksum constant 1, not Bech32m), with the standard 32-character alphabet `qpzry9x8gf2tvdw0s3jn54khce6mua7l` and a 90-character limit in its standard decoder. **No Decred address format uses Bech32**, and no dcrd code calls the package (its only import is a blank import that keeps it in dcrd's module requirements); it is published for other software, such as Lightning Network payment requests, which its documentation names.

PART II

Scripts and Transactions

The stack machine that guards every output, and the rules a transaction must satisfy before it may spend one.

CHAPTER 6

The Script System

Every transaction output carries a short program, the *public key script*, that states the conditions under which the output may be spent. Every transaction input that spends an output carries a second program, the *signature script*, that supplies the data needed to satisfy those conditions: typically a signature and a public key. To decide whether an input may spend an output, a node runs the two programs one after the other on a simple stack machine. The input is valid only if the machine finishes without error and leaves a value that counts as true.

This chapter specifies that machine as consensus requires it: the byte-level encoding of scripts, every limit, the numeric and boolean conventions, the semantics of all 256 opcode values, signature verification and the signature hash, pay-to-script-hash, the stake and treasury tag opcodes, the verification flags, signature-operation counting, and a few script predicates that other consensus code relies on. Decred's script language descends from Bitcoin's, but it differs in many details that are invisible in the common cases and fatal in the uncommon ones. Nothing in this chapter should be assumed to match Bitcoin unless the text says so. Appendix A lists all 256 opcode values in a single table.

Which transactions are checked, which inputs are skipped and which flags are passed are decided by Chapter 7; the per-block signature operation limit is in Chapter 10; the cryptographic primitives (hash functions, DER parsing, ECDSA, EC-Schnorr-DCRv0, Ed25519) are in Chapter 3; the serialization of transactions is in Chapter 4.

6.1 Overview and Terminology

A **script** is a byte string. It is read as a sequence of **opcodes**, each introduced by a single byte. Some opcodes (the **push** opcodes) are followed by inline data that they place on the stack; all other opcodes occupy exactly one byte.

A transaction output holds an amount, a 16-bit **script version** and a public key script. A transaction input holds, among other fields, a signature script. When input i of transaction T spends output o, the script engine is invoked with:

- the public key script of o and its script version,
- the transaction T and the input index i (the signature script is T's input i's signature script),
- a set of **verification flags** (Section 6.13), chosen by the caller.

The engine maintains two stacks of byte strings. The **data stack** is the main working stack; the **alternate stack** is auxiliary storage reachable only through two opcodes. An entry on either stack is called an **element** or **item**. The **top** of a stack is its most recently pushed element. In stack diagrams, the stack is written left to right from bottom to top, so in `[... x1 x2]` the element `x2` is on top.

The result of a script check is binary. If the check fails, the transaction containing the input is invalid, and so is any block containing it. The reference implementation reports a specific **error kind** for each failure (for example `ErrEvalFalse`); these names are given in the text as an aid to implementers and for cross-checking against dcrd's test data, but only the binary outcome is consensus.

6.2 Script Versions

The script version of an output selects the language its public key script is written in. Only version 0 is defined. The signature script and, for pay-to-script-hash, the redeem script are interpreted under the version of the public key script they redeem.

RULE 6.1 Non-zero script versions

When the spent output's script version is not 0, the engine performs every check of Rule 6.3 (the engine construction checks), and if all of them pass, the input **MUST** be treated as successfully verified without executing any script. Construction checks that depend on version 0 semantics (the redeem-script checks of step 4) are skipped, but all others, including the requirement that both scripts parse under version 0 rules, apply regardless of the version.

`dcrd: txscript/engine.go · Execute | txscript/engine.go · NewEngine`

IMPLEMENTATION PITFALL

Outputs with a non-zero script version are therefore spendable by anyone who can supply a signature script that, together with the public key script, passes the construction checks: both scripts at most 16384 bytes, both parseable as version 0 scripts, not both empty, and, if the public key script has the byte pattern of a pay-to-script-hash script, a push-only signature script. Because the parse check uses version 0 rules, a public key script that does not parse as version 0 is unspendable at any version. Several other predicates used by consensus (pay-to-script-hash detection, signature operation counting, Rule 6.35) also ignore the version entirely.

HISTORY

Before the `explicitverupgrades` agenda (DCP-0008), any 16-bit script version was permitted in any output. Once that agenda is active, outputs of regular (non-stake) transactions must carry script version 0, and stake transactions are constrained to specific versions by their own rules. Those limits are transaction-level rules specified in Chapter 7 and Chapter 8; how agenda activation is computed is in Chapter 13. Outputs created earlier with non-zero versions remain in the UTXO set and are governed by Rule 6.1.

6.3 Parsing

Parsing splits a script into opcodes and their inline data. It is defined for version 0 and is used both by the engine and by every other predicate in this chapter. Opcode bytes fall into four parsing classes, shown in Table 6.1.

Table 6.1 Parsing classes of opcode bytes

Byte(s)	Opcode(s)	Encoding
`0x00`	`OP_0`	The single byte. Pushes the empty byte string.
`0x01` – `0x4b`	`OP_DATA_1` … `OP_DATA_75`	The byte n followed by exactly n data bytes.
`0x4c`	`OP_PUSHDATA1`	The byte, a 1-byte length n, then n data bytes.
`0x4d`	`OP_PUSHDATA2`	The byte, a 2-byte little-endian length n, then n data bytes.
`0x4e`	`OP_PUSHDATA4`	The byte, a 4-byte little-endian length n, then n data bytes.
`0x4f` – `0xff`	all others	The single byte. No inline data.

RULE 6.2 Script parsing

A script is parsed from offset 0. At each offset k the byte b at k is read and:

1. If b is `0x00` or $b \geq$ `0x4f`, the opcode is b with no data and the next offset is $k + 1$.
2. If `0x01` $\leq b \leq$ `0x4b`, the script **MUST** contain at least $1 + b$ bytes from offset k; the data is the b bytes after b and the next offset is $k + 1 + b$.

3. If b is `0x4c`, `0x4d` or `0x4e`, let w be 1, 2 or 4 respectively. The script **MUST** contain at least w bytes after offset k. Let n be those w bytes read as an unsigned little-endian integer. The script **MUST** contain at least n further bytes after the length field. The data is those n bytes and the next offset is $k + 1 + w + n$.

Parsing ends successfully when the offset equals the script length. Any violation of a **MUST** above is a **parse failure** (`ErrMalformedPush`). The empty script parses successfully to zero opcodes.

dcrd: txscript/tokenizer.go · ScriptTokenizer.Next

NOTE
The reference implementation reads the `OP_PUSHDATA4` length into a signed 32-bit integer and rejects negative values; since any length of 2^{31} or more also exceeds the bytes remaining in any script that fits in a transaction, the condition "at least n further bytes" is equivalent.

Parsing does not inspect what an opcode does: unassigned, reserved and disabled opcode bytes all parse as one-byte opcodes. In what follows, the **data** of an opcode means its inline data (empty for `OP_0` and for every opcode without inline data), and an opcode is a **data push** if its byte is at most `0x4e`. This is narrower than the **push-only** test of Section 6.5, which admits every opcode up to `0x60` and so also the small-integer opcodes.

ALGORITHM 6.1 Script parsing

```
def parse(script: bytes) -> list[tuple[int, bytes]]:
    ops, k, L = [], 0, len(script)
    while k < L:
        b = script[k]
        if b == 0x00 or b >= 0x4f:
            ops.append((b, b"")); k += 1
        elif b <= 0x4b:                          # OP_DATA_1 .. OP_DATA_75
            if L - k < 1 + b: reject("ErrMalformedPush")
            ops.append((b, script[k+1 : k+1+b])); k += 1 + b
        else:                                    # OP_PUSHDATA1/2/4
            w = {0x4c: 1, 0x4d: 2, 0x4e: 4}[b]
            if L - k - 1 < w: reject("ErrMalformedPush")
            n = int.from_bytes(script[k+1 : k+1+w], "little")
            if n > L - k - 1 - w: reject("ErrMalformedPush")
            ops.append((b, script[k+1+w : k+1+w+n])); k += 1 + w + n
    return ops
```

dcrd: txscript/tokenizer.go · ScriptTokenizer.Next

Several predicates walk a script only **up to** its first parse failure, using the opcodes successfully parsed before it. Where that matters the text says so explicitly.

6.4 Limits

Table 6.2 collects the numeric limits of the engine. Each is restated in the rule that enforces it, with the exact comparison.

Table 6.2 Script engine limits

Constant	Value	Meaning and exact comparison
`MaxScriptSize`	16384	A signature script or public key script longer than this many bytes fails (`len > 16384`). Redeem scripts are bounded by the element size instead.
`MaxScriptElementSize`	2048	A data push whose data is longer than this many bytes fails (`len > 2048`), whether or not the push is executed. `OP_CAT` enforces the same bound on its result.

Constant	Value	Meaning and exact comparison
`MaxOpsPerScript`	255	Per script, the count of opcodes with byte value greater than `0x60`, plus the key counts of executed multisignature checks, must not exceed 255 (fails when the count becomes `> 255`).
`MaxStackSize`	1024	After each opcode, the combined number of elements on the data stack and alternate stack must not exceed 1024 (fails when `> 1024`).
`MaxPubKeysPerMultiSig`	20	The key count of `OP_CHECKMULTISIG(VERIFY)` must be in $[0, 20]$.
`MathOpCodeMaxScriptNumLen`	4	Maximum encoded length of most numeric operands (Section 6.7.2).
`CltvMaxScriptNumLen`	5	Maximum encoded length of the `OP_CHECKLOCKTIMEVERIFY` operand.
`CsvMaxScriptNumLen`	5	Maximum encoded length of the `OP_CHECKSEQUENCEVERIFY` operand.
—	1	Maximum encoded length of the `OP_CHECKSIGALT(VERIFY)` signature type operand.
`LockTimeThreshold`	500000000	Lock-time values below this are block heights, at or above it Unix times.

6.5 Engine Construction

Before anything executes, the engine checks the pair of scripts as a whole. These checks are part of consensus: a failure makes the input invalid exactly as an execution failure does (dcrd reports it as `ErrScriptMalformed` at the transaction level, but the distinction has no consensus effect).

Two script shapes are recognized by exact byte pattern. A **pay-to-script-hash** (P2SH) public key script is exactly 23 bytes: `a9 14` followed by 20 bytes followed by `87`, that is, `OP_HASH160 OP_DATA_20 <20 bytes> OP_EQUAL`. A **stake-tagged P2SH** public key script is exactly 24 bytes: a **stake tag** byte followed by those 23 bytes. The stake tag bytes are `0xba` – `0xbd` (`OP_SSTX`, `OP_SSGEN`, `OP_SSRTX`, `OP_SSTXCHANGE`) and, only when the `ScriptVerifyTreasury` flag is set, also `0xc1` – `0xc3` (`OP_TADD`, `OP_TSPEND`, `OP_TGEN`). A script is **P2SH-shaped** if it is either. P2SH is specified in Section 6.11.

A script is **push-only** if it parses (Rule 6.2) and every opcode in it has byte value at most `0x60` (`OP_16`). Note that this includes `OP_RESERVED` (`0x50`), which fails if executed.

RULE 6.3 Engine construction checks

Let S be the signature script, P the public key script and v the script version of the spent output. The following **MUST** all hold:

1. S and P are not both empty (`ErrEvalFalse`).
2. If the flag `ScriptVerifySigPushOnly` is set, S is push-only (`ErrNotPushOnly`).
3. If P is P2SH-shaped, S is push-only (`ErrNotPushOnly`). This check is independent of v.
4. If $v = 0$ and P is P2SH-shaped: let R be the data of the final opcode of S, taken from a full parse of S (if S is empty or fails to parse, R is empty). R **MUST** be non-empty (`ErrNotPushOnly`). R **MUST NOT** contain a stake tag opcode, where the opcodes considered are `0xba` – `0xbd`, plus `0xc1` – `0xc3` when `ScriptVerifyTreasury` is set (`ErrP2SHStakeOpCodes`); this scan stops at the first parse failure of R, and a parse failure reached before any stake tag opcode also fails (`ErrMalformedPush`).
5. $|S| \leq 16384$ and $|P| \leq 16384$ (`ErrScriptTooBig`).
6. S and P both parse under version 0 rules (`ErrMalformedPush`), whatever v is.

dcrd: txscript/engine.go · NewEngine | txscript/script.go · hasP2SHRedeemScriptStakeOpCodes

IMPLEMENTATION PITFALL

In step 4 the redeem script is the *inline data* of the last opcode of the signature script, not the top stack element the signature script will produce. The small-integer opcodes `OP_1` – `OP_16` and `OP_1NEGATE` have no inline data, so a P2SH signature script that ends with one of them fails construction even though executing it would leave a non-empty element on the stack. The same applies to `OP_0` and to an empty push such as `OP_PUSHDATA1 0x00`. A redeem script consisting of the single byte `0x51` must therefore be supplied as `01 51` (an `OP_DATA_1` push), never as `51`.

6.6 Execution Model

6.6.1 Order of Execution

The engine executes up to three scripts in sequence on a shared data stack:

1. the signature script S, starting from an empty data stack;
2. the public key script P, starting from the data stack S left behind;
3. for P2SH only, the **redeem script** R: the top element of the data stack as it was after S finished, executed on that stack with R itself removed (Section 6.11).

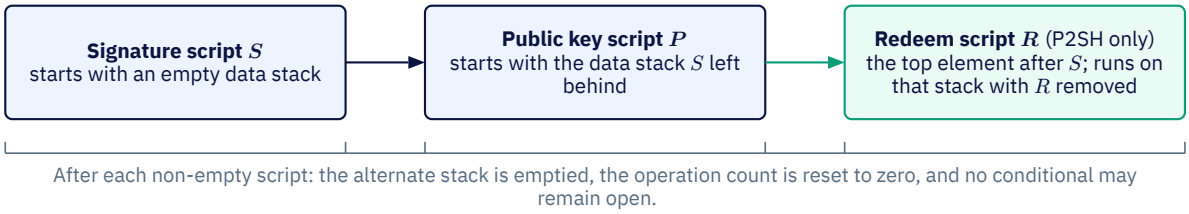


Figure 6.1 Order of script execution. The data stack is carried from one script to the next; the alternate stack, the operation count and the conditional state belong to a single script.

An empty script is skipped entirely: it executes no opcodes and none of the per-script end checks below apply to it. The alternate stack, the operation count and the conditional state are per script: the alternate stack is emptied and the operation count reset to zero at the end of each non-empty script.

RULE 6.4 Per-script end conditions

When the last opcode of a non-empty script has executed:

1. the conditional nesting depth (Section 6.6.3) **MUST** be 0 (`ErrUnbalancedConditional`): an `OP_IF` or `OP_NOTIF` block cannot span two scripts;
2. the alternate stack is then emptied, and the operation count reset to 0, before the next script starts.

`dcrd: txscript/engine.go · Engine.Step`

RULE 6.5 Combined stack size

After every opcode the engine processes, including opcodes in unexecuted branches, the number of elements on the data stack plus the number on the alternate stack **MUST NOT** exceed 1024 (`ErrStackOverflow`).

`dcrd: txscript/engine.go · Engine.Step`

6.6.2 Processing One Opcode

The engine walks each script opcode by opcode in parse order. Every opcode the program counter reaches is **processed**, whether or not it lies in a branch that is being executed; processing decides whether the opcode's operation is actually **executed**. A branch is **executing** when no enclosing `OP_IF` , `OP_NOTIF` or `OP_ELSE` has disabled it (Section 6.6.3).

RULE 6.6 Opcode processing

For each opcode o with data d reached by the program counter, in this order:

1. If o is `OP_CODESEPARATOR` (`0xab`), the script **MUST** fail (`ErrDisabledOpcode`), whether or not the branch is executing.
2. If o is `OP_VERIF` (`0x65`) or `OP_VERNOTIF` (`0x66`), the script **MUST** fail (`ErrReservedOpcode`), whether or not the branch is executing.

3. If $o >$ `0x60`, the per-script operation count is incremented by one, and if it now exceeds 255 the script **MUST** fail (`ErrTooManyOperations`). Otherwise ($o \leq$ `0x60`), if $|d| > 2048$ the script **MUST** fail (`ErrElementTooBig`). Both checks apply whether or not the branch is executing.
4. If the branch is not executing and o is not one of `OP_IF`, `OP_NOTIF`, `OP_ELSE`, `OP_ENDIF`, processing of o ends here: it has no further effect.
5. If the branch is executing and $o \leq$ `0x4e`, the push **MUST** be minimal (Rule 6.7).
6. The operation of o is executed, as specified in Section 6.8 and Appendix A. The conditional opcodes are executed in unexecuted branches as well, with the behaviour given in Section 6.6.3.

dcrd: txscript/engine.go · executeOpcode
txscript/engine.go · isOpcodeDisabled
txscript/engine.go · isOpcodeAlwaysIllegal

IMPLEMENTATION PITFALL
The operation count includes every opcode above `0x60` that the program counter passes over, including those in unexecuted branches, and excludes every opcode at or below `0x60`, including `OP_RESERVED` (`0x50`). The element size limit likewise applies to pushes in unexecuted branches. By contrast, the minimal-push rule applies only to executed pushes.

RULE 6.7 Minimal data pushes
An executed data push with opcode o (`0x00` – `0x4e`) and data d **MUST** satisfy the first matching condition (`ErrMinimalData`):

1. if $|d| = 0$: o is `OP_0`;
2. if $|d| = 1$ and $1 \leq d[0] \leq 16$: o is `OP_1` $+ d[0] - 1$ (that is, a small-integer opcode, which is not a data push and so never reaches this check);
3. if $|d| = 1$ and $d[0] =$ `0x81`: o is `OP_1NEGATE` (likewise impossible to satisfy with a data push);
4. if $|d| \leq 75$: $o = |d|$ (an `OP_DATA_n` opcode);
5. if $|d| \leq 255$: o is `OP_PUSHDATA1`;
6. if $|d| \leq 65535$: o is `OP_PUSHDATA2`.

dcrd: txscript/engine.go · checkMinimalDataPush

In effect, every executed push must use the shortest encoding: the empty string only via `OP_0`; the single bytes `0x01` – `0x10` and `0x81` only via `OP_1` – `OP_16` and `OP_1NEGATE`; other strings of 1 to 75 bytes only via `OP_DATA_n`; 76 to 255 bytes only via `OP_PUSHDATA1`; 256 to 2048 bytes only via `OP_PUSHDATA2`. An executed `OP_PUSHDATA4` always fails, because a push of at most 65535 bytes requires a shorter opcode and a longer one cannot fit in a script. The single bytes `0x00` and `0x80` may be pushed with `OP_DATA_1`.

6.6.3 Conditional Execution

Branching uses two per-script integers: the **nesting depth** D, the number of currently open `OP_IF` / `OP_NOTIF` blocks, and the **disable depth** X, which is either "none" (the branch is executing) or the nesting depth at which execution was disabled. Both start at $D = 0$, $X =$ none. Since D must return to 0 at the end of every script, both are effectively reset between scripts.

RULE 6.8 Conditional opcodes

- `OP_IF`: if X is none, pop the top data stack element (an empty stack fails with `ErrInvalidStackOperation`) and interpret it as a boolean (Rule 6.10); if it is false, set $X := D$. In all cases then set $D := D + 1$.
- `OP_NOTIF`: as `OP_IF`, except that $X := D$ is set when the popped value is true.
- `OP_ELSE`: if $D = 0$ the script **MUST** fail (`ErrUnbalancedConditional`). Otherwise let $d = D - 1$. If X is none, set $X := d$; else if $X = d$, set $X :=$ none; otherwise do nothing.

- `OP_ENDIF`: if $D = 0$ the script **MUST** fail (`ErrUnbalancedConditional`). Otherwise set $D := D - 1$, and then if $X = D$ set $X :=$ none.

dcrd: txscript/opcode.go · opcodeIf
txscript/opcode.go · opcodeNotIf
txscript/opcode.go · opcodeElse
txscript/opcode.go · opcodeEndif

Consequences worth stating explicitly: an `OP_IF` reached in an unexecuted branch consumes nothing from the stack but still opens a block; any number of `OP_ELSE` opcodes may appear in one block, each toggling execution; and the condition value is interpreted as a boolean, with no requirement that it be minimally encoded (Decred has no "minimal if" rule).

6.6.4 Successful Completion

RULE 6.9 Final result

After the last script has finished:

1. if the flag `ScriptVerifyCleanStack` is set, the data stack **MUST** contain exactly one element (`ErrCleanStack`);
2. otherwise the data stack **MUST** contain at least one element (`ErrEmptyStack`);
3. the top element, interpreted as a boolean, **MUST** be true (`ErrEvalFalse`).

dcrd: txscript/engine.go · CheckErrorCondition | txscript/engine.go · Engine.Execute

The clean-stack flag is part of the consensus flag set on every block (Section 6.13), so in consensus validation the data stack must end with exactly one element, which must be true. Contents of the alternate stack at the end are irrelevant (the alternate stack has been emptied by Rule 6.4).

For P2SH, an intermediate check precedes the redeem script; it is given in Rule 6.31.

6.6.5 Complete Input Verification

Algorithm 6.2 combines the rules of this section into a single procedure. The handlers for individual opcodes are specified in Section 6.8 and Section 6.9.

ALGORITHM 6.2 Verifying one input

```
MAX_SCRIPT_SIZE, MAX_ELEMENT, MAX_OPS, MAX_STACK = 16384, 2048, 255, 1024

def verify_input(tx, idx, pk_script, version, flags):
    S, P = tx.inputs[idx].sig_script, pk_script
    treasury = TREASURY in flags
    if len(S) == 0 and len(P) == 0: reject("ErrEvalFalse")
    if SIGPUSHONLY in flags and not is_push_only(S): reject("ErrNotPushOnly")
    p2sh = is_p2sh(P) or is_stake_p2sh(P, treasury)
    if p2sh and not is_push_only(S): reject("ErrNotPushOnly")
    if version == 0 and p2sh:
        R = final_opcode_data(S)            # b"" if S is empty or unparseable
        if len(R) == 0: reject("ErrNotPushOnly")
        # contains_stake_opcode tokenizes R and rejects with ErrMalformedPush
        # if R does not parse, unless a stake opcode is found first
        if contains_stake_opcode(R, treasury): reject("ErrP2SHStakeOpCodes")
    for s in (S, P):
        if len(s) > MAX_SCRIPT_SIZE: reject("ErrScriptTooBig")
        parse(s)                            # version 0 rules, may reject
    if version != 0:
        return                              # valid (rule: non-zero versions)
    vm = VM(tx, idx, flags)                 # empty data and alternate stacks
    run_script(vm, S)
    saved = list(vm.dstack)                 # used only for P2SH
    run_script(vm, P)
```

```
    if p2sh:
        if len(vm.dstack) < 1: reject("ErrEmptyStack")
        if not as_bool(vm.dstack.pop()): reject("ErrEvalFalse")
        R = saved[-1]
        parse(R)                          # redundant for version 0: checked above
        vm.dstack = saved[:-1]
        run_script(vm, R)
    if CLEANSTACK in flags and len(vm.dstack) != 1: reject("ErrCleanStack")
    if len(vm.dstack) < 1: reject("ErrEmptyStack")
    if not as_bool(vm.dstack.pop()): reject("ErrEvalFalse")

def run_script(vm, script):
    if len(script) == 0:
        return
    vm.script = script                    # the "current script"
    vm.num_ops, vm.depth, vm.disabled_at = 0, 0, None
    for op, data in parse(script):
        process_opcode(vm, op, data)
        if len(vm.dstack) + len(vm.astack) > MAX_STACK:
            reject("ErrStackOverflow")
    if vm.depth != 0: reject("ErrUnbalancedConditional")
    vm.astack = []

def process_opcode(vm, op, data):
    if op == 0xab: reject("ErrDisabledOpcode")          # OP_CODESEPARATOR
    if op in (0x65, 0x66): reject("ErrReservedOpcode")  # OP_VERIF, OP_VERNOTIF
    if op > 0x60:
        vm.num_ops += 1
        if vm.num_ops > MAX_OPS: reject("ErrTooManyOperations")
    elif len(data) > MAX_ELEMENT:
        reject("ErrElementTooBig")
    executing = vm.disabled_at is None
    if not executing and op not in (OP_IF, OP_NOTIF, OP_ELSE, OP_ENDIF):
        return
    if executing and op <= 0x4e:
        check_minimal_push(op, data)
    HANDLERS[op](vm, data)
```

dcrd: txscript/engine.go · NewEngine
txscript/engine.go · Engine.Step
txscript/engine.go · Engine.Execute

NOTE

For a version 0 P2SH spend, `saved[-1]` and `final_opcode_data(S)` are always the same byte string: S is push-only, construction has ensured its last opcode is a non-empty data push, and that push is the top of the stack S produces.

6.6.6 Example: Pay-to-Pubkey-Hash

The most common public key script is pay-to-pubkey-hash:

`OP_DUP OP_HASH160 <20-byte hash> OP_EQUALVERIFY OP_CHECKSIG`

It is redeemed by a signature script `<signature> <public key>`. Table 6.3 shows the data stack after each opcode.

Table 6.3 Execution of a pay-to-pubkey-hash spend

Script	Opcode	Data stack afterwards (bottom … top)
S	`OP_DATA_72`	`sig`
S	`OP_DATA_33`	`sig pk`
P	`OP_DUP`	`sig pk pk`

Script	Opcode	Data stack afterwards (bottom … top)
P	`OP_HASH160`	`sig pk H(pk)` where `H` = RIPEMD-160(BLAKE-256(·))
P	`OP_DATA_20`	`sig pk H(pk) h`
P	`OP_EQUALVERIFY`	`sig pk` (fails unless `H(pk) = h`)
P	`OP_CHECKSIG`	`1` if the signature verifies, else the empty string
—	final check	exactly one element, true: valid

6.7 Data Representation

Stack elements are untyped byte strings. Opcodes interpret them as raw bytes, as booleans or as numbers, and the interpretation is fixed per opcode.

6.7.1 Booleans

RULE 6.10 Boolean interpretation

A byte string x is **false** if every byte of x is `0x00`, or if every byte except the last is `0x00` and the last byte is `0x80` ("negative zero"). In particular the empty string is false. Every other byte string is **true**. There is no length limit on a value interpreted as a boolean.

An opcode that **pushes a boolean** pushes the one-byte string `01` for true and the empty string for false.

dcrd: txscript/stack.go · asBool

Booleans are consumed by `OP_IF`, `OP_NOTIF`, `OP_VERIFY`, `OP_IFDUP`, the `…VERIFY` forms of other opcodes, and the final check. `OP_NOT`, `OP_BOOLAND` and `OP_BOOLOR` are **numeric** opcodes and do not use this interpretation.

6.7.2 Script Numbers

Numbers on the stack are encoded as little-endian sign-magnitude byte strings: the magnitude is written in the fewest bytes possible, least significant byte first, and the most significant bit of the last byte is the sign. If the magnitude's most significant byte already has its top bit set, an extra byte `0x00` (positive) or `0x80` (negative) is appended. Zero is the empty string.

ALGORITHM 6.3 Script number encoding and decoding

```
def encode_num(n: int64) -> bytes:
    if n == 0:
        return b""
    neg, m = n < 0, abs(n)
    out = bytearray()
    while m > 0:
        out.append(m & 0xff); m >>= 8
    if out[-1] & 0x80:
        out.append(0x80 if neg else 0x00)
    elif neg:
        out[-1] |= 0x80
    return bytes(out)

def decode_num(b: bytes, max_len: int) -> int:
    if len(b) > max_len:
        reject("ErrNumOutOfRange")
    if len(b) > 0 and (b[-1] & 0x7f) == 0:          # minimal encoding
        if len(b) == 1 or (b[-2] & 0x80) == 0:
            reject("ErrMinimalData")
    if len(b) == 0:
        return 0
```

```
    mag = int.from_bytes(b, "little")
    if b[-1] & 0x80:
        return -(mag & ~(0x80 << (8 * (len(b) - 1))))
    return mag
```

dcrd: txscript/scriptnum.go · ScriptNum.Bytes
txscript/scriptnum.go · MakeScriptNum
txscript/scriptnum.go · checkMinimalDataEncoding

RULE 6.11 Numeric operands

Whenever an opcode interprets a stack element as a number, the element **MUST** decode under `decode_num` of Algorithm 6.3 with the maximum length for that opcode: its length **MUST NOT** exceed the maximum (`ErrNumOutOfRange`) and it **MUST** be minimally encoded (`ErrMinimalData`). The length check precedes the minimality check. The maximum length is 5 for the operand of `OP_CHECKLOCKTIMEVERIFY` and `OP_CHECKSEQUENCEVERIFY` , 1 for the signature-type operand of `OP_CHECKSIGALT(VERIFY)` , and 4 for every other numeric operand.

dcrd: txscript/scriptnum.go · MakeScriptNum | txscript/stack.go · stack.PopInt

A maximum length of 4 bytes admits exactly the integers $[-(2^{31}-1), 2^{31}-1]$; 5 bytes admits $[-(2^{39}-1), 2^{39}-1]$; 1 byte admits $[-127, 127]$. Note that -2^{31} is **not** a valid 4-byte operand: its encoding `00 00 00 80 80` is 5 bytes long.

The minimality requirement applies only when a value is interpreted as a number. Pushing the two-byte string `05 00` (with `OP_DATA_2` , which is the minimal **push** for it) is permitted even though it is a non-minimal encoding of the number 5; the element may be compared with `OP_EQUAL` , hashed or used as a boolean, and fails only if a numeric opcode consumes it.

Results of numeric opcodes are always pushed with `encode_num` , which never produces a non-minimal encoding. Results are not range-checked: an addition of two 4-byte operands may produce a 5-byte result such as $2^{32}-2$, which can be pushed, compared bytewise, hashed or interpreted as a boolean, but fails if it is later used as a 4-byte numeric operand.

NOTE

The reference implementation converts decoded operands to 32-bit integers with a saturating conversion (values above $2^{31}-1$ become $2^{31}-1$, values below -2^{31} become -2^{31}) wherever an operand is used as a 32-bit quantity. Because every operand so converted has already passed a 4-byte length check, it always lies in $[-(2^{31}-1), 2^{31}-1]$ and the saturation never changes a value. An implementation may use a plain conversion.

6.8 Opcode Semantics

This section specifies the operation performed when an opcode is **executed** (step 6 of Rule 6.6). All opcodes are listed by byte value in Appendix A. Unless stated otherwise:

- "pop" removes the top element of the data stack, and popping from an empty stack fails (`ErrInvalidStackOperation`); likewise any reference to an element deeper than the stack fails with that error;
- a numeric operand is popped and decoded under Rule 6.11 with maximum length 4;
- operands are popped top first: in `[... x1 x2]` , `x2` is popped first;
- results are pushed after all operands have been popped.

Because any failure invalidates the input, the order in which an opcode checks its operands only affects which error is reported; it is stated where an opcode can either fail or push a result depending on that order.

6.8.1 Push Opcodes

Table 6.4 Push opcodes

Byte(s)	Opcode	Operation
`0x00`	`OP_0`, `OP_FALSE`	Push the empty string.
`0x01` – `0x4b`	`OP_DATA_n`	Push the n inline data bytes.
`0x4c` – `0x4e`	`OP_PUSHDATA1/2/4`	Push the inline data bytes.
`0x4f`	`OP_1NEGATE`	Push the number −1 (`81`).
`0x51` – `0x60`	`OP_1` / `OP_TRUE` … `OP_16`	Push the number b — `0x50` (`01` … `10`).

6.8.2 Flow Control

Table 6.5 Flow-control opcodes

Byte	Opcode	Operation when executed
`0x61`	`OP_NOP`	None.
`0x62`	`OP_VER`	Fail (`ErrReservedOpcode`). Ignored in an unexecuted branch.
`0x63`	`OP_IF`	Rule 6.8.
`0x64`	`OP_NOTIF`	Rule 6.8.
`0x65`	`OP_VERIF`	Fail whenever reached, even in an unexecuted branch (Rule 6.6).
`0x66`	`OP_VERNOTIF`	Fail whenever reached, even in an unexecuted branch.
`0x67`	`OP_ELSE`	Rule 6.8.
`0x68`	`OP_ENDIF`	Rule 6.8.
`0x69`	`OP_VERIFY`	Pop x; if x is false, fail (`ErrVerify`).
`0x6a`	`OP_RETURN`	Fail (`ErrEarlyReturn`). Ignored in an unexecuted branch.

Every opcode named `…VERIFY` other than `OP_VERIFY`, `OP_CHECKLOCKTIMEVERIFY` and `OP_CHECKSEQUENCEVERIFY` behaves as its base opcode followed by `OP_VERIFY`, except that the error on a false result has its own kind (`ErrEqualVerify`, `ErrNumEqualVerify`, `ErrCheckSigVerify`, `ErrCheckMultiSigVerify`, `ErrCheckSigAltVerify`). A `…VERIFY` opcode counts as a single operation.

6.8.3 Stack Manipulation

In Table 6.6, a failure because the stack holds too few elements is always `ErrInvalidStackOperation`.

Table 6.6 Stack-manipulation opcodes

Byte	Opcode	Data stack effect	Notes
`0x6b`	`OP_TOALTSTACK`	`[x] → []`	`x` is pushed onto the alternate stack.
`0x6c`	`OP_FROMALTSTACK`	`[] → [x]`	`x` is popped from the alternate stack; fails if it is empty.
`0x6d`	`OP_2DROP`	`[x1 x2] → []`	
`0x6e`	`OP_2DUP`	`[x1 x2] → [x1 x2 x1 x2]`	
`0x6f`	`OP_3DUP`	`[x1 x2 x3] → [x1 x2 x3 x1 x2 x3]`	
`0x70`	`OP_2OVER`	`[x1 x2 x3 x4] → [x1 x2 x3 x4 x1 x2]`	
`0x71`	`OP_2ROT`	`[x1 … x6] → [x3 x4 x5 x6 x1 x2]`	Requires 6 elements.
`0x72`	`OP_2SWAP`	`[x1 x2 x3 x4] → [x3 x4 x1 x2]`	
`0x73`	`OP_IFDUP`	`[x] → [x x]` or `[x]`	Duplicates `x` only if `x` is true as a boolean.
`0x74`	`OP_DEPTH`	`[] → [n]`	`n` is the number of data stack elements before the push.
`0x75`	`OP_DROP`	`[x] → []`	
`0x76`	`OP_DUP`	`[x] → [x x]`	

Byte	Opcode	Data stack effect	Notes
0x77	OP_NIP	[x1 x2] → [x2]	
0x78	OP_OVER	[x1 x2] → [x1 x2 x1]	
0x79	OP_PICK	[xn … x0 n] → [xn … x0 xn]	See below.
0x7a	OP_ROLL	[xn … x0 n] → [… x0 xn]	See below.
0x7b	OP_ROT	[x1 x2 x3] → [x2 x3 x1]	
0x7c	OP_SWAP	[x1 x2] → [x2 x1]	
0x7d	OP_TUCK	[x1 x2] → [x2 x1 x2]	

RULE 6.12 OP_PICK and OP_ROLL

Pop a numeric operand n (4-byte maximum). Then, on the remaining stack of depth k, n **MUST** satisfy $0 \leq n < k$ (`ErrInvalidStackOperation`). The element at distance n from the top (distance 0 is the top) is copied to the top (`OP_PICK`) or moved to the top, closing the gap (`OP_ROLL`). `OP_ROLL` with $n = 0$ leaves the stack unchanged.

dcrd: txscript/opcode.go · opcodePick
txscript/opcode.go · opcodeRoll
txscript/stack.go · stack.PickN
txscript/stack.go · stack.RollN

6.8.4 Splice Operations

Decred enables the string opcodes that Bitcoin disabled, with its own semantics.

RULE 6.13 OP_CAT

`[... x1 x2]` → `[... x1 || x2]` . Pop `x2` , then `x1` . If both are empty, push the empty string. If exactly one is empty, push the other unchanged. Otherwise $|x_1| + |x_2|$ **MUST NOT** exceed 2048 (`ErrElementTooBig`), and the concatenation `x1 || x2` is pushed.

dcrd: txscript/opcode.go · opcodeCat

RULE 6.14 OP_SUBSTR

`[... s e b]` → `[... s[b:e]]` . Pop the numeric **start index** b (the top element), then the numeric **end index** e, then the string s (all numeric operands 4-byte maximum). If $|s| = 0$, push the empty string regardless of b and e. Otherwise, in this order, **MUST** hold: $b \geq 0$ and $e \geq 0$ (`ErrNegativeSubstrIdx`); $b \leq |s|$, $e \leq |s|$ and $b \leq e$ (`ErrOverflowSubstrIdx`). Push the bytes of s from index b up to but not including e (empty when $b = e$).

dcrd: txscript/opcode.go · opcodeSubstr

RULE 6.15 OP_LEFT and OP_RIGHT

`[... s n]` . Pop the numeric index n, then the string s. If $|s| = 0$, push the empty string regardless of n. Otherwise **MUST** hold: $n \geq 0$ (`ErrNegativeSubstrIdx`) and $n \leq |s|$ (`ErrOverflowSubstrIdx`). `OP_LEFT` pushes the first n bytes `s[0:n]` ; `OP_RIGHT` pushes the bytes from index n to the end, `s[n:]` .

dcrd: txscript/opcode.go · opcodeLeft | txscript/opcode.go · opcodeRight

`OP_SIZE` (`0x82`): `[... x]` → `[... x n]` , where `n` is $|x|$ encoded as a number; `x` is not removed.

IMPLEMENTATION PITFALL

The operand order of `OP_SUBSTR` is the reverse of what its name suggests: the start index is on top, the end index beneath it. `'abcd' 3 1 OP_SUBSTR` yields `'bc'` . The "empty string" shortcuts are taken after the indices have been decoded (so an over-long index still fails with `ErrNumOutOfRange`) but before any bounds check, so `'' 2147483647 2147483646 OP_SUBSTR` succeeds and pushes the empty string.

6.8.5 Bitwise Logic, Comparison and Rotation

The bitwise opcodes operate on numbers, not on byte strings. Each operand is decoded as a 4-byte script number and then treated as a 32-bit two's complement integer; the 32-bit result is interpreted as a signed integer and pushed with `encode_num`.

Table 6.7 Bitwise and byte-comparison opcodes

Byte	Opcode	Operation (`[... x1 x2]`, `x2` on top)
`0x83`	`OP_INVERT`	`[x] → [~x]`: bitwise complement as `int32`, equal to $-x-1$. `OP_INVERT` of $2^{31}-1$ yields -2^{31}, pushed as 5 bytes.
`0x84`	`OP_AND`	`int32(x1) & int32(x2)`.
`0x85`	`OP_OR`	`int32(x1) | int32(x2)`.
`0x86`	`OP_XOR`	`int32(x1) ^ int32(x2)`.
`0x87`	`OP_EQUAL`	Pop two elements; push true if they are identical byte strings (same length and bytes), else false. No numeric interpretation: `00` and the empty string differ.
`0x88`	`OP_EQUALVERIFY`	`OP_EQUAL` then verify (`ErrEqualVerify`).
`0x89`	`OP_ROTR`	Rule 6.16.
`0x8a`	`OP_ROTL`	Rule 6.16.

RULE 6.16 OP_ROTR and OP_ROTL

`[... v c]`. Pop the numeric **count** c, then the numeric **value** v (both 4-byte maximum). $c \geq 0$ **MUST** hold (`ErrNegativeRotation`) and $c \leq 31$ **MUST** hold (`ErrOverflowRotation`). Let $u =$ `uint32(v)`. The result is u rotated right (`OP_ROTR`) or left (`OP_ROTL`) by c bit positions within 32 bits, reinterpreted as `int32`. A rotation by 0 returns v unchanged.

dcrd: txscript/opcode.go · opcodeRotr | txscript/opcode.go · opcodeRotl

IMPLEMENTATION PITFALL

Because results are reinterpreted as signed and operands are limited to 4 bytes, a rotation or shift whose result has bit 31 set produces a negative number; if that number is -2^{31} its encoding is 5 bytes and it cannot be used as an operand again. For example `1 1 OP_ROTR` pushes -2^{31} (`00 00 00 80 80`), and feeding that to a second `OP_ROTR` fails with `ErrNumOutOfRange`, whereas `1 2 OP_ROTR` pushes 2^{30}, which remains a valid operand.

6.8.6 Arithmetic

All operands in Table 6.8 are 4-byte numeric operands. In the binary operations, a is the second element from the top (`x1`) and b the top element (`x2`), so `a b OP_SUB` computes $a-b$. Results marked "exact" are computed without overflow (they lie within $[-(2^{32}-2), 2^{32}-2]$ and may encode to 5 bytes). Booleans produced by numeric opcodes are the numbers 1 and 0, that is, `01` and the empty string.

Table 6.8 Arithmetic opcodes

Byte	Opcode	Result
`0x8b`	`OP_1ADD`	$a+1$, exact.
`0x8c`	`OP_1SUB`	$a-1$, exact.
`0x8d`	`OP_2MUL`	**No operation** (behaves as `OP_NOP`; nothing is popped or checked).
`0x8e`	`OP_2DIV`	**No operation** (behaves as `OP_NOP`).
`0x8f`	`OP_NEGATE`	$-a$.
`0x90`	`OP_ABS`	$\lvert a \rvert$.
`0x91`	`OP_NOT`	1 if $a=0$, else 0.
`0x92`	`OP_0NOTEQUAL`	0 if $a=0$, else 1.
`0x93`	`OP_ADD`	$a+b$, exact.

Byte	Opcode	Result
0x94	OP_SUB	$a - b$, exact.
0x95	OP_MUL	int32(a * b): the product reduced modulo 2^{32} and reinterpreted as signed (wrapping).
0x96	OP_DIV	tdiv(a, b) (truncating toward zero). $b = 0$ fails (ErrDivideByZero).
0x97	OP_MOD	tmod(a, b) (remainder of truncating division, sign of a). $b = 0$ fails (ErrDivideByZero).
0x98	OP_LSHIFT	Rule 6.17.
0x99	OP_RSHIFT	Rule 6.17.
0x9a	OP_BOOLAND	1 if $a \neq 0$ and $b \neq 0$, else 0.
0x9b	OP_BOOLOR	1 if $a \neq 0$ or $b \neq 0$, else 0.
0x9c	OP_NUMEQUAL	1 if $a = b$, else 0.
0x9d	OP_NUMEQUALVERIFY	OP_NUMEQUAL then verify (ErrNumEqualVerify).
0x9e	OP_NUMNOTEQUAL	1 if $a \neq b$, else 0.
0x9f	OP_LESSTHAN	1 if $a < b$, else 0.
0xa0	OP_GREATERTHAN	1 if $a > b$, else 0.
0xa1	OP_LESSTHANOREQUAL	1 if $a \leq b$, else 0.
0xa2	OP_GREATERTHANOREQUAL	1 if $a \geq b$, else 0.
0xa3	OP_MIN	$\min(a, b)$.
0xa4	OP_MAX	$\max(a, b)$.
0xa5	OP_WITHIN	[... x lo hi]: pop hi, lo, x; 1 if lo $\leq x <$ hi, else 0.

RULE 6.17 OP_LSHIFT and OP_RSHIFT

[... v c]. Pop the numeric **count** c, then the numeric **value** v (both 4-byte maximum). $c \geq 0$ **MUST** hold (ErrNegativeShift) and $c \leq 32$ **MUST** hold (ErrOverflowShift). With v as a 32-bit two's complement integer:

- OP_LSHIFT pushes int32(v << c), discarding bits shifted out of the 32-bit width; for $c = 32$ the result is 0;
- OP_RSHIFT pushes the arithmetic (sign-replicating) shift v >> c; for $c = 32$ the result is 0 if $v \geq 0$ and -1 if $v < 0$.

dcrd: txscript/opcode.go · opcodeLShift | txscript/opcode.go · opcodeRShift

IMPLEMENTATION PITFALL

Shifts by 32 are defined (Go semantics). In C and C++, shifting a 32-bit value by 32 is undefined; in Rust it panics in debug builds and silently masks the count in release builds, so that x << 32 yields x. Implementations in those languages must special-case it. The same applies to the 32 - c shift inside a rotation by 0.

NOTE

Overflow cannot occur in OP_DIV or OP_MOD: the only overflowing case of 32-bit truncating division, $-2^{31} \div -1$, needs the operand -2^{31}, which is not a valid 4-byte operand. OP_NOT and OP_0NOTEQUAL decode their operand numerically, so 'abcdef' OP_NOT fails with ErrNumOutOfRange rather than treating the string as a boolean.

6.8.7 Hash Opcodes

Each hash opcode pops one element, hashes its bytes, and pushes the digest. The functions are defined in Chapter 3.

Table 6.9 Hash opcodes

Byte	Opcode	Digest	Function
0xa6	OP_RIPEMD160	20	RIPEMD-160(x).

Byte	Opcode	Digest	Function
0xa7	OP_SHA1	20	SHA-1(x).
0xa8	OP_BLAKE256	32	BLAKE-256(x) (14 rounds, as used throughout Decred).
0xa9	OP_HASH160	20	RIPEMD-160(BLAKE-256(x)). **Not** SHA-256 as in Bitcoin.
0xaa	OP_HASH256	32	BLAKE-256(BLAKE-256(x)). **Not** double SHA-256.
0xc0	OP_SHA256	32	SHA-256(x), only when ScriptVerifySHA256 is set (Rule 6.18).

RULE 6.18 OP_SHA256

If the flag ScriptVerifySHA256 is set, executing byte 0xc0 pops x and pushes SHA-256(x). If the flag is not set, it behaves as an upgradable no-operation (Rule 6.21), named OP_UNKNOWN192 .

dcrd: txscript/opcode.go · opcodeSha256

HISTORY

OP_SHA256 was introduced by DCP-0002 [29] to support hash time-locked contracts that share a SHA-256 hash lock with other chains. It is enabled, together with OP_CHECKSEQUENCEVERIFY , by the lnfeatures agenda: the flag is set for a block exactly when that agenda is active for the block (Chapter 7 specifies the flag selection, Chapter 13 the activation). Before activation, byte 0xc0 is a no-operation. The meaning of the byte therefore depends on the flag of the block that contains the spend, and an implementation must select the flag per block. Unlike the lock-time opcodes, which only make a no-operation stricter, this change is not a pure tightening: <x> 0xc0 <SHA-256(x)> OP_EQUAL is invalid before activation and valid after it, and <h> 0xc0 <h> OP_EQUAL the reverse.

6.8.8 Lock-Time Opcodes

These opcodes compare a stack value with fields of the spending transaction. Neither removes its operand from the stack; scripts conventionally follow them with OP_DROP . Both operands are decoded with a 5-byte maximum so that values up to $2^{32} - 1$ (and beyond) can be expressed.

RULE 6.19 OP_CHECKLOCKTIMEVERIFY

If the flag ScriptVerifyCheckLockTimeVerify is not set, byte 0xb1 behaves as an upgradable no-operation (named OP_NOP2) (Rule 6.21). Otherwise, let L be the top element decoded as a number with maximum length 5, without popping it (an empty stack fails with ErrInvalidStackOperation), let T be the transaction's lock time and q the sequence number of the input being verified. The following **MUST** hold:

1. $L \geq 0$ (ErrNegativeLockTime);
2. L and T are of the same kind: both < 500000000 (block heights) or both ≥ 500000000 (Unix times) (ErrUnsatisfiedLockTime);
3. $L \leq T$ (ErrUnsatisfiedLockTime);
4. $q \neq$ 0xffffffff (ErrUnsatisfiedLockTime).

dcrd: txscript/opcode.go · opcodeCheckLockTimeVerify | txscript/opcode.go · verifyLockTime

The opcode thus proves that the transaction's own lock time is at least L, and the last condition ensures that the lock time is actually enforced for this transaction (a transaction whose inputs all have the maximum sequence number ignores its lock time; see Chapter 7).

RULE 6.20 OP_CHECKSEQUENCEVERIFY

If the flag ScriptVerifyCheckSequenceVerify is not set, byte 0xb2 behaves as an upgradable no-operation (named OP_NOP3) (Rule 6.21). Otherwise, let s be the top element decoded as a number with maximum length 5, without popping it, let V be the transaction version and q the sequence number of the input being verified. Then:

1. $s \geq 0$ **MUST** hold (ErrNegativeLockTime);

2. if bit 31 of s is set (s AND `0x80000000` $\neq 0$), the opcode succeeds with no further checks;
3. $V \geq 2$ **MUST** hold (`ErrUnsatisfiedLockTime`);
4. bit 31 of q **MUST** be clear (`ErrUnsatisfiedLockTime`);
5. let m = `0x0040ffff` (the type flag, bit 22, together with the low 16 bits), $s' = s$ AND m and $q' = q$ AND m. s' and q' **MUST** be of the same kind: both $<$ `0x00400000` or both $\geq$ `0x00400000` (`ErrUnsatisfiedLockTime`);
6. $s' \leq q'$ **MUST** hold (`ErrUnsatisfiedLockTime`).

dcrd: txscript/opcode.go · opcodeCheckSequenceVerify

NOTE
Bits 16 to 21 and 23 to 30 of both values, and bits 32 to 38 of the 5-byte operand, are ignored. The meaning of the input sequence number as a relative lock time (blocks if bit 22 is clear, units of 512 seconds if set) and its enforcement are specified in Chapter 7; this opcode only compares the script's requirement with the value the transaction commits to.

IMPLEMENTATION PITFALL
Both lock-time operands are subject to the minimal-encoding rule of Rule 6.11. The test vector table of DCP-0003 lists the public key script `05 00 00 00 00 00 b2 51` (a 5-byte non-minimal encoding of zero followed by `OP_CHECKSEQUENCEVERIFY OP_1`) as valid; dcrd rejects it with `ErrMinimalData` . (The non-minimal lock-time operands in dcrd's `txscript/testdata/tx_invalid.json` are 6 bytes long and fail the length check, `ErrNumOutOfRange` , before minimality is considered.) The reference implementation is authoritative.

HISTORY
`OP_CHECKLOCKTIMEVERIFY` is enforced on every block since the genesis block (the flag is unconditionally part of the consensus flags). `OP_CHECKSEQUENCEVERIFY` was introduced by DCP-0003 [30] and is enforced once the `lnfeatures` agenda is active; before that byte `0xb2` is a no-operation.

6.8.9 No-Operation and Unassigned Opcodes

RULE 6.21 No-operation opcodes
When executed:

- `OP_NOP` (`0x61`), `OP_2MUL` (`0x8d`), `OP_2DIV` (`0x8e`) and the stake tag opcodes `0xba` – `0xbd` have no effect under any flags;
- the **upgradable** no-operations `OP_NOP1` (`0xb0`), `OP_NOP4` – `OP_NOP10` (`0xb3` – `0xb9`) and `OP_UNKNOWN196` – `OP_UNKNOWN248` (`0xc4` – `0xf8`), and the flag-gated opcodes `0xb1` , `0xb2` , `0xc0` , `0xc1` , `0xc2` , `0xc3` when their enabling flag is not set, have no effect unless the flag `ScriptDiscourageUpgradableNops` is set, in which case they fail (`ErrDiscourageUpgradableNOPs`).

All of these count toward the operation limit when reached.

dcrd: txscript/opcode.go · opcodeNop

RULE 6.22 Reserved and invalid opcodes
Executing `OP_RESERVED` (`0x50`), `OP_VER` (`0x62`), or any of `0xf9` – `0xff` fails (`ErrReservedOpcode`). In an unexecuted branch they have no effect. `0x50` does not count toward the operation limit; the others do.

dcrd: txscript/opcode.go · opcodeReserved | txscript/opcode.go · opcodeInvalid

`ScriptDiscourageUpgradableNops` is a policy flag and is never set in consensus validation; consensus therefore treats every upgradable no-operation as a plain no-operation. Note also that `OP_2MUL` and `OP_2DIV` are no-operations that are *not* discouraged even under the policy flag.

6.8.10 Stake and Treasury Tag Opcodes

Bytes `0xba` – `0xbd` (`OP_SSTX`, `OP_SSGEN`, `OP_SSRTX`, `OP_SSTXCHANGE`) and `0xc1` – `0xc3` (`OP_TADD`, `OP_TSPEND`, `OP_TGEN`) are **tags**: when executed they do nothing. Their meaning is structural and is enforced outside the engine (Section 6.12). The stake tags `0xba` – `0xbd` are no-operations under all flags. The treasury tags are no-operations when `ScriptVerifyTreasury` is set; when it is not set they are the upgradable no-operations `OP_UNKNOWN193` – `OP_UNKNOWN195` (Rule 6.21).

6.9 Signature Verification

Six opcodes verify signatures: `OP_CHECKSIG` and `OP_CHECKSIGVERIFY` (ECDSA over secp256k1, one key), `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` (ECDSA, m of n keys), and `OP_CHECKSIGALT` and `OP_CHECKSIGALTVERIFY` (alternative suites: Ed25519 and EC-Schnorr-DCRv0). In every case the message signed is the 32-byte **signature hash** of Section 6.10, computed from the transaction, the input index, the hash type and a **subscript** derived from the current script.

A **signature element** on the stack is the encoded signature followed by one **hash type** byte. The engine distinguishes two kinds of failure. An **encoding** failure (an invalid hash type, a non-canonical DER signature or a disallowed public key format) makes the script fail immediately. A **verification** failure (a signature that is well encoded but does not verify, a public key that is not a valid curve point, and similar) makes the opcode push false; the script fails only if that false value is later verified or left as the result. The difference is observable (for example with `OP_CHECKSIG OP_NOT`) and is consensus.

6.9.1 Encoding Checks

RULE 6.23 Hash type encoding

A hash type byte t is valid if and only if t AND `0x7f` is 1, 2 or 3; that is, t is one of `0x01`, `0x02`, `0x03`, `0x81`, `0x82`, `0x83`. An invalid hash type is an encoding failure (`ErrInvalidSigHashType`).

dcrd: txscript/consensus.go · CheckHashTypeEncoding

RULE 6.24 Strict DER signature encoding

An ECDSA signature g (the signature element without its hash type byte) **MUST** satisfy all of the following; any violation is an encoding failure with the error kind shown. Let $L = |g|$.

1. $L \geq 8$ (`ErrSigTooShort`) and $L \leq 72$ (`ErrSigTooLong`).
2. $g[0] =$ `0x30` (`ErrSigInvalidSeqID`).
3. $g[1] = L - 2$ (`ErrSigInvalidDataLen`).
4. Let $r_{\text{len}} = g[3]$ and $s_{\text{type}} = 4 + r_{\text{len}}$. $s_{\text{type}} < L$ (`ErrSigMissingSTypeID`) and $s_{\text{type}} + 1 < L$ (`ErrSigMissingSLen`).
5. Let $s_{\text{len}} = g\left[s_{\text{type}} + 1\right]$. $s_{\text{type}} + 2 + s_{\text{len}} = L$ (`ErrSigInvalidSLen`).
6. $g[2] =$ `0x02` (`ErrSigInvalidRIntID`); $r_{\text{len}} \neq 0$ (`ErrSigZeroRLen`); $g[4]$ AND `0x80` $= 0$ (`ErrSigNegativeR`); not ($r_{\text{len}} > 1$ and $g[4] =$ `0x00` and $g[5]$ AND `0x80` $= 0$) (`ErrSigTooMuchRPadding`).
7. $g\left[s_{\text{type}}\right] =$ `0x02` (`ErrSigInvalidSIntID`); $s_{\text{len}} \neq 0$ (`ErrSigZeroSLen`); the first byte of S has its top bit clear (`ErrSigNegativeS`); not ($s_{\text{len}} > 1$ and S's first byte is `0x00` and its second byte has the top bit clear) (`ErrSigTooMuchSPadding`).
8. With leading zero bytes removed, S is at most 32 bytes, and as a big-endian integer $S < n$ and $S \leq \frac{n}{2}$ (rounded down), where n is the order of the secp256k1 group (`ErrSigHighS`).

The bytes of R are $g[4 : 4 + r_{\text{len}}]$ and those of S are $g\left[s_{\text{type}} + 2 : L\right]$.

dcrd: txscript/consensus.go · CheckSignatureEncoding

These checks do not ensure that R is non-zero or less than n, nor that S is non-zero; those conditions are checked when the signature is parsed (Chapter 3), and a failure there is a verification failure, not an encoding failure.

RULE 6.25 Public key encoding

A public key for `OP_CHECKSIG` or `OP_CHECKMULTISIG` **MUST** be either 33 bytes beginning with `0x02` or `0x03` (compressed) or 65 bytes beginning with `0x04` (uncompressed). Anything else, including the 65-byte "hybrid" formats beginning with `0x06` or `0x07`, is an encoding failure (`ErrPubKeyType`). Whether the bytes denote a point on the curve is checked later, and a failure there is a verification failure.

dcrd: txscript/consensus.go · CheckPubKeyEncoding | txscript/engine.go · isStrictPubKeyEncoding

6.9.2 The Subscript

The signature hash commits to the script containing the signature-checking opcode, called the **current script**: the public key script for an ordinary spend, the redeem script for a P2SH spend, or the signature script itself if the signature opcode appears there. Because `OP_CODESEPARATOR` is disabled (executing or passing over it fails), the subscript always starts at the beginning of the current script.

Before hashing, every push in the current script whose data **contains** a signature element being checked is removed. This exists for historical reasons and rarely matters, but it is consensus.

RULE 6.26 Subscript construction

The subscript for a signature element e is the current script with every opcode removed that satisfies both:

- it is **canonical** in the sense of Algorithm 6.4 (this differs from the minimal-push rule), and
- its data contains e as a contiguous substring.

The opcodes that remain are concatenated in their original encoding. If e is empty, nothing is removed. For `OP_CHECKMULTISIG` the removal is applied successively for every signature element popped, in the order they were popped (topmost first), before any signature is checked.

dcrd: txscript/opcode.go · opcodeCheckSig
txscript/script.go · removeOpcodeByData
txscript/script.go · isCanonicalPush

ALGORITHM 6.4 Removing signature pushes from the subscript

```
def is_canonical_for_removal(op, data) -> bool:
    if op > 0x60:                              # not a push-class opcode
        return True
    if 0x00 < op < 0x4c and len(data) == 1 and data[0] <= 16:
        return False                           # note: includes data == 00
    if op == 0x4c and len(data) < 0x4c:   return False
    if op == 0x4d and len(data) <= 0xff:  return False
    if op == 0x4e and len(data) <= 0xffff: return False
    return True

def remove_sig_pushes(script: bytes, e: bytes) -> bytes:
    if len(script) == 0 or len(e) == 0:
        return script
    out = b""
    for op, data, start, end in parse_with_offsets(script):
        if is_canonical_for_removal(op, data) and contains(data, e):
            continue                           # drop this opcode
        out += script[start:end]               # keep its original bytes
    return out
```

dcrd: txscript/script.go · removeOpcodeByData

Opcodes without data never match (a non-empty e is not a substring of the empty string). Note that `OP_DATA_1 00` counts as **non-canonical** here (although it is a minimal push under Rule 6.7), whereas `OP_DATA_1 81` counts as canonical.

6.9.3 OP_CHECKSIG

RULE 6.27 OP_CHECKSIG

`[... sig pk] → [... bool]`. In this order:

1. Pop p (the public key), then e (the signature element).
2. If $|e| = 0$, push false and stop. (The public key is not checked.)
3. Let t be the last byte of e and g the preceding bytes. t **MUST** pass Rule 6.23, g **MUST** pass Rule 6.24 and p **MUST** pass Rule 6.25; each failure is an encoding failure.
4. Build the subscript by Rule 6.26 for e and compute the signature hash h (Algorithm 6.6) with hash type t. If the hash cannot be computed (the `SIGHASH_SINGLE` case of Rule 6.30), push false and stop.
5. Parse p as a secp256k1 point and g as a DER signature (Chapter 3); if either fails, push false and stop.
6. Push true if the ECDSA signature g is valid for h under p, else false.

dcrd: txscript/opcode.go · opcodeCheckSig

`OP_CHECKSIGVERIFY` performs `OP_CHECKSIG` followed by verification of the result (`ErrCheckSigVerify`).

6.9.4 OP_CHECKMULTISIG

Decred's multisignature opcode does **not** pop an extra "dummy" element. A signature script that supplies Bitcoin's customary leading `OP_0` leaves that element on the stack, which fails the clean-stack rule in consensus validation.

RULE 6.28 OP_CHECKMULTISIG

`[... sig_1 … sig_m m pk_1 … pk_n n] → [... bool]`. In this order:

1. Pop the numeric operand n (4-byte maximum). $0 \le n \le 20$ **MUST** hold (`ErrInvalidPubKeyCount`).
2. Add n to the per-script operation count; the count **MUST NOT** exceed 255 (`ErrTooManyOperations`).
3. Pop n public keys. Call them $K_0, \ldots, K_{n-1}$ in popping order (K_0 was on top, so $K_0 =$ `pk_n`).
4. Pop the numeric operand m (4-byte maximum). $0 \le m \le n$ **MUST** hold (`ErrInvalidSignatureCount`).
5. Pop m signature elements $E_0, \ldots, E_{m-1}$ in popping order ($E_0 =$ `sig_m`).
6. Compute the subscript by applying Rule 6.26 for E_0, then E_1, and so on.
7. Match signatures to keys with Algorithm 6.5 and push its result.

dcrd: txscript/opcode.go · opcodeCheckMultiSig

ALGORITHM 6.5 OP_CHECKMULTISIG matching

```
def multisig_match(E, K, subscript, tx, idx) -> bool:
    parsed = [None] * len(E)          # None: not yet parsed; INVALID; or a signature
    keys_left, sigs_left = len(K) + 1, len(E)
    ki, si = -1, 0
    while sigs_left > 0:
        ki += 1; keys_left -= 1
        if sigs_left > keys_left:
            return False                          # cannot succeed any more
        e, k = E[si], K[ki]
        if len(e) == 0:
            continue                              # empty signature: next key
        t, g = e[-1], e[:-1]
        if parsed[si] is None:
            check_hash_type(t)                    # encoding failure: script fails
```

```
            check_der(g)                          # encoding failure: script fails
            parsed[si] = parse_der(g) or INVALID
            if parsed[si] is INVALID:
                continue
        elif parsed[si] is INVALID:
            continue
        check_pubkey_encoding(k)                  # encoding failure: script fails
        point = parse_pubkey(k)
        if point is None:
            continue
        h = signature_hash(subscript, t, tx, idx)     # SIGHASH_SINGLE out of range:
                                                      # the script FAILS here
        if ecdsa_verify(parsed[si], h, point):
            si += 1; sigs_left -= 1
    return True
```

dcrd: txscript/opcode.go · opcodeCheckMultiSig

Consequences that must be reproduced exactly:

- Signatures must appear in the same relative order as the keys they match. Matching starts from the topmost signature and topmost key.
- A key that fails to match the current signature is skipped and never reconsidered. Once fewer keys remain than signatures, the result is false without examining the rest.
- Encoding checks are lazy: a signature's hash type and DER encoding are checked only when that signature is first compared, and a key's encoding only when it is compared against a non-empty, parseable signature. Elements never reached are never checked, so a badly encoded key or signature beyond the point of early exit does not cause failure.
- An empty signature element matches no key; with $m \geq 1$ empty elements, the result is false after the keys are exhausted.
- With $m = 0$ the result is true and no key is examined; `0 0 OP_CHECKMULTISIG` pushes true.
- Unlike `OP_CHECKSIG`, an out-of-range `SIGHASH_SINGLE` makes the script **fail** rather than push false.
- The key count n is added to the operation count whenever the opcode is executed, regardless of how many keys are examined.

`OP_CHECKMULTISIGVERIFY` performs `OP_CHECKMULTISIG` followed by verification (`ErrCheckMultiSigVerify`).

6.9.5 OP_CHECKSIGALT

`OP_CHECKSIGALT` verifies a signature under one of several alternative signature suites, selected by a small integer on top of the stack. Table 6.10 lists the suites.

Table 6.10 Signature types for `OP_CHECKSIGALT`

Type	Suite	Public key	Element	Behaviour
0	(ECDSA, not allowed here)	—	—	Pushes false; nothing else popped.
1	Ed25519	32 bytes	65 bytes	Verified as below.
2	EC-Schnorr-DCRv0 (secp256k1)	33 bytes	65 bytes	Verified as below.
other	unassigned	—	—	Pushes **true**; nothing else popped.

RULE 6.29 OP_CHECKSIGALT

`[... sig pk type] → [... bool]`. In this order:

1. Pop the numeric operand y with maximum length **1** (so $-127 \leq y \leq 127$; a longer element fails with `ErrNumOutOfRange`).
2. If $y = 0$, push false and stop. If y is neither 1 nor 2, push true and stop. In both cases the public key and signature are not popped.

3. Pop p. If $|p| \neq 32$ (for $y = 1$) or $|p| \neq 33$ (for $y = 2$), push false and stop; the signature element is **not** popped.
4. Pop e. If $|e| \neq 65$, push false and stop.
5. Let $t = e[64]$ and $g = e[0:64]$. t **MUST** pass Rule 6.23 (encoding failure). No DER or public key format check applies.
6. Build the subscript by Rule 6.26 for e and compute the signature hash h with hash type t. If it cannot be computed, push false and stop.
7. For $y = 1$: parse p as an Ed25519 public key and g as an Ed25519 signature; for $y = 2$: parse p as a compressed secp256k1 public key (first byte `0x02` or `0x03`) and g as an EC-Schnorr-DCRv0 signature (r, s) with $r < p_{\text{field}}$ and $s < n$ (Chapter 3). If either parse fails, push false; otherwise push the result of verifying g over h under p.

dcrd: txscript/opcode.go · opcodeCheckSigAlt

`OP_CHECKSIGALTVERIFY` performs `OP_CHECKSIGALT` followed by verification (`ErrCheckSigAltVerify`).

IMPLEMENTATION PITFALL
The early exits of `OP_CHECKSIGALT` leave elements on the stack. For an unassigned type such as 3 the opcode consumes only the type and pushes true, so `3 OP_CHECKSIGALT` succeeds on an otherwise empty stack. With a wrong-length public key the signature element stays beneath the pushed false. Under the clean-stack rule these leftovers usually cause failure, but an implementation that pops all three operands up front produces different results for scripts such as `<sig> <short key> 1 OP_CHECKSIGALT OP_NOT OP_DROP`.

RATIONALE
Treating unknown signature types as success was intended to allow new suites to be introduced as soft forks, by making a previously always-succeeding opcode stricter. Type 0 is the identifier of ECDSA over secp256k1 (`STEcdsaSecp256k1`), which has its own opcodes; through `OP_CHECKSIGALT` it always yields false.

Table 6.11 summarizes which conditions fail the script and which push false.

Table 6.11 Failure modes of the signature opcodes

Condition	`CHECKSIG`	`CHECKMULTISIG`	`CHECKSIGALT`
Too few stack elements	fail	fail	fail[a]
Empty signature element	false	skip key	false (wrong length)
Invalid hash type	fail	fail (if reached)	fail
DER encoding violation	fail	fail (if reached)	n/a
Public key format violation	fail	fail (if reached)	false (length only)
Key not a valid point / signature unparseable	false	skip	false
`SIGHASH_SINGLE` index out of range	false	fail	false
Signature does not verify	false	skip key	false

[a] Only for elements actually popped: signature type 0 and unknown types return after popping the type alone, and a public key of the wrong length yields false before the signature is popped, leaving the signature on the stack (Rule 6.29).

6.10 Signature Hash

The signature hash is the 32-byte message that signatures commit to. Decred computes it from two intermediate hashes, mirroring its split of a transaction into a **prefix** (inputs' outpoints and sequences, outputs, lock time, expiry) and a **witness** (per-input data, Chapter 4). The prefix hash commits to the parts of the prefix selected by the hash type; the witness hash commits to the subscript, placed at the position of the input being signed. The final hash commits to the hash type and both intermediate hashes. All three hashes use BLAKE-256.

Table 6.12 Signature hash types

Value	Name	Commits to
`0x01`	`SIGHASH_ALL`	All inputs and all outputs.
`0x02`	`SIGHASH_NONE`	All inputs (other inputs' sequence numbers replaced by 0) and no outputs.
`0x03`	`SIGHASH_SINGLE`	All inputs (other inputs' sequence numbers replaced by 0) and the output at the same index as the input; earlier outputs are committed in blanked form, later outputs not at all.
`0x80`	`SIGHASH_ANYONECANPAY`	Flag combined with one of the above: only the input being signed is committed, in both the prefix and the witness.

Two constants select the serialization variant and are mixed into the version field of each intermediate serialization: `SigHashSerializePrefix` $= 1$ and `SigHashSerializeWitness` $= 3$.

RULE 6.30 SIGHASH_SINGLE with no matching output

If the hash type AND `0x1f` equals 3 and the input index is greater than or equal to the number of transaction outputs, the signature hash is undefined (`ErrInvalidSigHashSingleIndex`). The consequence is opcode-specific: `OP_CHECKSIG` and `OP_CHECKSIGALT` push false, and `OP_CHECKMULTISIG` fails the script (Table 6.11).

dcrd: txscript/sighash.go · calcSignatureHash

ALGORITHM 6.6 Signature hash

```
ALL, NONE, SINGLE, ANYONECANPAY = 0x01, 0x02, 0x03, 0x80
SER_PREFIX, SER_WITNESS = 1, 3

def signature_hash(sub: bytes, t: int, tx, idx: int) -> bytes:
    base = t & 0x1f
    if base == SINGLE and idx >= len(tx.outputs):
        fail("ErrInvalidSigHashSingleIndex")
    if t & ANYONECANPAY:
        ins, sign_in = [tx.inputs[idx]], 0
    else:
        ins, sign_in = tx.inputs, idx
    if base == NONE:
        outs = []
    elif base == SINGLE:
        outs = tx.outputs[: idx + 1]
    else:                                   # ALL (and any other value)
        outs = tx.outputs

    p = le32(tx.version | (SER_PREFIX << 16)) + varint(len(ins))
    for j, ti in enumerate(ins):
        seq = ti.sequence
        if base in (NONE, SINGLE) and j != sign_in:
            seq = 0
        p += ti.prev_hash + le32(ti.prev_index) + u8(ti.prev_tree) + le32(seq)
    p += varint(len(outs))
    for j, to in enumerate(outs):
        value, script = to.value, to.pk_script
        if base == SINGLE and j != idx:
            value, script = -1, b""          # version is kept
        p += le64(uint64(value)) + le16(to.version) + varint(len(script)) + script
    p += le32(tx.lock_time) + le32(tx.expiry)

    w = le32(tx.version | (SER_WITNESS << 16)) + varint(len(ins))
    for j in range(len(ins)):
        s = sub if j == sign_in else b""
        w += varint(len(s)) + s

    return BLAKE256(le32(t) + BLAKE256(p) + BLAKE256(w))
```

dcrd: txscript/sighash.go · calcSignatureHash | txscript/sighash.go · CalcSignatureHash

Notes on Algorithm 6.6:

- `tx.version` is the 16-bit transaction version; the low 16 bits of the 32-bit field are the version and the high 16 bits the serialization type. `varint` is the variable-length integer of Chapter 4. `prev_hash` is the 32-byte previous transaction hash in its internal (non-reversed) byte order, and `prev_tree` is the one-byte tree indicator of the outpoint.
- The final serialization is 68 bytes: the full hash type byte zero-extended to 32 bits and written little-endian, followed by the two 32-byte hashes.
- For `SIGHASH_ALL` without `SIGHASH_ANYONECANPAY`, the prefix serialization is byte-for-byte the transaction prefix serialization of Chapter 4, so the prefix hash equals the transaction hash. This is not true of the witness part: the signature-hash witness contains only the subscript for the signed input and an empty script for every other input, and none of the value, block height, block index or signature script fields of the wire witness serialization.
- In the `SIGHASH_SINGLE` case, every output before the signed index is committed with amount -1 (eight `0xff` bytes), its original script version, and an empty script; outputs after the index are omitted. With `SIGHASH_ANYONECANPAY`, the comparison `j != idx` still uses the original input index, since output positions are not renumbered.
- Within the engine the hash type has already passed Rule 6.23, so `base` is 1, 2 or 3. The mask `0x1f` and the treatment of other values as `SIGHASH_ALL` matter only to external callers of the function (dcrd's test vectors exercise them).
- `OP_CODESEPARATOR` has no role: it cannot appear in a script that executes. The subscript `sub` is the whole current script minus the pushes removed by Rule 6.26.

The signature hash vectors in Section 6.17 show the intermediate serializations for each hash type.

6.11 Pay-to-Script-Hash

Pay-to-script-hash [31] lets an output commit to the hash of an arbitrary **redeem script** rather than to the script itself. The spender reveals the redeem script as the last push of the signature script, together with the data that satisfies it. The engine first checks that the revealed script hashes to the committed value, then executes the redeem script. In Decred the hash is `OP_HASH160`, that is RIPEMD-160(BLAKE-256(·)).

Pay-to-script-hash is recognized purely by the byte pattern of the public key script (Section 6.5), regardless of the script version and of any flags other than `ScriptVerifyTreasury`, which widens the set of stake tags. There is no activation height: P2SH has been enforced since the genesis block.

Table 6.13 Pay-to-script-hash patterns

Form	Length	Bytes
P2SH	23	`a9 14` ‖ 20-byte hash ‖ `87`
Stake-tagged P2SH	24	tag ‖ `a9 14` ‖ 20-byte hash ‖ `87`, tag ∈ {`ba`, `bb`, `bc`, `bd`}, or also ∈ {`c1`, `c2`, `c3`} when `ScriptVerifyTreasury` is set

The 23-byte form is recognized only when the whole public key script is exactly that pattern; the bytes `a9 14 … 87` embedded in a longer script do not make it P2SH. A stake-tagged P2SH script executes its tag as a no-operation, then the ordinary hash comparison.

RULE 6.31 Pay-to-script-hash evaluation

If the public key script is P2SH-shaped:

1. the signature script **MUST** be push-only, and for version 0 the redeem-script conditions of Rule 6.3 step 4 **MUST** hold;

2. after the signature script, the data stack is saved, and the public key script is executed on it as usual (it compares the hash of the top element with the committed hash);
3. after the public key script, the data stack **MUST** be non-empty (`ErrEmptyStack`) and its top element **MUST** be true (`ErrEvalFalse`). No clean-stack check applies at this point;
4. the redeem script R is the top element of the saved stack. R **MUST** parse (`ErrMalformedPush`). The data stack is replaced by the saved stack with R removed, and R is executed as a third script, with its own operation count, alternate stack and conditional state;
5. the final result is then determined by Rule 6.9.

dcrd: txscript/engine.go · Engine.Step | txscript/engine.go · NewEngine

RULE 6.32 No stake opcodes in redeem scripts

For a version 0 spend of a P2SH-shaped output, the redeem script **MUST NOT** contain any of the opcodes `0xba` – `0xbd` , and, when `ScriptVerifyTreasury` is set, `0xc1` – `0xc3` (`ErrP2SHStakeOpCodes`). The redeem script is scanned opcode by opcode from the start; if a parse failure is met before any such opcode, the spend also fails.

dcrd: txscript/script.go · hasP2SHRedeemScriptStakeOpCodes

RATIONALE

Stake and treasury tags are meaningful only in the precise positions the stake system checks. Hiding one inside a redeem script, where no structural rule can see it, is prohibited outright so that tagged scripts cannot be smuggled into regular transactions.

NOTE

Redeem scripts are not subject to the 16384-byte script size limit, but as stack elements they are limited to 2048 bytes by Rule 6.6. Like every script, a redeem script has its own limit of 255 operations.

HISTORY

Before the `treasury` agenda (DCP-0006) is active, the flag `ScriptVerifyTreasury` is not set, so `0xc1` – `0xc3` are neither stake tags for P2SH detection nor prohibited in redeem scripts. A script such as `c1 a9 14 … 87` is then executed as an ordinary script whose first opcode is a no-operation (`OP_UNKNOWN193`), not as P2SH; dcrd's test data includes both cases.

6.12 Stake and Treasury Tags

Decred's staking and treasury transactions place tag opcodes at the start of specific output scripts. A ticket's voting-rights output begins with `OP_SSTX` , its change outputs with `OP_SSTXCHANGE` , vote outputs with `OP_SSGEN` , revocation outputs with `OP_SSRTX` , treasury additions with `OP_TADD` , treasury spend payouts with `OP_TGEN` , and the treasury spend signature script ends with `OP_TSPEND` . The tags let consensus code recognize these outputs, and impose maturity and other spending restrictions on them, without executing scripts.

The script engine itself enforces only the following; everything else about the tags is a transaction-level rule.

1. Executing a tag does nothing (Section 6.8.10). After the tag, a tagged pay-to-pubkey-hash script is executed exactly as the untagged form, and a tagged P2SH script is evaluated as P2SH (Rule 6.31).
2. Redeem scripts must not contain tags (Rule 6.32).
3. The treasury tags `0xc1` – `0xc3` are tags only when `ScriptVerifyTreasury` is set; otherwise they are upgradable no-operations.

The rules that restrict where tags may appear, and what must follow them, are specified elsewhere and summarized in Table 6.14.

Table 6.14 Where tag placement is enforced (informative)

Tag	Required forms	Specified in
`OP_SSTX`	Ticket output 0: tag followed by a P2PKH or P2SH script (26 or 24 bytes).	Chapter 8
`OP_SSTXCHANGE`	Ticket change outputs, and the optional change output (output 1) of a treasury add: tag followed by P2PKH or P2SH.	Chapter 8, Chapter 12
`OP_SSGEN`	Vote payment outputs: tag followed by P2PKH or P2SH.	Chapter 8
`OP_SSRTX`	Revocation outputs: tag followed by P2PKH or P2SH.	Chapter 8
`OP_TADD`	Treasury add and treasurybase output 0: the single byte `c1`.	Chapter 12
`OP_TGEN`	Treasury spend payouts: tag followed by P2PKH or P2SH.	Chapter 12
`OP_TSPEND`	Treasury spend signature script: `OP_DATA_64 <sig> OP_DATA_33 <key> OP_TSPEND` (100 bytes).	Chapter 12
any	Outputs of regular (non-stake) transactions must not contain any tag opcode (Rule 6.33).	Chapter 7

RULE 6.33 Detecting tag opcodes in a script

A script **contains a stake opcode** if, scanning its opcodes in order with version 0 parsing, an opcode in `0xba` – `0xbd` is found, or, when treasury rules are in effect, an opcode in `0xc1` – `0xc3`, before the end of the script or the first parse failure. If a parse failure is reached first, the predicate reports that failure as an error. Opcode bytes are only examined at opcode boundaries; a tag value inside push data does not count.

dcrd: txscript/script.go · ContainsStakeOpCodes | txscript/script.go · isStakeOpcode

The transaction-level rule that applies this predicate to regular transactions (and treats the parse error as invalidity) is in Chapter 7.

6.13 Verification Flags

The caller passes a set of flags that modify the engine. Table 6.15 lists every flag defined by dcrd, with its bit value and effect. Which flags consensus validation passes for a given block is specified in Chapter 7; as of the reference commit it is `ScriptVerifyCleanStack` and `ScriptVerifyCheckLockTimeVerify` for every block, plus `ScriptVerifyCheckSequenceVerify` and `ScriptVerifySHA256` when the `lnfeatures` agenda is active, plus `ScriptVerifyTreasury` when the `treasury` agenda is active. `ScriptDiscourageUpgradableNops` and `ScriptVerifySigPushOnly` are never used for consensus. dcrd's relay policy sets the first but not the second: it obtains the push-only requirement from a separate standardness check (Chapter 17).

Table 6.15 Script verification flags

Bit	Flag	Effect when set
`0x01`	`ScriptDiscourageUpgradableNops`	Executing an upgradable no-operation fails (Rule 6.21). Policy only.
`0x02`	`ScriptVerifyCheckLockTimeVerify`	Byte `0xb1` is `OP_CHECKLOCKTIMEVERIFY` (Rule 6.19); otherwise it is the no-operation `OP_NOP2`.
`0x04`	`ScriptVerifyCheckSequenceVerify`	Byte `0xb2` is `OP_CHECKSEQUENCEVERIFY` (Rule 6.20); otherwise it is `OP_NOP3`.
`0x08`	`ScriptVerifyCleanStack`	The data stack must hold exactly one element at the end (Rule 6.9).
`0x10`	`ScriptVerifySigPushOnly`	Every signature script must be push-only (Rule 6.3 step 2). Set by neither consensus nor dcrd's policy.
`0x20`	`ScriptVerifySHA256`	Byte `0xc0` is `OP_SHA256` (Rule 6.18); otherwise it is `OP_UNKNOWN192`.
`0x40`	`ScriptVerifyTreasury`	Bytes `0xc1` – `0xc3` are the treasury tags: they count as stake tags for P2SH detection and redeem-script prohibition. Otherwise they are `OP_UNKNOWN193` – `OP_UNKNOWN195`.

The signature operation counter (Section 6.14) and Rule 6.33 take a separate "treasury rules in effect" argument. It reflects the same agenda condition as `ScriptVerifyTreasury`, but is supplied by the caller rather than read from the engine flags.

6.14 Signature Operation Counting

To bound the cost of validating a block, consensus limits the number of **signature operations** a block may contain (Chapter 10). The count is a static estimate made without executing scripts. This section specifies the counting functions; the limit, and which transactions of a block are counted, are in Chapter 10.

ALGORITHM 6.7 Counting signature operations in one script

```
def count_sigops(script: bytes, precise: bool, treasury: bool) -> int:
    n, prev = 0, 0xff                       # prev starts as OP_INVALIDOPCODE
    for op, data in parse_prefix(script):   # opcodes before any parse failure
        if op == 0xc2:                      # OP_TSPEND
            if treasury: n += 1
        elif op in (0xac, 0xad, 0xbe, 0xbf):  # CHECKSIG(VERIFY), CHECKSIGALT(VERIFY)
            n += 1
        elif op in (0xae, 0xaf):            # CHECKMULTISIG(VERIFY)
            if precise and 0x51 <= prev <= 0x60:   # OP_1 .. OP_16
                n += prev - 0x50
            else:
                n += 20
        prev = op
    return n
```

dcrd: txscript/script.go · countSigOpsV0 | txscript/script.go · GetSigOpCount

`parse_prefix` yields the opcodes of the script that parse successfully before its first parse failure (all of them if there is none); a malformed script is counted up to the failure, not rejected. Opcodes are counted wherever they appear, including in branches that would not be executed and after an `OP_RETURN`. Push data is not inspected.

The **imprecise** count, `GetSigOpCount(script, treasury)`, is `count_sigops(script, False, treasury)`: every multisignature opcode counts 20. The **precise** count applies to P2SH spends.

ALGORITHM 6.8 Precise count for a P2SH spend

```
def precise_sigops(sig_script, pk_script, treasury) -> int:
    if not is_p2sh(pk_script):              # 23-byte untagged form only
        return count_sigops(pk_script, True, treasury)
    if len(sig_script) == 0 or not is_push_only(sig_script):
        return 0
    redeem = final_opcode_data(sig_script)   # b"" if unparseable or no data
    if len(redeem) == 0:
        return 0
    return count_sigops(redeem, True, treasury)
```

dcrd: txscript/script.go · GetPreciseSigOpCount | txscript/script.go · finalOpcodeData

In precise mode, a multisignature opcode counts the value of an immediately preceding `OP_1` – `OP_16`; it counts 20 if the preceding opcode is anything else, including `OP_0` and a numeric data push such as `01 11` (17). Despite its name, the "precise" count is therefore an upper bound for key counts 17 to 20.

RULE 6.34 Transaction signature operation count

The signature operation count of a transaction, given whether it is a coinbase, whether it is a vote, the UTXO view, and whether the treasury agenda is active (tr), is $A + B$, where:

A (the imprecise part):

- if tr and the transaction is a treasurybase, $A = 0$;
- otherwise A is the sum of `GetSigOpCount` over the signature scripts of all inputs (no inputs for a coinbase; all inputs except the first for a vote) plus the sum of `GetSigOpCount` over the public key scripts of all outputs.

B (the P2SH part):

- if the transaction is a coinbase, or if tr and it is a treasury spend or treasurybase, $B = 0$;
- otherwise, for every input (all inputs except the first for a vote), the referenced output **MUST** exist unspent in the view (`ErrMissingTxOut`); if its public key script is the 23-byte untagged P2SH form, add `precise_sigops(sig_script, pk_script, tr)` ; other inputs add nothing.

Every addition is performed in unsigned 32-bit arithmetic and **MUST NOT** overflow (`ErrTooManySigOps`).

dcrd: internal/blockchain/validate.go · CountTotalSigOps
internal/blockchain/validate.go · countSigOps
internal/blockchain/validate.go · countP2SHSigOps

IMPLEMENTATION PITFALL

Spends of stake-tagged P2SH outputs contribute nothing to B, because only the untagged 23-byte form is examined; the signature operations of their redeem scripts are not counted anywhere. Signature operations in a redeem script are also never counted by the imprecise pass, because the redeem script is push data within the signature script. And `OP_TSPEND` counts as one signature operation once the treasury agenda is active, although the engine executes it as a no-operation, since the treasury spend's signature is checked by the treasury rules (Chapter 12).

6.15 Other Consensus Predicates

Besides the engine, consensus code uses a few script predicates defined in the same package.

RULE 6.35 Provably unspendable outputs

An output with amount a and public key script P is **provably unspendable** if any of the following holds, regardless of its script version:

- $a = 0$;
- $|P| > 16384$;
- $|P| > 0$ and $P[0] =$ `0x6a` (`OP_RETURN`);
- P fails to parse under version 0 rules.

Provably unspendable outputs are never added to the UTXO set (Chapter 7), so no input can ever spend them.

dcrd: txscript/script.go · IsUnspendable
internal/blockchain/utxoviewpoint.go · UtxoViewpoint.AddTxOuts

IMPLEMENTATION PITFALL

This predicate ignores the script version and treats every zero-value output as unspendable. A zero-value output with script `51` (`OP_TRUE`), or a version-1 output whose script begins with `0x6a` , would pass the script engine but can never be spent, because it never enters the UTXO set. A script with `OP_RETURN` anywhere other than the first byte is not covered.

RULE 6.36 Strict null-data scripts

A script P with version v is a **strict null-data script with required length** ℓ if $v = 0$, $|P| \geq 1$, $P[0] =$ `0x6a` , and either $|P| = 1$ and $\ell = 0$, or the remainder $P[1:]$ parses as exactly one opcode which is canonical under Algorithm 6.4's `is_canonical_for_removal` and which is either a small-integer

> opcode (`OP_0` or `OP_1` – `OP_16`) with $\ell = 1$, or an opcode at most `0x4b` (`OP_DATA_75`) whose data length is exactly ℓ.
>
> dcrd: txscript/consensus.go · IsStrictNullData

The treasury rules use this predicate to check the `OP_RETURN` outputs of treasurybase and treasury spend transactions (Chapter 12). A companion predicate, **strict compressed public key encoding**, holds for exactly the 33-byte strings beginning with `0x02` or `0x03`; it is used by the treasury spend checks and by policy.

6.16 Standard Script Classes

The forms in this section are **policy**: dcrd's mempool and wallets recognize them, but consensus never requires a script to be standard (except where the stake and treasury rules require specific forms, which those chapters specify). The mapping between addresses and these templates is in Chapter 5; standardness policy is in Chapter 17.

Recognition is defined for script version 0 only; a script with any other version is non-standard. Table 6.16 lists the templates. In the templates, `‖` is concatenation, `H` is a 20-byte hash, `K33` a 33-byte compressed secp256k1 key (first byte `02` or `03`), `K65` a 65-byte uncompressed key (first byte `04`) and `E32` a 32-byte Ed25519 key.

Table 6.16 Standard version 0 script templates (policy)

Class	Bytes	Template
`pubkey`	35 / 67	`21` ‖ `K33` ‖ `ac` or `41` ‖ `K65` ‖ `ac`
`pubkey-ed25519`	35	`20` ‖ `E32` ‖ `51` ‖ `be`
`pubkey-schnorr-secp256k1`	36	`21` ‖ `K33` ‖ `52` ‖ `be`
`pubkeyhash`	25	`76 a9 14` ‖ `H` ‖ `88 ac`
`pubkeyhash-ed25519`	26	`76 a9 14` ‖ `H` ‖ `88 51 be`
`pubkeyhash-schnorr-secp256k1`	26	`76 a9 14` ‖ `H` ‖ `88 52 be`
`scripthash`	23	`a9 14` ‖ `H` ‖ `87`
`multisig`	var.	`OP_m` ‖ (n pushes of `K33`) ‖ `OP_n` ‖ `ae`, $1 \le m \le n \le 16$
`nulldata`	1–260	`6a`, or `6a` ‖ one canonical push of at most 256 bytes
`stakesubmission-pubkeyhash`	26	`ba` ‖ `pubkeyhash`
`stakesubmission-scripthash`	24	`ba` ‖ `scripthash`
`stakegen-pubkeyhash`	26	`bb` ‖ `pubkeyhash`
`stakegen-scripthash`	24	`bb` ‖ `scripthash`
`stakerevoke-pubkeyhash`	26	`bc` ‖ `pubkeyhash`
`stakerevoke-scripthash`	24	`bc` ‖ `scripthash`
`stakechange-pubkeyhash`	26	`bd` ‖ `pubkeyhash`
`stakechange-scripthash`	24	`bd` ‖ `scripthash`
`treasuryadd`	1	`c1`
`treasurygen-pubkeyhash`	26	`c3` ‖ `pubkeyhash`
`treasurygen-scripthash`	24	`c3` ‖ `scripthash`

Details of the less obvious templates:

- **Alternative-signature forms.** The signature type is pushed with a small-integer opcode: `51` (`OP_1`) for Ed25519 and `52` (`OP_2`) for EC-Schnorr-DCRv0. Type 0 is never standard. The Schnorr pay-to-pubkey form additionally requires the key's first byte to be `02` or `03`; the Ed25519 form imposes no condition on the 32 key bytes.
- **Multisig.** The first opcode is `OP_1` – `OP_16` giving m (`OP_0` is rejected). It is followed by one or more opcodes whose data is a strict compressed key (any push opcode qualifies, not only `OP_DATA_33`; the run

of keys ends at the first opcode whose data is not such a key), then a small-integer opcode equal to the number of keys n, with $n \geq m$, and then `ae` (`OP_CHECKMULTISIG`) as the last and only remaining byte. Uncompressed keys are not standard in multisig. The 16-key maximum is narrower than the consensus limit of 20.

- **Null data.** After `6a`, exactly one opcode of byte value at most `0x60` that is canonical under `is_canonical_for_removal` of Algorithm 6.4 and whose data is at most 256 bytes (`MaxDataCarrierSizeV0`).
- **Precedence.** A script is assigned the first class that matches, in the order of Table 6.16. The forms are mutually exclusive in practice, so the order rarely matters.

NOTE
dcrd also recognizes, for informational purposes only, the hash-based atomic swap contract below, intended to be used through P2SH. It is not a standard output class.

```
IF SIZE <n> EQUALVERIFY SHA256 <32 bytes> EQUALVERIFY DUP HASH160 <20 bytes>
ELSE <locktime> CHECKLOCKTIMEVERIFY DROP DUP HASH160 <20 bytes>
ENDIF EQUALVERIFY CHECKSIG
```

6.17 Test Vectors

All vectors in this section were produced by running dcrd's `txscript` package at the reference commit. Hexadecimal strings are byte strings in natural order; hashes are not reversed.

6.17.1 Numbers, Booleans and Hashes

Table 6.17 Script number encodings

Value	Encoding	Value	Encoding
0	(empty)	−256	`00 81`
1	`01`	32767	`ff 7f`
−1	`81`	−32768	`00 80 80`
16	`10`	8388607	`ff ff 7f`
127	`7f`	−8388608	`00 00 80 80`
−127	`ff`	2147483647	`ff ff ff 7f`
128	`80 00`	−2147483647	`ff ff ff ff`
−128	`80 80`	2147483648	`00 00 00 80 00`
255	`ff 00`	−2147483648	`00 00 00 80 80`
−255	`ff 80`	4294967294	`fe ff ff ff 00`
256	`00 01`	549755813887	`ff ff ff ff 7f`

Table 6.18 Script number decoding (maximum length 4 and 5)

Element	Max length 4	Max length 5
(empty)	0	0
`00`	`ErrMinimalData`	`ErrMinimalData`
`80`	`ErrMinimalData`	`ErrMinimalData`
`01 00`	`ErrMinimalData`	`ErrMinimalData`
`ff 00`	255	255
`ff 80`	−255	−255
`7f`	127	127
`ff`	−127	−127

Element	Max length 4	Max length 5
`00 80`	`ErrMinimalData`	`ErrMinimalData`
`ff ff ff 7f`	2147483647	2147483647
`ff ff ff ff`	−2147483647	−2147483647
`00 00 00 80 00`	`ErrNumOutOfRange`	2147483648
`ff ff ff ff 7f`	`ErrNumOutOfRange`	549755813887
`00 00 00`	`ErrMinimalData`	`ErrMinimalData`

Table 6.19 Boolean interpretation

Element	Value	Element	Value	Element	Value	Element	Value
(empty)	false	`00`	false	`80`	false	`00 80`	false
`80 00`	true	`00 00 80`	false	`00 01`	true	`01`	true

Table 6.20 Hash opcode outputs

Opcode	Input	Output
`OP_RIPEMD160`	(empty)	`9c1185a5c5e9fc54612808977ee8f548b2258d31`
`OP_RIPEMD160`	`"abc"`	`8eb208f7e05d987a9b044a8e98c6b087f15a0bfc`
`OP_SHA1`	(empty)	`da39a3ee5e6b4b0d3255bfef95601890afd80709`
`OP_SHA1`	`"abc"`	`a9993e364706816aba3e25717850c26c9cd0d89d`
`OP_SHA256`	(empty)	`e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`
`OP_SHA256`	`"abc"`	`ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad`
`OP_BLAKE256`	(empty)	`716f6e863f744b9ac22c97ec7b76ea5f5908bc5b2f67c61510bfc4751384ea7a`
`OP_BLAKE256`	`"abc"`	`1833a9fa7cf4086bd5fda73da32e5a1d75b4c3f89d5c436369f9d78bb2da5c28`
`OP_HASH160`	(empty)	`413320bee32a3bdb92b145d337316739d54a9287`
`OP_HASH160`	`"abc"`	`1625a1b44671b89a24386333ef9e00b207bca403`
`OP_HASH256`	(empty)	`d8ee5f957b78a961fb729098b4efb56440a14e05e3c55890f5edbc626380aaa6`
`OP_HASH256`	`"abc"`	`f1a44bb867d5ac7053d5e071d62d4513d5b243420d9863e2718823a161c1344a`

6.17.2 Signature Hash

The vectors below use a sample transaction with version 1, two inputs and two outputs. Input 0 spends output 1 of a transaction whose hash bytes are `00 01 … 1f`, in the regular tree, with sequence `ffffffff`; input 1 spends output 0 of `20 21 … 3f`, in the stake tree, with sequence `fffffffe`. The inputs' witness fields (value 5000000000 and 100000000 atoms, block heights 100 and 200, block indexes 2 and 3, signature scripts `51` and `52`) appear in the full serialization but do not affect the signature hash. Output 0 pays 1000 atoms to script version 0 `76 a9 14 01 02 … 14 88 ac`; output 1 pays 2000 atoms to `6a`. The lock time is `0x01020304` and the expiry `0x05060708`. The subscript is output 0's script, used here simply as a 25-byte example.

TEST VECTOR Signature hash: the sample transaction

Item	Value
Full serialization	`0100000002000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f0100000000ffffffff2021222324252627282922b2c2d2e2f303132333435363738393a3b3c3d3e3f00000000001feffffff02e8030000000000000000001976a9140102030405060708090a0b0c0d0e0f101112131488acd007000000000000000016a040302010807060502002f2052a0100000064000000020000000151000e1f50500000000c8000000030000000152`
Transaction hash (internal order)	`f3deadbbf50b1c407d09bc893408432844a0685df21ea15c4a73a9eba6480c5a`
Subscript	`76a9140102030405060708090a0b0c0d0e0f101112131488ac`

TEST VECTOR Signature hash: SIGHASH_ALL (`0x1`), input 0

Item	Value
Prefix serialization	0100010002000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f010000000ffffffff202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f0000000001feffffff02e8030000000000000000001976a9140102030405060708090a0b0c0d0e0f101112131488acd007000000000000000016a0403020108070605
Prefix hash	f3deadbbf50b1c407d09bc893408432844a0685df21ea15c4a73a9eba6480c5a
Witness serialization	01000300021976a9140102030405060708090a0b0c0d0e0f101112131488ac00
Witness hash	bb75f775cbeee9c804eb983c29fcc2d97a2264e6fa75a87a15a06dd4286982e5
Final serialization	01000000f3deadbbf50b1c407d09bc893408432844a0685df21ea15c4a73a9eba6480c5abb75f775cbeee9c804eb983c29fcc2d97a2264e6fa75a87a15a06dd4286982e5
Signature hash	f4ce6e37ab599bcd9c770d086c79749a77f99e4a50e774caf5ff50a123feff91

TEST VECTOR Signature hash: SIGHASH_NONE (`0x2`), input 0

Item	Value
Prefix serialization	0100010002000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f010000000ffffffff202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f0000000001000000000000403020108070605
Prefix hash	8aa0fda4bf9cc8936b0eec1cabc6795b0de37badda0b26b7063d475d74bd967b
Witness serialization	01000300021976a9140102030405060708090a0b0c0d0e0f101112131488ac00
Witness hash	bb75f775cbeee9c804eb983c29fcc2d97a2264e6fa75a87a15a06dd4286982e5
Final serialization	020000008aa0fda4bf9cc8936b0eec1cabc6795b0de37badda0b26b7063d475d74bd967bbb75f775cbeee9c804eb983c29fcc2d97a2264e6fa75a87a15a06dd4286982e5
Signature hash	03319bed79cbb29b3c6fe90e4ff6e301071ddeb3568d16462ca5ffeed787d44b

TEST VECTOR Signature hash: SIGHASH_SINGLE (`0x3`), input 1

Item	Value
Prefix serialization	0100010002000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f010000000000000002021222324252627282 92a2b2c2d2e2f303132333435363738393a3b3c3d3e3f0000000001feffffff02ffffffffffffffff000000d007000000000000000016a0403020108070605
Prefix hash	1c60e4485f51debb2dcdc9893293a6b4e6290a3d66c57756ed95cb74b7548927
Witness serialization	0100030002001976a9140102030405060708090a0b0c0d0e0f101112131488ac
Witness hash	4902291a3dcd005c1a087f947d7d37cc0de0f06daa470a5d7713f8d53b093cc7
Final serialization	030000001c60e4485f51debb2dcdc9893293a6b4e6290a3d66c57756ed95cb74b754892749022 91a3dcd005c1a087f947d7d37cc0de0f06daa470a5d7713f8d53b093cc7
Signature hash	4641bb9be1f590a5cae964b188317cf168e158d24cfa27c7280941c684e42dc2

TEST VECTOR Signature hash: SIGHASH_ALL | SIGHASH_ANYONECANPAY (`0x81`), input 0

Item	Value
Prefix serialization	0100010001000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f0100000000ffffffff02e8030000000000000000001976a9140102030405060708090a0b0c0d0e0f101112131488acd007000000000000000016a0403020108070605
Prefix hash	12912c99d91053050096a5745dc0d45c829c73fd45f7fa9164649c8fada6b272
Witness serialization	01000300011976a9140102030405060708090a0b0c0d0e0f101112131488ac
Witness hash	3db60c902d6e97e3b0a3ec8b55a7c330392e4740a46a3b79a93445c680e58a45
Final serialization	8100000012912c99d91053050096a5745dc0d45c829c73fd45f7fa9164649c8fada6b2723db60c902d6e97e3b0a3ec8b55a7c330392e4740a46a3b79a93445c680e58a45
Signature hash	4a1ecb30301eb4a7445ffca36202deda5f9552479fd2b223e3589f8738dbbde3

Table 6.21 Signature hashes of the sample transaction for every hash type

Input	Hash type	Signature hash
0	`0x1`	`f4ce6e37ab599bcd9c770d086c79749a77f99e4a50e774caf5ff50a123feff91`
0	`0x2`	`03319bed79cbb29b3c6fe90e4ff6e301071ddeb3568d16462ca5ffeed787d44b`
0	`0x3`	`e8fb9b72477a691a3c5c05aa4147a3fb5feaa4e43940ead1107ef94de9392cd4`
0	`0x81`	`4a1ecb30301eb4a7445ffca36202deda5f9552479fd2b223e3589f8738dbbde3`
0	`0x82`	`487f799838a05a534aa50daa92487e17c2715cc1ada7ce48c3a9fa052d05740f`
0	`0x83`	`e2e9e2e61df3dc7a668dc0418f5e1fa4d56f77d2dd2a8a427780d8bc423e54f3`
1	`0x1`	`901d5f6eac2d838d4ca1f3cc5a4ea4e2c10394ac507a7a2bb891958412d03871`
1	`0x2`	`93deae66a64e67888a1a03a319923cd3f948f27a375ed00d926ac19365b3fa4d`
1	`0x3`	`4641bb9be1f590a5cae964b188317cf168e158d24cfa27c7280941c684e42dc2`
1	`0x81`	`9df281682b102567a25fb8d3ca09f392efe5ee5499d8c12c0254974c05b070c5`
1	`0x82`	`93c4e51c4dc384c723940f0b64f4ef450dee23d12f9806516d83470ea2fa399c`
1	`0x83`	`d71ccf15b435685367f77257aef4ac96a959838c42d4164de2a795da0a9254bb`

The `SIGHASH_ALL` prefix hash equals the transaction hash, as stated in Section 6.10. In the `SIGHASH_SINGLE` vector for input 1, output 0 appears blanked (`ff…ff`, version `0000`, script length `00`) and input 0's sequence is 0. Signing input 2 of a two-output transaction with `SIGHASH_SINGLE` has no defined hash (Rule 6.30); dcrd's `txscript/testdata/sighash.json` contains such cases, together with real mainnet transactions and undefined hash types, and all of its entries are reproduced by Algorithm 6.6.

6.17.3 Signed Spends

TEST VECTOR Signed pay-to-pubkey-hash spend (input 0 of the sample transaction)

Item	Value
Private key	`0000000000000000000000000000000000000000000000000000000000000001`
Public key (compressed)	`0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798`
HASH160 of public key	`e280cb6e66b96679aec288b1fbdbd4db08077a1b`
Public key script	`76a914e280cb6e66b96679aec288b1fbdbd4db08077a1b88ac`
Signature hash (ALL)	`5a5f06a3652928aae830f38a45325564e1973fb10073ef671df0be1e69466331`
Signature element	`3045022100a944fc7a28deab0b5c3f07b834995b2d2d5f42d708d0d7a04e3bec572c5779b202202d2730e2e9116305e8e731c7de302e11f43c2e9f1d87e19dd21d8c165175bab001`
Signature script	`483045022100a944fc7a28deab0b5c3f07b834995b2d2d5f42d708d0d7a04e3bec572c5779b202202d2730e2e9116305e8e731c7de302e11f43c2e9f1d87e19dd21d8c165175bab001210279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798`
Result (consensus flags)	valid
Same, one bit of R flipped (a bit that leaves the DER encoding valid)	invalid: `ErrEvalFalse` (`OP_CHECKSIG` pushes false)
Same, hash type byte `04`	invalid: `ErrInvalidSigHashType` (encoding failure)

TEST VECTOR EC-Schnorr-DCRv0 via `OP_CHECKSIGALT` (type 2, same key)

Item	Value
Public key script	`210279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f8179852be`
Signature hash (ALL)	`99b24bd56eb2ff75a229d24a2a0d11f3025380add7655abce1e64a2fe5ceebf4`
Signature script	`415326e2256467605135e786593379842f6087245aa4fcd1fa428849e20a8cb412d59effa4e8366ebe1cfbc7bac65a04e1aa9946e0ec4b10e154800e05edcdee5a01`
Result (consensus flags)	valid

TEST VECTOR Ed25519 via `OP_CHECKSIGALT` (type 1)

Item	Value
Public key script	`204cb5abf6ad79fbf5abbccafcc269d85cd2651ed4b885b5869f241aedf0a5ba2951be`
Signature hash (ALL)	`d9fbdf699eafab052dad4c7c873444809717dd3643329adbe6e2c043c7b869c3`
Signature script	`410a299ecbc9e4800db91797d9090b21fdd76aba7fcab02c5dba370cc5ef4b79472705d52fe57bfe9ed7a034663cc0301c191006e2790f134a38b2bb77ef57fc0101`
Result (consensus flags)	valid

TEST VECTOR 2-of-3 multisignature, bare and through P2SH

Item	Value
Private keys	1, 3 and 6 (as 32-byte big-endian scalars); signatures by keys 1 and 6
Public key script	`52210279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f817982102f9308a019258c31049344f85f89d5229b531c845836f99b08601f113bce036f92103fff97bd5755eeea420453a14355235d382f6472f8568a18b2f057a146029755653ae`
Signature script (no dummy element)	`483045022100f5600c0a319d9d71df82ccbee4d0dd08a6a1b9aa840b29700b86cecf4660ad1f022030475c9aa791e59d0ff7cb08df2112743fe28886127ea9d9ed12a9e3b21bb6a00147304402206194d7eecb75ef075cda4e46d50bf66caef91350b12ccc767dd13e2d7f7c2669022046f8c544e3c4a4cb031f8058be3bc2882fd2b9c556f9fad200df758b3ef165e201`
Result	valid
Signatures in reverse order	invalid: `ErrEvalFalse`
With a leading `OP_0` dummy	invalid under consensus flags: `ErrCleanStack` (valid without the clean-stack flag)
P2SH public key script	`a914f86b5a7c6d32566aa4dccc04d1533530b4d64cf387`
P2SH signature script	`483045022100f5600c0a319d9d71df82ccbee4d0dd08a6a1b9aa840b29700b86cecf4660ad1f022030475c9aa791e59d0ff7cb08df2112743fe28886127ea9d9ed12a9e3b21bb6a00147304402206194d7eecb75ef075cda4e46d50bf66caef91350b12ccc767dd13e2d7f7c2669022046f8c544e3c4a4cb031f8058be3bc2882fd2b9c556f9fad200df758b3ef165e2014c6952210279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f817982102f9308a019258c31049344f85f89d5229b531c845836f99b08601f113bce036f92103fff97bd5755eeea420453a14355235d382f6472f8568a18b2f057a146029755653ae`
P2SH result	valid; imprecise signature operations 0 + 0, precise (P2SH) count 3

6.17.4 Lock-Time Opcodes

In Table 6.22 the public key script is the listed opcode sequence followed by `OP_DROP OP_1`, the signature script is empty, and the consensus flags of a block with all agendas active are used.

Table 6.22 `OP_CHECKLOCKTIMEVERIFY` and `OP_CHECKSEQUENCEVERIFY`

Script	Tx ver.	Lock time	Sequence	Result
`100 CLTV`	1	100	`0`	valid
`101 CLTV`	1	100	`0`	`ErrUnsatisfiedLockTime`

Script	Tx ver.	Lock time	Sequence	Result
`500000000 CLTV`	1	100	`0`	`ErrUnsatisfiedLockTime` (kind mismatch)
`100 CLTV`	1	100	`ffffffff`	`ErrUnsatisfiedLockTime` (input final)
`-1 CLTV`	1	100	`0`	`ErrNegativeLockTime`
`4294967295 CLTV`	1	4294967295	`0`	valid (5-byte operand)
`549755813887 CLTV`	1	4294967295	`0`	`ErrUnsatisfiedLockTime`
`10 CSV`	2	0	`0000000a`	valid
`11 CSV`	2	0	`0000000a`	`ErrUnsatisfiedLockTime`
`10 CSV`	1	0	`0000000a`	`ErrUnsatisfiedLockTime` (version)
`2147483648 CSV`	1	0	`0000000a`	valid (disable bit in operand)
`10 CSV`	2	0	`8000000a`	`ErrUnsatisfiedLockTime` (disable bit in input)
`4194314 CSV`	2	0	`0000000a`	`ErrUnsatisfiedLockTime` (kind mismatch)
`4194314 CSV`	2	0	`0040000a`	valid
`4294967306 CSV`	2	0	`0000000a`	valid (bits above 31 ignored)
`-1 CSV`	2	0	`0000000a`	`ErrNegativeLockTime`

6.17.5 Execution Behaviour

Table 6.23 gives representative cases. Scripts are written in the short form of dcrd's test data: a decimal number is its minimal push, `'s'` pushes the ASCII bytes of `s`, `0x…` inserts raw bytes, `<…>{k}` repeats the enclosed tokens k times, `'b'{k}` pushes a single element of k copies, and names are opcodes without the `OP_` prefix. An entry number refers to the zero-based position in `txscript/testdata/script_tests.json`; the other rows were run directly. Unless noted, no flags are set and the script version is 0.

Table 6.23 Script execution vectors

Entry	Signature script	Public key script	Result
541	`0`	`IF 'b'{2049} ENDIF 1`	`ErrElementTooBig` (unexecuted push)
542	`0`	`IF <NOP>{256} ENDIF 1`	`ErrTooManyOperations` (unexecuted ops count)
2239	`0`	`IF 0x50{255} ENDIF 1`	valid (`0x50` is not counted)
589	`0`	`IF ELSE 1 ELSE VERIF ENDIF`	`ErrReservedOpcode`
1838	`0`	`IF CODESEPARATOR ENDIF 1`	`ErrDisabledOpcode`
—	`1`	`0 IF VER ENDIF`	valid (`OP_VER` unexecuted)
547	`0 IF 0x4c 0x00 ENDIF 1`	(empty)	valid (unexecuted non-minimal push)
389	`0x4c 0x00`	`DROP 1`	`ErrMinimalData`
413	`0x01 0x00`	`1`	valid
433	`0x01 0x00`	`NOT DROP 1`	`ErrMinimalData` (numeric use)
524	`1 0x01 0x80`	`IF 0 ENDIF`	valid (negative zero is false)
606	`1 0x05 0x0100000000`	`VERIFY`	valid (long boolean)
508	`0`	`IF 0 ELSE 1 ELSE 0 ENDIF`	valid (two `ELSE`s)
543	`1 IF`	`1 ENDIF`	`ErrUnbalancedConditional`
1379	`2147483647 DUP ADD`	`4294967294 EQUAL`	valid (5-byte result)
1319	`2147483647`	`1ADD 1SUB 1`	`ErrNumOutOfRange`
1331	`2 2MUL`	`4 EQUAL`	`ErrEvalFalse` (`OP_2MUL` is a no-op)
1439	`-7 3`	`MOD -1 EQUAL`	valid
1484	`1 31`	`LSHIFT -2147483648 EQUAL`	valid

Entry	Signature script	Public key script	Result
1485	`1 32`	`LSHIFT 0 EQUAL`	valid
—	(empty)	`-1 32 RSHIFT -1 EQUAL`	valid
1493	`1 33`	`LSHIFT 0 EQUAL`	`ErrOverflowShift`
—	(empty)	`65536 65536 MUL 0 EQUAL`	valid (wrapping)
920	`'abcd' 3 1`	`SUBSTR 'bc' EQUAL`	valid
925	`'' 2147483647 2147483646`	`SUBSTR '' EQUAL`	valid
2040	(empty)	`0 0 CHECKMULTISIGVERIFY DEPTH 0 EQUAL`	valid
2037	`1`	`<0 0 CHECKMULTISIGVERIFY>{255}`	valid (255 operations)
2038	`1`	`<0 0 CHECKMULTISIGVERIFY>{256}`	`ErrTooManyOperations`
2039	`1`	`<NOP>{3} <0 'a' … 't' 20 CHECKMULTISIGVERIFY>{12}`	valid ($3 + 12 \times 21 = 255$)
1921	`0`	`0x21 <key 3> CHECKSIG`	`ErrEvalFalse` (empty signature)
—	(empty)	`3 CHECKSIGALT`	valid (unassigned type)
—	(empty)	`128 CHECKSIGALT`	`ErrNumOutOfRange`
—	`'x' 'y'`	`1 CHECKSIGALT NOT`, clean stack	`ErrCleanStack` (`'x'` left behind)
2246	`1 3 DATA_2 ADD SSTX`	`HASH160 DATA_20 0xdcbd…ff8f EQUAL`	`ErrP2SHStakeOpCodes`
2263	`1 3 DATA_2 ADD TADD`	`HASH160 DATA_20 0xdd89…fb05 EQUAL`, `TREASURY`	`ErrP2SHStakeOpCodes`
2267	as 2263	as 2263, no flags	valid (`0xc1` is `OP_UNKNOWN193`)
2088	`NOP1`	`NOP10`	`ErrEmptyStack`
—	(empty)	`RETURN`, version 1	valid (not executed)
—	(empty)	(empty), version 1	`ErrEvalFalse` (construction)
—	`NOP`	`HASH160 0x14 <20 zero bytes> EQUAL`, version 1	`ErrNotPushOnly`
—	(empty)	`0x4c`, version 1	`ErrMalformedPush`

For the P2SH output whose redeem script is the single byte `51` (public key script `a914f5a8302ee8695bf836258b8f2b57b38a0be14e4787`): the signature script `01 51` is valid; `51`, `00`, the empty script and `61 01 51` fail construction with `ErrNotPushOnly`; and `01 51 01 51` is valid without the clean-stack flag but fails with it (`ErrCleanStack`).

6.17.6 Signature Operation Counts

Table 6.24 Signature operation counts (treasury rules in effect unless noted)

Script	Imprecise	Precise
`DUP HASH160 <20> EQUALVERIFY CHECKSIG`	1	1
`2 <K33> <K33> <K33> 3 CHECKMULTISIG`	20	3
`0 CHECKMULTISIG`	20	20
`17 CHECKMULTISIG` (17 is pushed as `01 11`)	20	20
`16 CHECKMULTISIGVERIFY`	20	16
`CHECKSIG CHECKSIGVERIFY CHECKSIGALT CHECKSIGALTVERIFY`	4	4
`TSPEND`	1 (0 without treasury rules)	1
`CHECKSIG 0x4c` (parse failure after the first opcode)	1	1
`CHECKSIG 0x01 0xac` (`0xac` is push data)	1	1
`0 IF CHECKSIG ENDIF`	1	1

6.17.7 Provably Unspendable Outputs

Table 6.25 Provably unspendable outputs

Amount	Script	Unspendable
0	`51`	yes
1	`6a`	yes
1	`51 6a`	no
1	`4c` (parse failure)	yes
1	`51`	no
1	(empty)	no

6.18 Error Kinds

Table 6.26 maps every script error kind that can arise in consensus validation to the rule that produces it. The kinds `ErrInvalidIndex`, `ErrUnsupportedScriptVersion`, `ErrScriptUnfinished` and `ErrInvalidProgramCounter` indicate misuse of the reference implementation's interface and cannot arise from transaction data; `ErrP2SHTreasuryOpCodes` is defined but never produced (treasury tags in redeem scripts produce `ErrP2SHStakeOpCodes`); `ErrDiscourageUpgradableNOPs` and `ErrNotPushOnly` from the push-only flag arise only under policy flags.

Table 6.26 Script error kinds and the rules that produce them

Error kind	Rule
`ErrMalformedPush`	Rule 6.2, Rule 6.3, Rule 6.31
`ErrScriptTooBig`	Rule 6.3
`ErrNotPushOnly`	Rule 6.3
`ErrP2SHStakeOpCodes`	Rule 6.3, Rule 6.32
`ErrEvalFalse`, `ErrEmptyStack`, `ErrCleanStack`	Rule 6.9, Rule 6.31, Rule 6.3
`ErrUnbalancedConditional`	Rule 6.4, Rule 6.8
`ErrStackOverflow`	Rule 6.5
`ErrDisabledOpcode`, `ErrTooManyOperations`, `ErrElementTooBig`	Rule 6.6, Rule 6.28, Rule 6.13
`ErrReservedOpcode`	Rule 6.6, Rule 6.22
`ErrMinimalData`	Rule 6.7, Rule 6.11
`ErrNumOutOfRange`	Rule 6.11
`ErrInvalidStackOperation`	Section 6.8 (all opcodes), Rule 6.12
`ErrVerify`, `ErrEqualVerify`, `ErrNumEqualVerify`	Section 6.8.2
`ErrCheckSigVerify`, `ErrCheckMultiSigVerify`, `ErrCheckSigAltVerify`	Section 6.9
`ErrEarlyReturn`	Table 6.5
`ErrNegativeSubstrIdx`, `ErrOverflowSubstrIdx`	Rule 6.14, Rule 6.15
`ErrNegativeRotation`, `ErrOverflowRotation`	Rule 6.16
`ErrNegativeShift`, `ErrOverflowShift`	Rule 6.17
`ErrDivideByZero`	Table 6.8
`ErrNegativeLockTime`, `ErrUnsatisfiedLockTime`	Rule 6.19, Rule 6.20
`ErrInvalidSigHashType`	Rule 6.23
`ErrSigTooShort` … `ErrSigHighS` (16 kinds)	Rule 6.24
`ErrPubKeyType`	Rule 6.25
`ErrInvalidPubKeyCount`, `ErrInvalidSignatureCount`	Rule 6.28
`ErrInvalidSigHashSingleIndex`	Rule 6.30 (fails only via `OP_CHECKMULTISIG`)

CHAPTER 7

Transactions

A transaction moves value. It consumes previously created, still unspent transaction outputs and creates new ones. This chapter specifies every consensus rule whose subject is a single transaction: what makes a transaction well formed, how it is classified, when it may appear in a block, which outputs it may spend, how its fee is determined, and how its scripts are invoked. Rules whose subject is a whole block (sizes, sigop totals, header commitments, the order of block connection) are specified in Chapter 10; the detailed structure of tickets, votes and revocations in Chapter 8; of treasury transactions in Chapter 12; and of the coinbase in Chapter 11. The serialization of transactions and their hashes is given in Chapter 4, and the script engine in Chapter 6.

7.1 Overview

Every transaction in a block passes through the stages listed in Table 7.1. A block is invalid if any transaction in it fails any stage. The stages are listed approximately in the order in which dcrd applies them (dcrd checks expiry before the context stage, and runs the input, sequence-lock and script stages first for the whole stake tree and then for the whole regular tree); the order affects only which error is reported, except where noted in Section 7.10, because the state of the set of unspent outputs changes as transactions are connected.

Table 7.1 Stages of transaction validation within block validation

Stage	Content	Section
Sanity	Context-free structural checks: counts, size, amounts, duplicate inputs.	Section 7.4
Context	Checks that depend on agenda state but not on the set of unspent outputs: versions, input forms per type, stake opcodes in regular outputs, placement of each type in the correct tree.	Section 7.3, Section 7.5
Expiry	The block height must be below the transaction's expiry.	Section 7.6
Finality	The absolute lock time must have passed.	Section 7.7
Inputs	Each input must reference an available output; maturity, tag, value and fee rules; the transaction is then connected.	Section 7.9, Section 7.10
Sequence locks	Relative lock times must be satisfied.	Section 7.8
Scripts	Every input script is executed against the script of the output it spends.	Section 7.12

> **NOTE**
> A node deciding whether to accept an unconfirmed transaction into its memory pool applies these same rules from the point of view of a hypothetical next block, together with additional, stricter policy rules (standardness, fees, limits on versions and script forms) that are not consensus. Policy is described in Chapter 17.

7.1.1 Agendas affecting transaction rules

Several transaction rules changed through stakeholder votes. Each is identified below by its agenda identifier. Throughout this chapter, "agenda *X* is active" always means that *X* is active **for the block being validated**

(the block that contains the transaction), which dcrd determines from the parent of that block as described in Chapter 13.

Table 7.2 Agendas that change transaction-level rules

Agenda id	DCP	Effect on transaction rules
`lnfeatures`	DCP-0002, DCP-0003	Enables sequence locks; finality uses the parent's past median time instead of the header timestamp; enables `OP_CHECKSEQUENCEVERIFY` and `OP_SHA256` in scripts.
`fixlnseqlocks`	DCP-0004	Corrected the set of outputs visible to sequence-lock evaluation. No longer consulted by dcrd; see Section 7.8.4.
`treasury`	DCP-0006	Treasury transaction types; coinbase must be version 3; treasury opcodes; maturity of vote and revocation outputs raised to `CoinbaseMaturity`; `OP_TGEN` maturity; `OP_TADD` outputs unspendable.
`explicitverupgrades`	DCP-0008	Transaction versions above 3 and regular-transaction output script versions above 0 are rejected.
`autorevocations`	DCP-0009	Revocations must be version 2; scripts of version 2 or later revocations are not executed.

7.1.2 Constants

Table 7.3 Constants and parameters used in this chapter

Name	Value	Meaning
`MaxAmount`	2100000000000000	Largest valid amount in atoms ($21 \times 10^6 \times 10^8$).
`MaxTxSize`	393216 (mainnet)	Largest serialized transaction, in bytes. 1000000 on the other networks.
`CoinbaseMaturity`	256 (mainnet)	Blocks before coinbase and related outputs may be spent. 16 on the other networks.
`SStxChangeMaturity`	1	Blocks before ticket change outputs may be spent; 1 on every network.
`TicketMaturity`	256 (mainnet)	Used by staking maturity rules; 16 on the other networks.
`StakeBaseSigScript`	`0x0000` (mainnet)	Required signature script of a vote's stakebase input (two bytes). `0x7357` on regnet, `0x0000` on testnet3 and simnet.
`MinCoinbaseScriptLen`	2	Minimum coinbase and stakebase signature script length.
`MaxCoinbaseScriptLen`	100	Maximum coinbase and stakebase signature script length.
`TSpendScriptLen`	100	Exact treasury spend signature script length.
`LockTimeThreshold`	500000000	Lock times below this are heights, others are Unix times.
`NoExpiryValue`	0	Expiry value meaning "never expires".
`NullValueIn`	−1	Placeholder input amount allowed by the sanity rules.
`NullBlockHeight`	0	Null input witness block height.
`NullBlockIndex`	`ffffffff`	Null input witness block index.
`MaxScriptSize`	16384	Output scripts longer than this are never spendable.
`MaxSigOpsPerBlock`	5000	Block signature-operation limit (Chapter 10).
`TxVersionSeqLock`	2	Lowest transaction version subject to sequence locks.
`TxVersionTreasury`	3	Version of treasury transactions and post-treasury coinbases.
`TxVersionAutoRevocations`	2	Required revocation version under `autorevocations`.

7.2 The UTXO Model

Decred, like Bitcoin [1], keeps no account balances. The ledger state is the **UTXO set**: the set of transaction outputs that exist on the chain and have not been consumed. A transaction input names one entry of the set by its **outpoint** and consumes it; the transaction's outputs become new entries. Validation of a transaction therefore needs, for each input, the entry it references.

7.2.1 Outpoints and transaction trees

A block carries two ordered lists of transactions, the **regular tree** and the **stake tree** (Chapter 4). Stake transactions (tickets, votes, revocations and the treasury types) live in the stake tree; all others, including the coinbase, live in the regular tree. The tree is part of an output's identity. An outpoint is the triple

$$(\text{hash}, \text{index}, \text{tree}) \tag{7.1}$$

where `hash` is the transaction hash of the creating transaction, `index` its zero-based output index (`uint32`), and `tree` is 0 for the regular tree and 1 for the stake tree (`int8`). Two outpoints are equal only if all three fields are equal.

RULE 7.1 Outpoint tree must match the creating transaction

The UTXO entry created by output k of transaction T is keyed by $(\text{hash}(T), k, t)$, where $t = 1$ if T is of any stake type (Section 7.3) and $t = 0$ otherwise. An input whose outpoint carries the other tree value references a nonexistent entry and the transaction **MUST** be rejected under Rule 7.26.

dcrd: internal/blockchain/utxoviewpoint.go · addTxOuts
internal/blockchain/validate.go · CheckTransactionInputs

The **null outpoint** is the outpoint with the all-zero hash, index `ffffffff` and tree 0. It references nothing and is used as the placeholder input of transactions that create value from nothing (coinbase, vote stakebase, treasurybase, treasury spend).

7.2.2 Unspent output entries

An entry records the information consensus later needs about the output and its creating transaction (Table 7.4). These fields are fixed at creation; only the spent/unspent state changes.

Table 7.4 Content of a UTXO entry as used by consensus

Field	Type	Meaning
amount	`int64`	Value of the output in atoms.
script version	`uint16`	Version of the output script.
script	bytes	The output script.
height	`uint32`	Height of the block containing the creating transaction.
index	`uint32`	Position of the creating transaction within its tree (regular-tree position for regular transactions, stake-tree position for stake transactions).
coinbase flag	bool	The creating transaction is a coinbase (Algorithm 7.1, evaluated with the `treasury` state of the creating block).
expiry flag	bool	The creating transaction's expiry field is nonzero.
transaction type	enum	`stake_type` of the creating transaction (Algorithm 7.2; a coinbase is recorded as `REGULAR`), used by staking rules.
ticket outputs	list	For output 0 of a ticket only: the amount, script version and script of every output of the ticket, needed later to validate the vote or revocation that spends it (Chapter 8).

RULE 7.2 Outputs that never enter the UTXO set

When a transaction is connected, an entry is created for each of its outputs **except**:

1. any output whose amount is 0;
2. any output whose script is longer than `MaxScriptSize` (16384) bytes;
3. any output whose script is non-empty and begins with the byte `0x6a` (`OP_RETURN`);
4. any output whose script does not parse as a version 0 script (Chapter 6), regardless of the output's script version;
5. output 0 of a treasury add;
6. every output of the treasurybase (when `treasury` is active, the stake transaction at index 0 adds no entries at all).

Such outputs can never be spent: an input referencing one **MUST** be rejected under Rule 7.26.

dcrd: internal/blockchain/utxoviewpoint.go · addTxOuts, connectStakeTransaction
txscript/script.go · IsUnspendable

An entry is **spent** once a connected transaction has consumed it; from then on it is no longer part of the UTXO set and no later transaction may reference it. Spending is undone only when the spending transaction is disconnected, either by a chain reorganization or because the next block disapproves the regular tree containing it (Chapter 10); disconnection restores the entry with all of its original fields.

NOTE
Condition 4 cannot produce a difference for a valid block, because every output of a regular transaction must already tokenize as version 0 (Rule 7.12) and every output of a stake transaction is a version 0 script of a fixed form. The genesis block is never connected to the UTXO set; its outputs are not spendable.

7.2.3 Input witness fields

Each input carries, in the witness part of its serialization (Chapter 4), three fields that duplicate information about the referenced output: `ValueIn` (`int64`), `BlockHeight` (`uint32`) and `BlockIndex` (`uint32`). They were designed to let lightweight clients prove fraud and are called the **fraud proof**. Consensus constrains them as follows.

Table 7.5 Constraints on input witness fields

Input	`ValueIn`	`BlockHeight`, `BlockIndex`
Any input	-1 or in $[0, \texttt{MaxAmount}]$ (Rule 7.8).	—
Input spending a UTXO entry	Equal to the entry's amount (Rule 7.3).	Equal to the entry's height and index (Rule 7.3).
Coinbase input	Equal to the block's coinbase subsidy (Chapter 11).	0 and `ffffffff` (Rule 7.13).
Vote stakebase (input 0)	Equal to the vote subsidy (Chapter 8).	0 and `ffffffff` (part of the vote's structural definition, Chapter 8).
Treasurybase input	Equal to the treasury subsidy (Chapter 11, Chapter 12).	0 and `ffffffff` (Rule 7.16).
Treasury spend input	The amount withdrawn, matching the commitment in output 0 (Chapter 12).	0 and `ffffffff` (Rule 7.17).

RULE 7.3 Witness fields of spending inputs
For every input that references a UTXO entry (every input except those listed in Rule 7.25), the input's `ValueIn` **MUST** equal the entry's amount, `int64(BlockHeight)` **MUST** equal the entry's height, and `BlockIndex` **MUST** equal the entry's index. These comparisons are exact.

dcrd: internal/blockchain/validate.go · CheckTransactionInputs

IMPLEMENTATION PITFALL
The witness fields are not covered by the transaction hash (Chapter 4), so two encodings of the same transaction with different witness values have the same hash, and only one of them is valid in a given chain context. For an output created earlier in the same block, `BlockHeight` is the height of the block itself and `BlockIndex` is the creating transaction's position in its own tree.

7.3 Classification

Many rules depend on what kind of transaction is being validated. Decred has no explicit type field: the type is inferred from the structure of the transaction. The structural predicates are defined elsewhere and are only named here: `is_ticket`, `is_vote`, `is_revocation` in Chapter 8; `is_treasury_add`, `is_treasury_spend`,

`is_treasurybase` in Chapter 12. Each predicate is a pure function of the transaction alone. The coinbase predicate is given below because it depends on the `treasury` agenda.

ALGORITHM 7.1 Coinbase form

```
ZERO_HASH = bytes(32)

def is_coinbase_form(tx, treasury_active: bool) -> bool:
    if treasury_active and tx.version != 3:
        return False
    if len(tx.inputs) != 1:
        return False
    prev = tx.inputs[0].prev_out
    if prev.index != 0xffffffff or prev.hash != ZERO_HASH:
        return False                    # prev.tree is NOT examined here
    if treasury_active and is_tspend_like(tx):
        return False
    return True

def is_tspend_like(tx) -> bool:         # only reached with exactly one input
    if len(tx.outputs) < 2:
        return False
    sig = tx.inputs[0].sig_script
    if len(sig) == 0 or len(tx.outputs[0].pk_script) == 0 \
            or len(tx.outputs[1].pk_script) == 0:
        return False
    return (sig[-1] == 0xc2                          # OP_TSPEND
            and tx.outputs[0].pk_script[0] == 0x6a   # OP_RETURN
            and tx.outputs[1].pk_script[0] == 0xc3)  # OP_TGEN
```

dcrd: blockchain/standalone/tx.go · IsCoinBaseTx

ALGORITHM 7.2 Transaction type

```
STAKE_TYPES = {TICKET, VOTE, REVOCATION, TREASURY_ADD, TREASURY_SPEND, TREASURYBASE}

def stake_type(tx):                     # independent of every agenda
    if is_ticket(tx):       return TICKET
    if is_vote(tx):         return VOTE
    if is_revocation(tx):   return REVOCATION
    if tx.version >= 3:
        if is_treasury_add(tx):    return TREASURY_ADD
        if is_treasury_spend(tx): return TREASURY_SPEND
        if is_treasurybase(tx):    return TREASURYBASE
    return REGULAR

def classify(tx, treasury_active):
    t = stake_type(tx)
    if t == REGULAR and is_coinbase_form(tx, treasury_active):
        return COINBASE
    return t
```

dcrd: blockchain/stake/staketx.go · DetermineTxType
internal/blockchain/validate.go · checkTransactionContext

The six stake predicates are mutually exclusive, so the order of the tests in `stake_type` does not matter for well-formed input; it is given as dcrd applies it. A transaction that almost, but not exactly, satisfies a stake predicate is classified `REGULAR` and is then subject to the regular rules, which normally reject it (for example a malformed vote has a null outpoint as input 0 and fails Rule 7.19).

IMPLEMENTATION PITFALL
`stake_type` does not consult the `treasury` agenda: a version 3 transaction of treasury-add, treasury-spend or treasurybase form is classified as such even before the agenda is active. Before activation such a transaction is invalid in either tree (Rule 7.4). The coinbase predicate, in contrast, does depend on the agenda, and once `treasury` is active a treasurybase also satisfies `is_coinbase_form`; code that tests for a coinbase in the stake tree must test for the treasurybase first.

Table 7.6 Transaction types, their trees and where their structure is specified

Type	Tree	Placement condition	Structure
Coinbase	regular	Exactly one, at index 0.	Chapter 11
Regular	regular	Any index other than 0.	this chapter
Ticket	stake	Any position (after the treasurybase when `treasury` is active).	Chapter 8
Vote	stake	Only at or above `StakeValidationHeight`.	Chapter 8
Revocation	stake	Only at or above `StakeValidationHeight`.	Chapter 8
Treasurybase	stake	Exactly one, at stake index 0, when and only when `treasury` is active.	Chapter 12
Treasury add	stake	Only when `treasury` is active.	Chapter 12
Treasury spend	stake	Only when `treasury` is active, at or above `StakeValidationHeight`, on a treasury vote interval.	Chapter 12

RULE 7.4 Tree placement of transaction types

1. Every transaction in the regular tree **MUST** have `stake_type` equal to `REGULAR` (`ErrStakeTxInRegularTree`).
2. Every transaction in the stake tree **MUST** have `stake_type` different from `REGULAR` (`ErrRegTxInStakeTree`).
3. If `treasury` is not active, the stake tree **MUST NOT** contain transactions of type `TREASURY_ADD`, `TREASURY_SPEND` or `TREASURYBASE`. (dcrd enforces this by counting only the other three types when the agenda is inactive and requiring the counts to add up to the size of the stake tree, `ErrNonstandardStakeTx`.)

The remaining placement conditions of Table 7.6 are block rules specified in Chapter 10, Chapter 8 and Chapter 12.

dcrd: internal/blockchain/validate.go · checkBlockContext

7.4 Context-Free Sanity Rules

These rules depend on nothing but the transaction and the network parameters. dcrd applies them to every transaction of both trees when it first receives a block, before any agenda state is known, and they are identical for all heights.

RULE 7.5 Inputs and outputs present
A transaction **MUST** have at least one input (`ErrNoTxInputs`) and at least one output (`ErrNoTxOutputs`).

dcrd: blockchain/standalone/tx.go · CheckTransactionSanity

RULE 7.6 Maximum serialized size
The size in bytes of the serialization of the transaction, in the serialization type with which it was decoded, **MUST NOT** exceed `MaxTxSize`. For every transaction that can appear in a valid block this is the full serialization (prefix and witness, Chapter 4), because a transaction of any other serialization

type is always rejected by other rules. `MaxTxSize` is 393216 on mainnet and 1000000 on the other networks (`ErrTxTooBig`). The limit is inclusive: a transaction of exactly `MaxTxSize` bytes is valid.

dcrd: blockchain/standalone/tx.go · CheckTransactionSanity

RULE 7.7 Output amounts

For each output in order, its amount v **MUST** satisfy $0 \leq v \leq \texttt{MaxAmount}$, and the running total of the amounts of that output and all preceding outputs **MUST** satisfy $0 \leq \text{total} \leq \texttt{MaxAmount}$ (`ErrBadTxOutValue`). A zero amount is valid (such an output is never spendable, Rule 7.2).

dcrd: blockchain/standalone/tx.go · CheckTransactionSanity

RULE 7.8 Input witness amounts

For each input, `ValueIn` **MUST** either equal `NullValueIn` (-1) or satisfy $0 \leq \text{ValueIn} \leq \texttt{MaxAmount}$ (`ErrFraudAmountIn`). No constraint is placed on the sum of the `ValueIn` fields here.

dcrd: blockchain/standalone/tx.go · CheckTransactionSanity

RULE 7.9 No duplicate inputs

No two inputs of a transaction may have equal outpoints, where equality compares hash, index and tree (`ErrDuplicateTxInputs`).

dcrd: blockchain/standalone/tx.go · CheckTransactionSanity

NOTE

Because the tree is part of outpoint equality, two inputs that differ only in the tree byte pass Rule 7.9; at most one of them can reference an existing entry, so the transaction still fails input validation. Null outpoints count for the purpose of the duplicate check.

ALGORITHM 7.3 Context-free sanity check

```
MAX_AMOUNT = 2_100_000_000_000_000

def check_sanity(tx, max_tx_size):
    if len(tx.inputs) == 0:  reject("ErrNoTxInputs")
    if len(tx.outputs) == 0: reject("ErrNoTxOutputs")
    if full_serialize_size(tx) > max_tx_size: reject("ErrTxTooBig")
    total = 0
    for out in tx.outputs:
        if out.value < 0 or out.value > MAX_AMOUNT: reject("ErrBadTxOutValue")
        total = total + out.value      # cannot exceed 2*MAX_AMOUNT: no overflow
        if total > MAX_AMOUNT:          reject("ErrBadTxOutValue")
    for txin in tx.inputs:
        if (txin.value_in < 0 and txin.value_in != -1) or txin.value_in > MAX_AMOUNT:
            reject("ErrFraudAmountIn")
    seen = set()
    for txin in tx.inputs:
        key = (txin.prev_out.hash, txin.prev_out.index, txin.prev_out.tree)
        if key in seen: reject("ErrDuplicateTxInputs")
        seen.add(key)
```

dcrd: blockchain/standalone/tx.go · CheckTransactionSanity

TEST VECTOR Sanity checks (results from `standalone.CheckTransactionSanity`)

```
one output of 2100000000000000                    valid
one output of 2100000000000001                    ErrBadTxOutValue
outputs 2100000000000000 and 1                    ErrBadTxOutValue (total)
```

```
one output of 0                                    valid
one output of -1                                   ErrBadTxOutValue
input ValueIn -2                                   ErrFraudAmountIn
input ValueIn 2100000000000000                     valid
input ValueIn 2100000000000001                     ErrFraudAmountIn
inputs (h, 1, tree 0) and (h, 1, tree 1)           valid
inputs (h, 1, tree 0) and (h, 1, tree 0)           ErrDuplicateTxInputs
```

7.5 Contextual Structure Rules

The rules in this section depend on the transaction and on the agenda state for the containing block, but not on the UTXO set. They apply to every transaction of both trees. The type `t` used below is `classify(tx, treasury_active)` (Algorithm 7.2).

7.5.1 Versions

The transaction version is a 16-bit field (Chapter 4). Script versions are 16-bit fields of each output.

RULE 7.10 Maximum transaction version

If `explicitverupgrades` (DCP-0008) is active, the transaction version **MUST** be at most 3. Otherwise every version from 0 to 65535 is permitted by this rule (`ErrTxVersionTooHigh`). This applies to all types in both trees.

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 7.11 Maximum output script version of regular transactions

If `t` is `REGULAR` or `COINBASE` and `explicitverupgrades` is active, every output **MUST** have script version 0 (`ErrScriptVersionTooHigh`). Without the agenda any script version is permitted by this rule. Stake transactions are exempt from this rule because their structural definitions already require version 0 for every output.

dcrd: internal/blockchain/validate.go · checkTransactionContext

RATIONALE

DCP-0008 makes every future change to transaction or script semantics introduce a new version number, so that the version itself proves the enabling agenda is active and old software cannot be tricked into treating new-version outputs as anyone-can-spend. Before the agenda, output scripts with a nonzero version always succeed when executed (Chapter 6), provided the script engine can be created for the spend (Rule 7.43); such outputs created before activation remain in the UTXO set and remain spendable afterwards, because Rule 7.11 constrains only the outputs a transaction creates.

Version-specific semantics elsewhere: sequence locks apply to versions 2 and above (Section 7.8); the coinbase must be version 3 when `treasury` is active (Algorithm 7.1); revocations must be version 2 under `autorevocations` (Rule 7.15); treasury transactions are version 3 by definition (Chapter 12); votes carrying treasury-spend votes are version 3 (Chapter 8).

7.5.2 Regular transaction outputs

RULE 7.12 No stake opcodes in regular outputs

If `t` is `REGULAR` or `COINBASE` , every output script is tokenized as a version 0 script (Chapter 6), whatever its declared script version.

1. If tokenization fails (for example a data push extends past the end of the script), the transaction **MUST** be rejected (`ErrScriptMalformed`).

2. No opcode of the script (data inside pushes is not an opcode) may be one of `OP_SSTX` (`0xba`), `OP_SSGEN` (`0xbb`), `OP_SSRTX` (`0xbc`), `OP_SSTXCHANGE` (`0xbd`); and, if `treasury` is active, also `OP_TADD` (`0xc1`), `OP_TSPEND` (`0xc2`), `OP_TGEN` (`0xc3`) (`ErrRegTxCreateStakeOut`).

dcrd scans as it tokenizes, so a script with a forbidden opcode before a malformed push is reported with the second error kind; the transaction is rejected either way.

dcrd: internal/blockchain/validate.go · checkTransactionContext
txscript/script.go · ContainsStakeOpCodes

IMPLEMENTATION PITFALL
Output scripts with a nonzero script version are nonetheless tokenized with version 0 rules here. Before `explicitverupgrades`, a regular output with script version 1 whose bytes do not tokenize as version 0 makes the whole block invalid.

7.5.3 Input forms by type

RULE 7.13 Coinbase input
If `t` is `COINBASE`:

1. its single input's outpoint **MUST** be the null outpoint, including tree 0 (`ErrBadCoinbaseOutpoint`; `is_coinbase_form` does not examine the tree);
2. the input's `BlockHeight` **MUST** be 0 and `BlockIndex` **MUST** be `ffffffff` (`ErrBadCoinbaseFraudProof`);
3. the length l of its signature script **MUST** satisfy $2 \leq l \leq 100$ (`ErrBadCoinbaseScriptLen`).

The coinbase's `ValueIn`, outputs and height commitment are specified in Chapter 11.

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 7.14 Vote stakebase and ticket input
If `t` is `VOTE`:

1. the length l of the signature script of input 0 (the stakebase) **MUST** satisfy $2 \leq l \leq 100$ (`ErrBadStakebaseScriptLen`);
2. the signature script of input 0 **MUST** equal `StakeBaseSigScript` byte for byte (`ErrBadStakebaseScrVal`);
3. the outpoint of input 1 **MUST NOT** be the null outpoint (`ErrBadTxInput`).

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 7.15 Revocation version and input
If `t` is `REVOCATION`:

1. if `autorevocations` (DCP-0009) is active, the transaction version **MUST** equal 2 (`ErrInvalidRevocationTxVersion`);
2. the outpoint of input 0 **MUST NOT** be the null outpoint (`ErrBadTxInput`).

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 7.16 Treasurybase input
If `t` is `TREASURYBASE`: input 0 **MUST** have the null outpoint (`ErrBadTreasurybaseOutpoint`), `BlockHeight` 0 and `BlockIndex` `ffffffff` (`ErrBadTreasurybaseFraudProof`), and an empty signature script (`ErrBadTreasurybaseScriptLen`). The first and third conditions are already implied by `is_treasurybase`.

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 7.17 Treasury spend input

If `t` is `TREASURY_SPEND`: input 0 **MUST** have the null outpoint (`ErrBadTSpendOutpoint`), `BlockHeight` 0 and `BlockIndex` `ffffffff` (`ErrBadTSpendFraudProof`), and a signature script of exactly 100 bytes (`ErrBadTSpendScriptLen`, already implied by `is_treasury_spend`). Note that `is_treasury_spend` itself does not require the null outpoint.

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 7.18 Treasury add change

If `t` is `TREASURY_ADD` and it has exactly two outputs, the amount of output 1 (the change) **MUST NOT** be 0 (`ErrInvalidTAddChange`). Treasury adds are additionally subject to Rule 7.19.

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 7.19 Non-null inputs

If `t` is `REGULAR`, `TICKET` or `TREASURY_ADD`, no input may have the null outpoint (`ErrBadTxInput`).

dcrd: internal/blockchain/validate.go · checkTransactionContext

NOTE

The coinbase, treasurybase and treasury spend each have exactly one input by their definitions, so the rules above constrain all of their inputs. A vote has exactly two inputs and a revocation exactly one (Chapter 8).

ALGORITHM 7.4 Contextual structure check

```
def check_context(tx, params, treasury, explicit_upgrades, auto_revocations):
    if explicit_upgrades and tx.version > 3: reject("ErrTxVersionTooHigh")
    t = classify(tx, treasury)
    i0 = tx.inputs[0]
    if t == VOTE:
        if not 2 <= len(i0.sig_script) <= 100: reject("ErrBadStakebaseScriptLen")
        if i0.sig_script != params.stake_base_sig_script:
            reject("ErrBadStakebaseScrVal")
        if is_null_outpoint(tx.inputs[1].prev_out):  reject("ErrBadTxInput")
    elif t == REVOCATION:
        if auto_revocations and tx.version != 2: reject("ErrInvalidRevocationTxVersion")
        if is_null_outpoint(i0.prev_out): reject("ErrBadTxInput")
    elif t == COINBASE:
        if not is_null_outpoint(i0.prev_out): reject("ErrBadCoinbaseOutpoint")
        if not is_null_witness(i0):           reject("ErrBadCoinbaseFraudProof")
        if not 2 <= len(i0.sig_script) <= 100: reject("ErrBadCoinbaseScriptLen")
    elif t == TREASURYBASE:
        if not is_null_outpoint(i0.prev_out): reject("ErrBadTreasurybaseOutpoint")
        if not is_null_witness(i0):           reject("ErrBadTreasurybaseFraudProof")
        if len(i0.sig_script) != 0:           reject("ErrBadTreasurybaseScriptLen")
    elif t == TREASURY_SPEND:
        if not is_null_outpoint(i0.prev_out): reject("ErrBadTSpendOutpoint")
        if not is_null_witness(i0):           reject("ErrBadTSpendFraudProof")
        if len(i0.sig_script) != 100:         reject("ErrBadTSpendScriptLen")
    else:                                     # REGULAR, TICKET, TREASURY_ADD
        if t == TREASURY_ADD and len(tx.outputs) == 2 and tx.outputs[1].value == 0:
            reject("ErrInvalidTAddChange")
        for txin in tx.inputs:
            if is_null_outpoint(txin.prev_out): reject("ErrBadTxInput")
    if t in (REGULAR, COINBASE):
        for out in tx.outputs:
            if explicit_upgrades and out.version > 0: reject("ErrScriptVersionTooHigh")
            ops = tokenize_v0(out.pk_script)  # reject("ErrScriptMalformed") on failure
            if any(is_stake_opcode(op, treasury) for op in ops):
                reject("ErrRegTxCreateStakeOut")
```

```
def is_null_outpoint(op):
    return op.hash == ZERO_HASH and op.index == 0xffffffff and op.tree == 0

def is_null_witness(txin):          # ValueIn is not part of this test
    return txin.block_height == 0 and txin.block_index == 0xffffffff
```

dcrd: internal/blockchain/validate.go · checkTransactionContext

7.6 Expiry

A transaction may name a height from which it can no longer be mined. This lets a transaction that fails to confirm in time become permanently invalid instead of lingering. The `Expiry` field (`uint32`) of value 0 means no expiry.

RULE 7.20 Expired transactions

Let h be the height of the block containing the transaction and E its `Expiry`. If $E \neq 0$ and $h \geq E$, the transaction is expired and the block **MUST** be rejected (`ErrExpiredTx`). Equivalently, a transaction with $E \neq 0$ is valid only in blocks of height at most $E - 1$. This applies to every transaction in both trees, the coinbase included.

dcrd: internal/blockchain/validate.go · IsExpiredTx
internal/blockchain/validate.go · checkBlockDataPositional

Expiry has one further consensus effect: outputs of a transaction with a nonzero expiry require `CoinbaseMaturity` confirmations before they can be spent (Rule 7.30). Treasury spends have additional expiry requirements tied to the treasury voting windows, specified in Chapter 12.

TEST VECTOR Expiry (results from `IsExpiredTx`)

```
Expiry 0,    height 1000000   not expired
Expiry 1000, height 999       not expired
Expiry 1000, height 1000      expired
Expiry 1000, height 1001      expired
```

7.7 Lock Time Finality

The `LockTime` field (`uint32`) sets the earliest point at which a transaction may be mined, either as a block height or as a Unix time. A transaction that may be mined is **final**. As in Bitcoin, setting every input's sequence number to its maximum value `ffffffff` disables the lock time.

RULE 7.21 Lock time finality

Every transaction in both trees of a block at height h **MUST** be final, otherwise the block is invalid (`ErrUnfinalizedTx`). Let L be the transaction's `LockTime`. The transaction is final if and only if at least one of the following holds:

1. $L = 0$;
2. $L < 500000000$ (`LockTimeThreshold`) and $L < h$;
3. $L \geq 500000000$ and $L < \tau$, where τ is the **lock-time reference time** defined below;
4. every input has sequence number `ffffffff`.

All comparisons are strict as written. The reference time τ is:

- if `lnfeatures` is **not** active: the timestamp field of the header of the block being validated;

- if `lnfeatures` is active: the past median time of the **parent** of the block being validated (the median of the timestamps of the 11 blocks ending with the parent, or of all blocks back to the genesis block if there are fewer, as defined in Chapter 10).

dcrd: internal/blockchain/validate.go · IsFinalizedTransaction
internal/blockchain/validate.go · checkBlockContext

HISTORY
DCP-0003 changed the reference time from the block's own timestamp to the parent's past median time, because the median time never decreases while miner-chosen timestamps need not. On mainnet `lnfeatures` became active at block 189568.

IMPLEMENTATION PITFALL
An input sequence number of `fffffffe` disables relative locks for that input (its bit 31 is set, Section 7.8) but does **not** count toward condition 4. Condition 4 is evaluated on all inputs, including the null inputs of a coinbase or stakebase.

TEST VECTOR Finality (results from `IsFinalizedTransaction`; h = block height, τ = reference time)

LockTime	sequences	h	tau	final
0	00000000	100	1600000000	yes
99	00000000	100	1600000000	yes
100	00000000	100	1600000000	no
101	00000000	100	1600000000	no
100	ffffffff	100	1600000000	yes
100	ffffffff fffffffe	100	1600000000	no
499999999	00000000	499999999	1600000000	no
499999999	00000000	500000000	1	yes
500000000	00000000	999999999	500000000	no
500000000	00000000	1	500000001	yes
1600000000	00000000	1	1599999999	no
1600000000	00000000	1	1600000000	no
1600000000	00000000	1	1600000001	yes

7.8 Relative Lock Times (Sequence Locks)

DCP-0003 repurposes the 32-bit sequence number of each input to express a **relative** lock: a minimum age of the spent output, measured in blocks or in units of 512 seconds from the block that created the output. The script opcode `OP_CHECKSEQUENCEVERIFY` lets an output script require such a lock (Chapter 6); the consensus rule in this section enforces it.

7.8.1 Encoding

Table 7.7 Interpretation of an input sequence number under sequence locks

Bits	Mask	Meaning
31	`80000000`	Disable flag (`SequenceLockTimeDisabled`). If set, the input has no relative lock.
30–23	—	Ignored.
22	`00400000`	Type flag (`SequenceLockTimeIsSeconds`). Set: the value counts units of $2^9 = 512$ seconds. Clear: the value counts blocks.
21–16	—	Ignored.
15–0	`0000ffff`	Value v (`SequenceLockTimeMask`), $0 \le v \le 65535$.

The maximum relative lock is therefore 65535 blocks or $65535 \times 512 = 33553920$ seconds.

IMPLEMENTATION PITFALL
The text of DCP-0003 states the type flag the other way round ("when set … blocks"). The reference implementation, and therefore consensus, treats a **set** bit 22 as time-based, as in Table 7.7; the DCP's own reference code agrees with this table.

7.8.2 Applicability

RULE 7.22 Transactions subject to sequence locks
Sequence locks are enforced only when `lnfeatures` is active. When it is, a transaction is subject to them if and only if all of the following hold:

1. its version is at least 2 (any value from 2 to 65535);
2. it is not of coinbase form, `is_coinbase_form(tx, treasury_active)` (this excludes the coinbase and, when `treasury` is active, the treasurybase);
3. it is not a vote.

This includes regular transactions other than the coinbase, tickets, revocations, treasury adds and treasury spends. Within a subject transaction, an input participates if and only if bit 31 of its sequence number is clear.

dcrd: internal/blockchain/sequencelock.go · calcSequenceLock
internal/blockchain/validate.go · checkConnectBlock

RULE 7.23 Treasury spend sequence number
When `lnfeatures` and `treasury` are both active, the input of a treasury spend **MUST** have bit 31 of its sequence number set. If it is clear, the sequence-lock calculation looks up the null outpoint, which never exists, and the block is rejected (`ErrMissingTxOut`).

dcrd: internal/blockchain/sequencelock.go · calcSequenceLock

7.8.3 Calculation and enforcement

For each participating input, the lock is computed from the UTXO entry it spends: its height h_o and, for time locks, the past median time of the block **preceding** the one that contains the output. Each transaction gets one pair (MinHeight, MinTime), the maxima over its inputs, with -1 meaning "no constraint".

ALGORITHM 7.5 Sequence lock of a transaction

```
def sequence_lock(tx, block, view, treasury_active):
    # block: the block being validated (height block.height).
    min_height, min_time = -1, -1                       # int64
    if tx.version < 2 or is_coinbase_form(tx, treasury_active) or is_vote(tx):
        return (min_height, min_time)
    for txin in tx.inputs:
        seq = txin.sequence                             # uint32
        if seq & 0x80000000:
            continue
        entry = view.lookup_including_spent(txin.prev_out)
        if entry is None:
            reject("ErrMissingTxOut")
        h_o = entry.height
        value = seq & 0x0000ffff
        if seq & 0x00400000:                            # time based
            prev_h = max(h_o - 1, 0)
            mtp = past_median_time(ancestor(block, prev_h))   # blocks chapter
            t = mtp + (value << 9) - 1
            min_time = max(min_time, t)
        else:                                           # height based
```

```
            min_height = max(min_height, h_o + value - 1)
    return (min_height, min_time)
```

dcrd: internal/blockchain/sequencelock.go · calcSequenceLock

`lookup_including_spent` returns the entry whether or not it has been marked spent: the calculation runs after the transaction's tree has been connected (Section 7.10), so the entries it reads have already been spent by the transaction itself. `ancestor(block, n)` is the block at height n on the chain ending with `block`; for n equal to the parent's height it is the parent.

RULE 7.24 Sequence lock enforcement

When `lnfeatures` is active, let h be the height of the block being validated and m the past median time of its parent. For every transaction subject to sequence locks in either tree, with lock (MinHeight, MinTime) from Algorithm 7.5, the block **MUST** be rejected (`ErrUnfinalizedTx`) if $h \leq$ MinHeight or $m \leq$ MinTime. Equivalently, each block-based input requires $h \geq h_o + v$ and each time-based input requires $m \geq \mathrm{mtp}_o + 512v$.

dcrd: internal/blockchain/validate.go · SequenceLockActive
internal/blockchain/validate.go · checkConnectBlock

NOTE

A relative lock of $v = 0$ is always satisfied. A regular transaction spending an output created earlier in the same block ($h_o = h$) can carry only $v = 0$ locks: a block lock of 1 gives MinHeight $= h$, and a time lock of $v \geq 1$ gives MinTime $\geq m$ because the preceding block is the parent. The value `0x7fffffff` that dcrd's sequence-lock code treats as an "unconfirmed" height is used only by mempool policy and never occurs in block validation.

TEST VECTOR Sequence locks (from `calcSequenceLock` and `SequenceLockActive`)

Chain of regnet blocks with timestamps $1600000000 + 512h$ for $h \geq 1$. Past median times: block 19 → 1600007168, block 24 (the parent) → 1600009728. Output A is created at height 20 and output B at height 25; the transaction is validated in the block at height 25.

version	inputs (sequence)	MinHeight	MinTime	satisfied
1	A:00000005	-1	-1	yes
2	A:80000005	-1	-1	yes
2	A:00000000	19	-1	yes
2	A:00000005	24	-1	yes
2	A:00000006	25	-1	no
2	A:7f3f0005 (reserved bits set)	24	-1	yes
2	A:00400005	-1	1600009727	yes
2	A:00400006	-1	1600010239	no
2	A:00400000	-1	1600007167	yes
2	B:00000000	24	-1	yes
2	B:00000001	25	-1	no
2	A:00400005 B:00000000 A:80000000	24	1600009727	yes
65535	A:00000005	24	-1	yes

For the time locks: $1600007168 + 5 \times 512 - 1 = 1600009727 < 1600009728$, so the lock with $v = 5$ is satisfied, and the lock with $v = 6$ (MinTime $= 1600010239$) is not.

7.8.4 The DCP-0004 correction

HISTORY

From the activation of `lnfeatures` (mainnet block 189568) dcrd in effect evaluated sequence locks of regular-tree transactions against a restricted view of the outputs, a by-product of the transaction-indexed UTXO storage of the time (from January 2019 reproduced by an explicitly constructed "legacy" view). That view

contained only: outputs of transactions of the parent block's regular tree and outputs of transactions referenced by that tree's inputs (both only if the block approved its parent), and outputs of transactions referenced by, or contained in, the current block's stake tree, together with whatever the node's working view happened to contain. As a result, a version 2 transaction with an enabled relative lock was rejected (with `ErrMissingTxOut`) when it spent an output of an earlier transaction in the same block, an output from a block before the parent whose transaction was not otherwise referenced, or an output of a transaction shared with a disapproved parent.

DCP-0004 [32] (agenda `fixlnseqlocks`, mainnet activation at block 342784) replaced this with the rule of this section: the view is exactly the set of outputs that exist from the point of view of the block being validated. The legacy behaviour also depended on the node's prior history (for example on the contents of the working view after a reorganization) and so cannot be specified as a pure function of the chain.

dcrd no longer implements the legacy view: at the commit this document describes, Rule 7.24 with Algorithm 7.5 applies to every block for which `lnfeatures` is active, and `fixlnseqlocks` is not consulted. Because the corrected rule is strictly less restrictive, every block of the historical chains also satisfies it, as DCP-0004 anticipates ("new implementations will not require any additional code"). An implementation that follows this document therefore matches the reference implementation exactly.

7.9 Input Validation

Input validation checks every input of a transaction against the UTXO set that exists when the transaction is connected (Section 7.10), enforces maturity and tag restrictions on what is spent, sums the input amounts and computes the fee. Several transaction types have additional input rules specified in other chapters; Table 7.10 says where each one plugs in.

7.9.1 Inputs without a referenced output

RULE 7.25 Inputs exempt from UTXO lookup

The following inputs do not reference a UTXO entry and are exempt from Rule 7.26 through Rule 7.36:

1. the input of the coinbase (the coinbase is exempt from input validation altogether, and its fee is 0 by definition);
2. input 0 (the stakebase) of a vote;
3. the input of the treasurybase, when `treasury` is active;
4. the input of a treasury spend, when `treasury` is active.

For items 2 to 4 the input's `ValueIn` is added to the input total (Rule 7.37). Its correctness is a rule of the chapter that owns the type (Table 7.10).

dcrd: internal/blockchain/validate.go · CheckTransactionInputs

7.9.2 Per-input rules

In the following rules, h is the height of the block containing the transaction, e the UTXO entry referenced by the input, and age $= h - e$.height. A script is **tagged with** opcode o when its script version is 0, its first byte is o, and the remaining bytes are exactly a pay-to-pubkey-hash script (25 bytes: `OP_DUP OP_HASH160 OP_DATA_20 <20 bytes> OP_EQUALVERIFY OP_CHECKSIG`) or a pay-to-script-hash script (23 bytes: `OP_HASH160 OP_DATA_20 <20 bytes> OP_EQUAL`).

RULE 7.26 Referenced output exists and is unspent

The outpoint of each input **MUST** identify an entry of the UTXO set that is unspent at the moment the transaction is connected (`ErrMissingTxOut`). The set is that of Section 7.10: it reflects every transaction connected before this one, including earlier transactions of the same block.

dcrd: internal/blockchain/validate.go · CheckTransactionInputs

RULE 7.27 Stake transactions do not spend same-block outputs
If the transaction is of any stake type and e.height $= h$, the transaction **MUST** be rejected (`ErrMissingTxOut`), unless the input is one of the historical exceptions of Table 7.12.
dcrd: internal/blockchain/validate.go · CheckTransactionInputs

RULE 7.28 Zero-value outputs are not spendable
If e.amount $= 0$ the transaction **MUST** be rejected (`ErrZeroValueOutputSpend`). (Such entries cannot exist by Rule 7.2; the check is defensive.)
dcrd: internal/blockchain/validate.go · CheckTransactionInputs

The witness rule Rule 7.3 applies at this point.

RULE 7.29 Coinbase maturity
If e has the coinbase flag, age **MUST** be at least `CoinbaseMaturity` (256 on mainnet) (`ErrImmatureSpend`). An output of the coinbase in block b is thus first spendable in block $b +$ `CoinbaseMaturity`.
dcrd: internal/blockchain/validate.go · CheckTransactionInputs

RULE 7.30 Maturity of outputs of expiring transactions
If e has the expiry flag (its creating transaction had a nonzero `Expiry`), age **MUST** be at least `CoinbaseMaturity` (`ErrExpiryTxSpentEarly`).
dcrd: internal/blockchain/validate.go · CheckTransactionInputs

RATIONALE
A transaction with an expiry can become invalid if a reorganization moves it past its expiry height. Requiring the same maturity as the coinbase limits the damage to descendants of such a transaction, just as coinbase maturity does for coinbases invalidated by reorganizations.

RULE 7.31 Treasury generation output maturity
If `treasury` is active and e's script is tagged with `OP_TGEN` (`0xc3`), age **MUST** be at least `CoinbaseMaturity` (`ErrImmatureSpend`).
dcrd: internal/blockchain/validate.go · CheckTransactionInputs

RULE 7.32 Ticket outputs are spent only by votes and revocations
If e's script is tagged with `OP_SSTX` (`0xba`), the spending transaction **MUST** be a vote or a revocation (`ErrTxSStxOutSpend`).
dcrd: internal/blockchain/validate.go · CheckTransactionInputs

RULE 7.33 Treasury add outputs are unspendable
If `treasury` is active, e's script version is 0 and its script is the single byte `0xc1` (`OP_TADD`), the transaction **MUST** be rejected (`ErrBadTxInput`).
dcrd: internal/blockchain/validate.go · CheckTransactionInputs

NOTE
Treasury add and treasurybase `OP_TADD` outputs never enter the UTXO set (Rule 7.2). Rule 7.33 matters for regular outputs consisting of the single byte `0xc1` created before `treasury` activated, when `0xc1` was not a stake opcode: they become permanently unspendable on activation.

RULE 7.34 Vote and revocation output maturity
If e's script is tagged with `OP_SSGEN` (`0xbb`) or `OP_SSRTX` (`0xbc`), age **MUST** be at least M (`ErrImmatureSpend`), where $M =$ `CoinbaseMaturity` if `treasury` is active and $M =$ `SStxChangeMaturity` (1 on every network) otherwise.
dcrd: internal/blockchain/validate.go · CheckTransactionInputs

HISTORY
Before the `treasury` agenda (DCP-0006), outputs of votes and revocations required only `SStxChangeMaturity` (1) confirmation, although dcrd's comments have long described them as subject to coinbase maturity. The agenda raised the requirement to `CoinbaseMaturity`. Both forms are normative; the selection is by the agenda state of the block containing the **spending** transaction.

RULE 7.35 Stake change maturity
If e's script is tagged with `OP_SSTXCHANGE` (`0xbd`), age **MUST** be at least `SStxChangeMaturity` (`ErrImmatureSpend`). This covers ticket change outputs and treasury add change outputs.
dcrd: internal/blockchain/validate.go · CheckTransactionInputs

RULE 7.36 Referenced amount range
e.amount **MUST** satisfy $0 \leq e.\text{amount} \leq$ `MaxAmount` (`ErrBadTxOutValue`). (This cannot fail for entries created under Rule 7.7.)
dcrd: internal/blockchain/validate.go · CheckTransactionInputs

All maturity rules are independent: an input may be subject to several (for example a ticket change output of a ticket with a nonzero expiry needs both 1 and `CoinbaseMaturity` confirmations). Every maturity comparison is "age $\geq M$", with age computed from the height of the block containing the spending transaction.

Table 7.8 All maturity requirements on spent outputs

Spent output	Required age	Rule
Coinbase output	`CoinbaseMaturity`	Rule 7.29
Any output of a transaction with nonzero expiry	`CoinbaseMaturity`	Rule 7.30
`OP_TGEN`-tagged (treasury spend payouts), `treasury` active	`CoinbaseMaturity`	Rule 7.31
`OP_SSGEN` / `OP_SSRTX`-tagged, `treasury` active	`CoinbaseMaturity`	Rule 7.34
`OP_SSGEN` / `OP_SSRTX`-tagged, `treasury` inactive	`SStxChangeMaturity`	Rule 7.34
`OP_SSTXCHANGE`-tagged	`SStxChangeMaturity`	Rule 7.35
Ticket output 0 spent by a vote	`TicketMaturity` $+1$	Chapter 8
Ticket output 0 spent by a revocation	`TicketMaturity` $+2$, or $+1$ with `autorevocations`	Chapter 8
Treasury balance	`CoinbaseMaturity`	Chapter 12

7.9.3 Spending across trees

Taken together, the rules above and the structural rules of the stake transactions determine which outputs each kind of transaction may spend (Table 7.9). Apart from the restrictions listed, an output of either tree may be spent by a transaction of either tree.

Table 7.9 Which spenders may consume which kinds of outputs (outside the same block)

Output kind	Who may spend it	Rules
`OP_SSTX`-tagged (ticket output 0)	Only a vote (as its input 1) or a revocation (as its input 0).	Rule 7.32, Chapter 8

Output kind	Who may spend it	Rules
`OP_SSTXCHANGE`-tagged (ticket and treasury add change)	Any transaction after `SStxChangeMaturity`.	Rule 7.35
`OP_SSGEN` / `OP_SSRTX`-tagged (vote and revocation payouts)	Any transaction after the maturity of Rule 7.34.	Rule 7.34
`OP_TGEN`-tagged (treasury spend payouts)	Any transaction after `CoinbaseMaturity`.	Rule 7.31
Single-byte `OP_TADD` script	Nobody, once `treasury` is active.	Rule 7.2, Rule 7.33
Coinbase outputs	Any transaction after `CoinbaseMaturity`.	Rule 7.29
Other outputs	Any transaction, subject to the input rules of the spender's type.	—

Votes and revocations have no inputs other than the ticket (and, for votes, the stakebase); treasury spends and treasurybases have no real inputs. Tickets and treasury adds are therefore the only stake transactions that spend ordinary outputs. Tickets are further limited to version 0 outputs that are plain P2PKH or P2SH, or P2PKH or P2SH tagged with `OP_SSGEN`, `OP_SSRTX` or `OP_SSTXCHANGE` (so a ticket cannot spend an `OP_TGEN` output directly; Chapter 8). Treasury adds may spend any output permitted by the rules of this chapter.

7.9.4 Value and fee

RULE 7.37 Input total

The input total is accumulated in a signed 64-bit integer, in input order, from the `ValueIn` of each input of Rule 7.25 items 2–4 and the entry amount e.amount of each other input. After each addition the accumulation **MUST NOT** have overflowed (signed overflow) and the running total **MUST NOT** exceed `MaxAmount` (`ErrBadTxOutValue` in both cases).

dcrd: internal/blockchain/validate.go · CheckTransactionInputs

RULE 7.38 Outputs do not exceed inputs; fee

Let I be the input total and O the sum of the output amounts (bounded by Rule 7.7). The transaction **MUST** satisfy $I \geq O$ (`ErrSpendTooHigh`). Its **fee** is $I - O$. There is no minimum fee in consensus. The coinbase's fee is defined as 0. The block-level use of fees (the coinbase may claim them) is specified in Chapter 11.

dcrd: internal/blockchain/validate.go · CheckTransactionInputs

7.9.5 Delegated input rules

Table 7.10 Type-specific input rules specified in other chapters, in the order dcrd applies them before the per-input rules

Type	Rules	Chapter
Coinbase	`ValueIn` equals the subsidy; output total bounded by subsidy plus fees; block-one ledger.	Chapter 11
Ticket	Every input spends an existing unspent version 0 P2PKH or P2SH output (plain, or tagged with `OP_SSGEN`, `OP_SSRTX` or `OP_SSTXCHANGE`); each commitment amount plus change equals its input amount (`checkTicketPurchaseInputs`). Ticket price versus header: Chapter 10.	Chapter 8
Vote	Stakebase `ValueIn` equals the vote subsidy; input 1 spends output 0 of an unspent, mature ticket; payouts match the ticket's commitments (`checkVoteInputs`); the ticket is a winner (block rule).	Chapter 8
Revocation	Input 0 spends output 0 of an unspent, mature ticket eligible for revocation; payouts match the commitments (`checkRevocationInputs`).	Chapter 8

Type	Rules	Chapter
Treasury spend	Public key is a sanctioned Pi key (`ErrUnknownPiKey`); signature valid (`ErrInvalidPiSignature`); `ValueIn` in range and equal to the output 0 commitment (`checkTreasurySpendInputs`); voting, window and expenditure checks.	Chapter 12
Treasurybase	`ValueIn` equals the treasury subsidy; output structure.	Chapter 12, Chapter 11

7.9.6 Algorithm

ALGORITHM 7.6 **Input validation and fee**

```
COINBASE_MATURITY = params.coinbase_maturity
SSTX_CHANGE_MATURITY = params.sstx_change_maturity

def check_inputs(tx, h, view, treasury, auto_revocations, params) -> int:
    t = classify(tx, treasury)          # treasurybase tested before coinbase form
    if t == COINBASE:
        return 0                        # see subsidy chapter
    if t == TICKET:         staking.check_ticket_purchase_inputs(tx, view)
    if t == VOTE:           staking.check_vote_inputs(tx, h, view, ...)
    if t == REVOCATION:     staking.check_revocation_inputs(tx, h, view, ...)
    if t == TREASURY_SPEND: treasury.check_tspend_key_signature_and_value(tx)
    stake_out_maturity = COINBASE_MATURITY if treasury else SSTX_CHANGE_MATURITY
    total_in = 0                                        # int64
    for idx, txin in enumerate(tx.inputs):
        if idx == 0 and t in (VOTE, TREASURYBASE, TREASURY_SPEND):
            total_in = add_amount(total_in, txin.value_in)
            continue
        e = view.lookup(txin.prev_out)
        if e is None or e.spent:            reject("ErrMissingTxOut")
        if t in STAKE_TYPES and e.height == h \
                and not is_sbss_exception(params.net, h, txin.prev_out.hash):
            reject("ErrMissingTxOut")
        if e.amount == 0:                   reject("ErrZeroValueOutputSpend")
        if txin.value_in != e.amount:       reject("ErrFraudAmountIn")
        if txin.block_height != e.height:   reject("ErrFraudBlockHeight")
        if txin.block_index != e.index:     reject("ErrFraudBlockIndex")
        age = h - e.height
        if e.is_coinbase and age < COINBASE_MATURITY: reject("ErrImmatureSpend")
        if e.has_expiry and age < COINBASE_MATURITY:  reject("ErrExpiryTxSpentEarly")
        if treasury and tagged(e, 0xc3) and age < COINBASE_MATURITY:
            reject("ErrImmatureSpend")
        if t not in (VOTE, REVOCATION) and tagged(e, 0xba):
            reject("ErrTxSStxOutSpend")
        if treasury and e.script_version == 0 and e.pk_script == b"\xc1":
            reject("ErrBadTxInput")
        if (tagged(e, 0xbb) or tagged(e, 0xbc)) and age < stake_out_maturity:
            reject("ErrImmatureSpend")
        if tagged(e, 0xbd) and age < SSTX_CHANGE_MATURITY:
            reject("ErrImmatureSpend")
        if e.amount < 0 or e.amount > MAX_AMOUNT:
            reject("ErrBadTxOutValue")
        total_in = add_amount(total_in, e.amount)
    total_out = sum(out.value for out in tx.outputs)
    if total_in < total_out:
        reject("ErrSpendTooHigh")
    return total_in - total_out

def add_amount(total: int64, amount: int64) -> int64:
    s = int64(total + amount)                       # two's complement wrap
    if (s > total) != (amount > 0):  reject("ErrBadTxOutValue")   # overflow
    if s > MAX_AMOUNT:               reject("ErrBadTxOutValue")
```

```
    return s

def tagged(e, opcode) -> bool:
    s = e.pk_script
    return (e.script_version == 0 and len(s) >= 1 and s[0] == opcode
            and (is_p2pkh(s[1:]) or is_p2sh(s[1:])))
```

dcrd: internal/blockchain/validate.go · CheckTransactionInputs

7.10 Connecting Transactions and In-Block Spending

After a transaction passes input validation it is **connected**: the entries it spends are marked spent and entries for its outputs are created (Rule 7.2). The UTXO set against which a transaction is validated is the set as it stands at that moment, so the order of connection determines which outputs are available. The block-level orchestration is specified in Chapter 10; the consequences for a single transaction are stated here.

RULE 7.39 Connection order and output availability

For a block B at height h, transactions are validated and connected in this order, starting from the UTXO set as of B's parent:

1. if $h > 1$ and B's header disapproves its parent's regular tree, the effects of that regular tree are undone first: its outputs are removed and the outputs it spent are restored (Chapter 10);
2. the transactions of B's stake tree, in order;
3. the transactions of B's regular tree, in order.

Each transaction is validated (Algorithm 7.6) against the set produced by all preceding steps and is then connected before the next is validated. Consequently:

- an input **MAY** reference an output of an **earlier** transaction of the same tree or of a tree connected before, subject to all other rules;
- an input referencing an output of the same transaction or of a **later** transaction in the block references a nonexistent entry and **MUST** be rejected;
- two inputs anywhere in the block that spend the same output are a double spend: the second one to be connected references a spent entry and **MUST** be rejected.

dcrd: internal/blockchain/validate.go · checkConnectBlock, checkTransactionsAndConnect
internal/blockchain/utxoviewpoint.go · connectBlock

The combination of Rule 7.39 with the maturity and same-block rules yields the following availability matrix.

Table 7.11 May a transaction spend an output created in the same block?

Spender	Creator	Verdict
Regular	Earlier regular transaction (not the coinbase)	Yes, unless the creator has a nonzero expiry (Rule 7.30).
Regular	Coinbase	No: coinbase maturity.
Regular	Same or later regular transaction	No: entry does not yet exist.
Regular	Stake transaction of the same block	The entry exists, but every spendable stake output is tagged and subject to a maturity of at least `SStxChangeMaturity` = 1 (`OP_SSTX` outputs are not spendable by regular transactions at all), so: no.
Stake	Any transaction of the same block	No (Rule 7.27), except Table 7.12.

RATIONALE

The stake tree is connected before the regular tree, and may not use outputs of the same block, because the next block's votes may disapprove this block's regular tree while its stake tree remains valid regardless. A stake transaction depending on the regular tree could otherwise be left spending outputs that no longer exist.

IMPLEMENTATION PITFALL
Disapproval is applied before the current block's stake tree is validated. Neither tree of a block that disapproves its parent can spend outputs created by the parent's regular tree, and outputs that the parent's regular transactions had spent are available again to both trees of the block.

7.10.1 Historical same-block stake spends

Rules enforced by earlier software versions allowed stake transactions in a handful of blocks to spend output 0 of a regular transaction of the same block. These blocks are part of the main and test chains and must remain valid.

RULE 7.40 Historical same-block stake spend exceptions
On the networks and at the heights listed in Table 7.12, if the regular tree of the block at that height contains a transaction with the listed hash, then before the stake tree of that block is connected (and after any disapproval of the parent is applied), an entry for output 0 of that transaction is added to the UTXO set as if that transaction had been connected (height h, index equal to its regular-tree position, subject to Rule 7.2). Rule 7.27 does not apply to inputs of that block that reference an output of the listed transaction (only output 0 is made available in this way). When the regular tree is later connected, the listed transaction's outputs are created as usual, which sets output 0 unspent again even if a stake transaction spent it.

dcrd: `internal/blockchain/validate.go · isSBSSViolation, CheckTransactionInputs`
`internal/blockchain/utxoviewpoint.go · addSBSSViolations`

Table 7.12 Historical same-block stake spend exceptions (transaction hashes in display form)

Network	Height	Regular transaction whose output 0 may be spent
mainnet	1106817	`0875ecdc5f12c8aa6ee0d54331fd5b5e786639882a3adb09882de5e3b939613b`
mainnet	1106831	`714e71358937cd33480fd2900853d43ea3440ca67b38bacdfeaeb0c37c80ee27`
mainnet	1106837	`aeba09a4efa334e3c13e68f27f40bd37cda7643c263f88d69f38a33c2cdacd18`
mainnet	1106854	`17228266d9d6dd7084aa2e479319aca6d755c2a9d2dbb3cf28c8151470c7a439`
mainnet	1106860	`20d6d3ed1b609f03969474dca1f452c25e035766457dfd58061f9b9270fc4e6d`
mainnet	1107195	`5d2e1898fe0c631cb795b1906fb3cf8d9772f1826be8dea64dd3ab85ac0ab2d3`
testnet3	1980161	`451b6eed1a777bc5bbb5c1dbe10a9e444cbeb7d863adb75fc6b7a676b3dcbb58`

IMPLEMENTATION PITFALL
The exception is keyed on the network, the height of the block being validated and the hash of the referenced transaction; it applies on side chains at those heights as well. After such a block, output 0 of the listed transaction is an unspent entry of the UTXO set, even though a stake transaction in the block consumed it (unless a regular transaction later in the same block spends it again). An implementation that marks it spent diverges from the reference UTXO set.

7.11 Signature Operations

Blocks limit the number of signature-checking operations they contain (Chapter 10). The limit is defined in terms of two per-transaction counts built on the script-level counting functions of Chapter 6: `sigop_count`, which counts `OP_CHECKSIG`-class opcodes in one script with the quick, imprecise method, and `precise_p2sh_sigop_count`, which, given a signature script and the P2SH output script it spends, counts the opcodes of the redeem script.

ALGORITHM 7.7 Per-transaction signature operation counts

```
def basic_sigops(tx, is_coinbase, is_vote, treasury) -> uint32:
    if treasury and is_treasurybase(tx):
        return 0
    inputs = [] if is_coinbase else (tx.inputs[1:] if is_vote else tx.inputs)
    n = 0
    for txin in inputs:
        n = checked_add_u32(n, sigop_count(txin.sig_script, treasury))
    for out in tx.outputs:
        n = checked_add_u32(n, sigop_count(out.pk_script, treasury))
    return n

def p2sh_sigops(tx, is_coinbase, is_vote, view, treasury) -> uint32:
    if is_coinbase:
        return 0
    if treasury and (is_treasury_spend(tx) or is_treasurybase(tx)):
        return 0
    n = 0
    for txin in (tx.inputs[1:] if is_vote else tx.inputs):
        e = view.lookup(txin.prev_out)
        if e is None or e.spent: reject("ErrMissingTxOut")
        if not is_p2sh(e.pk_script):          # untagged 23-byte form only
            continue
        n = checked_add_u32(n, precise_p2sh_sigop_count(txin.sig_script,
                                                        e.pk_script, treasury))
    return n

def total_sigops(tx, is_coinbase, is_vote, view, treasury) -> uint32:
    return checked_add_u32(basic_sigops(tx, is_coinbase, is_vote, treasury),
                           p2sh_sigops(tx, is_coinbase, is_vote, view, treasury))
# checked_add_u32 rejects with ErrTooManySigOps on uint32 overflow.
```

dcrd: internal/blockchain/validate.go · countSigOps, countP2SHSigOps, CountTotalSigOps

The arguments `is_coinbase` and `is_vote` are derived differently at dcrd's two call sites: from the form of the transaction for every transaction of both trees in the contextual check, and from its position (first of the regular tree; a vote in the stake tree) during connection. The two agree for every block that passes the other rules (Chapter 10).

No limit applies to a single transaction. Chapter 10 specifies the two block limits: the sum of `basic_sigops` over both trees, and, separately for each tree during connection, the running sum of `total_sigops`, each bounded by `MaxSigOpsPerBlock` (5000).

IMPLEMENTATION PITFALL
`p2sh_sigops` recognizes only the untagged pay-to-script-hash form. Inputs spending stake-tagged P2SH outputs (for example `OP_SSGEN` followed by a P2SH script) contribute only their quick count from `basic_sigops`. During connection the `p2sh_sigops` lookup happens before the transaction is connected, against the same UTXO state as Algorithm 7.6.

7.12 Script Validation

Ownership of an output is proved by script execution: for each input, the signature script of the input and the output script of the referenced output are executed together by the script engine (Chapter 6). This section specifies which inputs are executed, with which parameters, and the effect of a failure.

RULE 7.41 Inputs whose scripts are executed
For every transaction in both trees, every input whose outpoint index is **not** `ffffffff` (no UTXO entry can have that index) **MUST** have its scripts executed successfully, except that, if `autorevocations` is

active, no input of a stake-tree transaction that is a revocation (`is_revocation`) with version at least 2 is executed. Inputs with index `ffffffff` (the null inputs of the coinbase, vote stakebase, treasurybase and treasury spend) are never executed. The treasury spend's authorization is verified outside the script engine (Chapter 12).

dcrd: internal/blockchain/scriptval.go · checkBlockScripts
internal/blockchain/validate.go · checkConnectBlock

RATIONALE
Under DCP-0009, consensus already forces a version 2 revocation to pay exactly the amounts committed by the ticket to the committed addresses, with an empty signature script and no fee, so anyone may publish it and no signature is needed.

RULE 7.42 Script execution parameters
An executed input i of transaction T spending entry e is evaluated by the script engine with: output script e.script; script version e.script version (the version of the **spent output**, not of any output of T); the transaction T and input index i (for signature hashing); and the flag set F of Table 7.13, determined by the agenda state of the block containing T. The entry is read from the UTXO set after the whole tree containing T has been connected, ignoring its spent state.

dcrd: internal/blockchain/scriptval.go · validateHandler
internal/blockchain/validate.go · consensusScriptVerifyFlags

Table 7.13 Consensus script verification flags

Flag	Included when
`ScriptVerifyCleanStack`	Always (from the genesis block).
`ScriptVerifyCheckLockTimeVerify`	Always (from the genesis block).
`ScriptVerifyCheckSequenceVerify`	`lnfeatures` is active.
`ScriptVerifySHA256`	`lnfeatures` is active.
`ScriptVerifyTreasury`	`treasury` is active.
`ScriptDiscourageUpgradableNops`	Never in consensus (a policy flag, Chapter 17).
`ScriptVerifySigPushOnly`	Never in consensus, and not set by dcrd's policy either, which checks for push-only signature scripts separately (Chapter 17).

The meaning of each flag is specified in Chapter 6.

RULE 7.43 Script failure invalidates the block
If creating the script engine for an executed input fails (malformed scripts, a non-push-only signature script spending a P2SH-shaped output, and the other preconditions of Chapter 6; `ErrScriptMalformed`), or if execution fails (`ErrScriptValidation`), the block **MUST** be rejected. An output script with a nonzero script version passes execution once the engine is created (Chapter 6).

dcrd: internal/blockchain/scriptval.go · validateHandler

NOTE
dcrd validates scripts in parallel and caches verified signatures; the cache and the evaluation order have no effect on the verdict. dcrd skips script execution (and expiry and finality checks) for blocks that are ancestors of its configured assumed-valid block; that is a trust optimization described in Chapter 10, not a change to the rules.

7.13 Error Kinds

Table 7.14 maps each error kind of the reference implementation that arises from the rules of this chapter to the rule that produces it. Error kinds do not affect consensus; they are listed as a cross-check.

Table 7.14 Error kinds produced by the rules of this chapter

Error kind	Rule
`ErrNoTxInputs`, `ErrNoTxOutputs`	Rule 7.5
`ErrTxTooBig`	Rule 7.6
`ErrBadTxOutValue`	Rule 7.7, Rule 7.36, Rule 7.37
`ErrFraudAmountIn`	Rule 7.8, Rule 7.3
`ErrFraudBlockHeight`, `ErrFraudBlockIndex`	Rule 7.3
`ErrDuplicateTxInputs`	Rule 7.9
`ErrTxVersionTooHigh`	Rule 7.10
`ErrScriptVersionTooHigh`	Rule 7.11
`ErrRegTxCreateStakeOut`	Rule 7.12
`ErrScriptMalformed`	Rule 7.12, Rule 7.43
`ErrBadCoinbaseOutpoint`, `ErrBadCoinbaseFraudProof`, `ErrBadCoinbaseScriptLen`	Rule 7.13
`ErrBadStakebaseScriptLen`, `ErrBadStakebaseScrVal`	Rule 7.14
`ErrInvalidRevocationTxVersion`	Rule 7.15
`ErrBadTreasurybaseOutpoint`, `ErrBadTreasurybaseFraudProof`, `ErrBadTreasurybaseScriptLen`	Rule 7.16
`ErrBadTSpendOutpoint`, `ErrBadTSpendFraudProof`, `ErrBadTSpendScriptLen`	Rule 7.17
`ErrInvalidTAddChange`	Rule 7.18
`ErrBadTxInput`	Rule 7.14, Rule 7.15, Rule 7.19, Rule 7.33
`ErrStakeTxInRegularTree`, `ErrRegTxInStakeTree`, `ErrNonstandardStakeTx`	Rule 7.4
`ErrExpiredTx`	Rule 7.20
`ErrUnfinalizedTx`	Rule 7.21, Rule 7.24
`ErrMissingTxOut`	Rule 7.1, Rule 7.2, Rule 7.26, Rule 7.27, Rule 7.23, Rule 7.39
`ErrZeroValueOutputSpend`	Rule 7.28
`ErrImmatureSpend`	Rule 7.29, Rule 7.31, Rule 7.34, Rule 7.35
`ErrExpiryTxSpentEarly`	Rule 7.30
`ErrTxSStxOutSpend`	Rule 7.32
`ErrSpendTooHigh`	Rule 7.38
`ErrTooManySigOps`	Algorithm 7.7 (overflow); limits in Chapter 10
`ErrScriptValidation`	Rule 7.43

PART III

Consensus

Tickets and votes, proof of work, the two difficulty algorithms, block validation and chain selection, issuance, the treasury, and the process by which stakeholders change the rules.

CHAPTER 8

The Stake System

Decred's proof-of-stake layer lets holders of the currency lock funds in **tickets**. Every block draws a small set of tickets at random from the pool of live tickets; the holders of the drawn tickets publish **votes** that approve or disapprove the regular transactions of the block they vote on and express preferences on proposed rule changes. The next block is only valid if it carries votes from a majority of the drawn tickets. A ticket that is drawn but whose vote is not included is **missed**; a ticket that is never drawn eventually **expires**. The locked funds of missed and expired tickets are returned by a **revocation**.

This chapter specifies the three stake transaction types (ticket purchase, vote and revocation) byte for byte, the rules that tie their amounts to one another, the ticket database that every node maintains, the lottery that draws the winning tickets, and every block-level rule whose subject is a ticket, a vote or a revocation. Several neighboring subjects are specified elsewhere and only summarized here: the ticket price algorithm (stake difficulty) in Chapter 9; the value of the vote subsidy in Chapter 11; the treasury transactions that also live in the stake tree in Chapter 12; the meaning of the vote bits other than bit 0, and the vote version, in Chapter 13; the consequences of a disapproved block for its regular transactions in Chapter 10; the general transaction rules (sanity, fraud proofs, script execution, fees) in Chapter 7 and Chapter 6.

8.1 Overview

8.1.1 Terminology

Stake tree The second of the two transaction lists of a block (the first is the **regular tree**). Every ticket purchase, vote and revocation, and every treasury transaction, is in the stake tree; no other transaction may be (Rule 8.34).

Ticket A **ticket purchase transaction** (dcrd: *SStx*, "stake submission"). Its first output locks the **ticket price** under a script tagged `OP_SSTX`. The ticket is identified by its transaction hash.

Stake difficulty The minimum ticket price, carried in the `SBits` field of the block header and computed by the algorithm of Chapter 9.

Commitment An `OP_RETURN` output of a ticket that names, for one funding input, the hash of the address that will receive that input's share of the ticket's eventual return, the amount contributed, and optional fee limits (Section 8.3.1).

Immature ticket A ticket that has been mined but has not yet entered the live ticket pool.

Live ticket A ticket in the **live ticket pool**, eligible to be drawn.

Winner One of the `TicketsPerBlock` tickets drawn by the lottery run on block N; the winners of block N are the only tickets that may vote in block $N+1$, and their votes are votes *on* block N.

Vote A **vote transaction** (dcrd: *SSGen*, "stake generation") that spends a winning ticket, names the block it votes on, carries the vote bits, and pays the ticket price plus the **vote subsidy** to the ticket's commitments.

Stakebase The first input of a vote. It spends nothing and introduces the vote subsidy, much as a coinbase introduces the proof-of-work subsidy.

Missed ticket A winner whose vote is not included in the next block.

Expired ticket A live ticket that was not drawn within `TicketExpiry` blocks of entering the pool.

Revocation A **revocation transaction** (dcrd: *SSRtx*, "stake revocation") that spends a missed or expired ticket and returns the ticket price to the commitments.

8.1.2 Life of a ticket

Table 8.1 follows a ticket mined in the block at height p through its life, using main-network parameter values. All heights are consequences of rules stated later in this chapter, which are referenced in the last column.

Table 8.1 Timeline of a ticket mined at height p (main network: `TicketMaturity` $= 256$, `TicketExpiry` $= 40960$)

Height	Event	Reference
p	Ticket purchase mined. The price is at least the stake difficulty in the header of block p.	Rule 8.6
$p + 256$	The ticket enters the live pool while block $p + 256$ is connected, provided $p + 256 \geq$ `StakeEnabledHeight`.	Algorithm 8.9
$p + 256$	First lottery that can draw it: the lottery run on block $p + 256$ selects the winners for block $p + 257$.	Algorithm 8.9
$p + 257$	Earliest block that can contain its vote.	Rule 8.15
$p + 257$	Earliest block in which it can become missed (and, with DCP-0009 active, be revoked).	Rule 8.15
$p + 258$	Earliest revocation of a missed ticket before DCP-0009.	Rule 8.15
$p + 256 + 40960$	Last block in which it can vote. If it is not a winner for this block it expires while this block is connected.	Algorithm 8.9

8.1.3 Parameters and constants

Table 8.2 lists the network parameters used in this chapter with their main-network values; the values for the other networks are tabulated in Chapter 14. Table 8.3 lists the network-independent constants.

Table 8.2 Stake parameters (main network)

Parameter	Value	Meaning
`TicketsPerBlock`	5	Winners drawn per block; maximum votes per block.
`TicketMaturity`	256	Blocks between a ticket's purchase and its entry into the live pool.
`TicketExpiry`	40960	Blocks a ticket may remain live before it expires.
`StakeEnabledHeight`	512	First height at which tickets can enter the live pool and at which the ticket database processes votes, expiries and revocations.
`StakeValidationHeight`	4096	First height that must contain votes; the lottery first runs on the block one below it.
`MaxFreshStakePerBlock`	20	Maximum ticket purchases per block.
`MinimumStakeDiff`	200000000	Absolute minimum ticket price in atoms (2 DCR).
`StakeBaseSigScript`	`0000`	Required signature script of every stakebase input.
`SStxChangeMaturity`	1	Maturity of ticket change outputs (and of vote and revocation outputs before the treasury agenda).
`CoinbaseMaturity`	256	Maturity of vote and revocation outputs once the treasury agenda is active.
`TicketPoolSize`	8192	Not used by this chapter; used by the stake difficulty algorithm (Chapter 9).

> **NOTE**
> `StakeBaseSigScript` differs between networks: it is `0000` on the main, test and simulation networks and `7357` on the regression test network. `TicketsPerBlock` (5), `MaxFreshStakePerBlock` (20) and `SStxChangeMaturity` (1) are the same on all four networks; every other parameter in Table 8.2 differs from its main-network value on at least one other network; see Chapter 14.

Table 8.3 Network-independent constants

Name	Value	Meaning
`MaxInputsPerSStx`	64	Maximum inputs of a ticket.
`MaxOutputsPerSStx`	129	Maximum outputs of a ticket ($2 \times 64 + 1$).
`NumInputsPerSSGen`	2	Exact number of inputs of a vote.
`MaxOutputsPerSSGen`	66	Maximum outputs of a vote ($64 + 2$), including any treasury vote output. A vote for a ticket with 64 commitments therefore has no room for that output and can never cast treasury votes.
`NumInputsPerSSRtx`	1	Exact number of inputs of a revocation.
`MaxOutputsPerSSRtx`	64	Maximum outputs of a revocation.
`SStxPKHMinOutSize`	32	Minimum length of a commitment script.
`SStxPKHMaxOutSize`	77	Maximum length of a commitment script.
`SSGenBlockReferenceOutSize`	38	Exact length of a vote's block reference script.
`SSGenVoteBitsOutputMinSize`	4	Minimum length of a vote's vote bits script.
`SSGenVoteBitsOutputMaxSize`	77	Maximum length of a vote's vote bits script.
`MaxDataCarrierSize`	256	Maximum data in a null data script (Rule 8.3).
`TxVersionAutoRevocations`	2	Revocation version introduced by DCP-0009.
`TxVersionTreasury`	3	Transaction version required for a vote that carries treasury votes.
`maxRevocationsPerBlock`	255	Maximum revocations per block.
`earlyVoteBitsValue`	`0x0001`	Required header vote bits below `StakeValidationHeight`.
`seedConst`	`243f6a8885a308d3`	Constant appended to the lottery seed (the first 64 bits of the fractional part of π).

8.1.4 Rule changes affecting this chapter

Several consensus votes changed the rules in this chapter. Each rule that depends on one names the agenda; "the agenda is active" always means active *for the block being validated*, as defined in Chapter 13 (dcrd evaluates it on the block's parent).

Table 8.4 Agendas that modify stake rules

Agenda	DCP	Effect in this chapter
`treasury`	DCP-0006	Votes may carry treasury spend votes in a final output (Section 8.4.3); the number of required vote payments excludes that output; vote and revocation outputs need `CoinbaseMaturity` instead of `SStxChangeMaturity` confirmations (Rule 8.22); treasury transactions join the stake tree.
`autorevocations`	DCP-0009	Revocations must be version 2, carry an empty signature script, pay zero fee and distribute the rounding remainder pseudorandomly; they skip script validation; tickets may be revoked in the block in which they become missed or expired; blocks must revoke every such ticket.
`explicitverupgrades`	DCP-0008	Caps the transaction version at 3 (Chapter 7); this bounds the versions of stake transactions.
`changesubsidysplit`, `changesubsidysplitr2`	DCP-0010, DCP-0012	Change the vote subsidy value (Chapter 11); the formulas of this chapter are unchanged.

8.2 Classification and Common Script Forms

Whether a transaction is a ticket, a vote or a revocation is decided purely from its own contents, by three structural predicates defined in Section 8.3, Section 8.4 and Section 8.5. The classification does not depend on

the chain, on agenda state, or on the tree in which the transaction appears. A transaction that fails a predicate is not thereby invalid; it is simply not of that type. It will usually be invalid for some other reason, for example because it is a regular transaction placed in the stake tree (Rule 8.34) or because it creates stake-tagged outputs outside a stake transaction (Chapter 7).

RULE 8.1 Stake transaction type

The type of a transaction is the first of the following that applies:

1. *ticket*, if it satisfies the ticket predicate (Algorithm 8.1);
2. *vote*, if it satisfies the vote predicate (Algorithm 8.2);
3. *revocation*, if it satisfies the revocation predicate (Algorithm 8.3);
4. if its version is at least `TxVersionTreasury` $= 3$: *treasury add*, *treasury spend* or *treasurybase*, by the predicates of Chapter 12 in that order;
5. otherwise *regular*.

The three stake predicates are mutually exclusive, so the order of the first three tests does not matter in practice.

dcrd: blockchain/stake/staketx.go · DetermineTxType

8.2.1 Tagged scripts

Every non-`OP_RETURN` output of a stake transaction is a pay-to-pubkey-hash (P2PKH) or pay-to-script-hash (P2SH) script prefixed with one byte, a **stake tag** opcode, that marks how it was created and therefore how it may be spent (Chapter 6 gives the opcodes no execution semantics beyond a no-op).

Table 8.5 Stake tag opcodes and tagged script forms. h is a 20-byte hash.

Tag	Byte	Used for
`OP_SSTX`	`0xba`	Ticket output 0: the ticket price and the right to vote.
`OP_SSGEN`	`0xbb`	Vote payment outputs.
`OP_SSRTX`	`0xbc`	Revocation payment outputs.
`OP_SSTXCHANGE`	`0xbd`	Ticket change outputs.

Tagged P2PKH (26 bytes): $T \parallel 76 \parallel$ a9 $\parallel 14 \parallel h \parallel 88 \parallel$ ac Tagged P2SH (24 bytes): $T \parallel$ a9 $\parallel 14 \parallel h \parallel 87$

RULE 8.2 Tagged script

A public key script s with script version v is **tagged with** opcode T if and only if $v = 0$, $|s| \geq 1$, $s[0] = T$, and $s[1:]$ is exactly one of the following, where h is any 20 bytes:

- a 25-byte P2PKH script `76 a9 14` h `88 ac`, that is, `OP_DUP OP_HASH160 OP_DATA_20` h `OP_EQUALVERIFY OP_CHECKSIG`;
- a 23-byte P2SH script `a9 14` h `87`, that is, `OP_HASH160 OP_DATA_20` h `OP_EQUAL`.

No other length or form is accepted.

dcrd: blockchain/stake/scripttype.go · isTaggedScript

8.2.2 Null data scripts

RULE 8.3 Null data script

A public key script s with script version v is a **null data script** if and only if $v = 0$, $s[0] =$ `0x6a` (`OP_RETURN`), and either $|s| = 1$, or $s[1:]$ parses as exactly one opcode that consumes all of $s[1:]$, where that opcode is one of `OP_0` (`0x00`), a direct push `OP_DATA_1` … `OP_DATA_75` (`0x01` – `0x4b`), `OP_PUSHDATA1`, `OP_PUSHDATA2` or `OP_PUSHDATA4` (`0x4c` – `0x4e`), or `OP_1` … `OP_16` (`0x51` – `0x60`), and the number of bytes it pushes is at most `MaxDataCarrierSize` $= 256$. A push whose length prefix or data runs past the end

> of the script does not parse. Pushes need not be minimally encoded. `OP_0` and `OP_1` … `OP_16` push no data bytes for the purpose of this length test.
>
> dcrd: blockchain/stake/staketx.go · IsNullDataScript

8.3 Ticket Purchase Transactions

A ticket is funded by one or more inputs, possibly owned by different parties (this is how stake pools and ticket splitting work). For each input i the ticket carries a pair of outputs: a **commitment** naming who receives input i's share when the ticket is later spent, and a **change** output returning whatever part of input i was not contributed. Output 0 holds the ticket price; whoever can satisfy its script can vote with the ticket.

Table 8.6 Layout of a ticket with n inputs ($1 \le n \le 64$)

Position	Content
Inputs $0 \ldots n-1$	Any outpoints (subject to Rule 8.4). Input i funds the commitment at output $2i+1$.
Output 0	Ticket price. Script tagged `OP_SSTX` (voting rights).
Output $2i+1$	Commitment for input i: a null data script with the 30-byte payload of Table 8.7.
Output $2i+2$	Change for input i. Script tagged `OP_SSTXCHANGE`. The value may be zero.

8.3.1 The commitment output

Table 8.7 Commitment output script (output $2i+1$ of a ticket); offsets are into the public key script

Field	Type	Bytes	Description
`[0]`	`uint8`	1	`0x6a` (`OP_RETURN`).
`[1]`	`uint8`	1	Push opcode `0x1e` … `0x4b` (`OP_DATA_30` … `OP_DATA_75`); equals $\lvert s \rvert - 2$.
`[2:22]`	`bytes`	20	Payment hash h: a public key hash (P2PKH) or a script hash (P2SH), selected by the flag in bit 63 of the next field.
`[22:30]`	`uint64 LE`	8	Encoded amount a. Bit 63 (the top bit of byte 29) is the P2SH flag; the committed amount is a AND $(2^{63}-1)$.
`[30:32]`	`uint16 LE`	2	Fee limits f (Table 8.8).
`[32:]`	`bytes`	0–45	Arbitrary, ignored.

Table 8.8 Bits of the 16-bit fee limits field f

Bits	Mask	Meaning
0–5	`0x003f`	k_{vote}: base-2 logarithm of the fee limit for votes.
6	`0x0040`	Vote fee limit flag. When clear, each vote output must pay exactly its computed amount.
7	`0x0080`	Unused, ignored.
8–13	`0x3f00`	k_{rev}: base-2 logarithm of the fee limit for revocations, $(f$ AND 0x3f00$) \gg 8$.
14	`0x4000`	Revocation fee limit flag (ignored for revocations once DCP-0009 is active).
15	`0x8000`	Unused, ignored.

How the fee limits constrain votes and revocations is specified in Rule 8.18.

8.3.2 Structure

ALGORITHM 8.1 **Ticket predicate**

```
OP_RETURN, OP_SSTX, OP_SSGEN, OP_SSRTX, OP_SSTXCHANGE = 0x6a, 0xba, 0xbb, 0xbc, 0xbd
```

```
def is_ticket(tx) -> bool:
    if len(tx.inputs) > 64:                          # MaxInputsPerSStx
        return False
    if len(tx.outputs) > 129:                        # MaxOutputsPerSStx
        return False
    if len(tx.outputs) == 0:
        return False
    for out in tx.outputs:
        if out.script_version != 0:
            return False
    if not is_tagged(0, tx.outputs[0].script, OP_SSTX):
        return False
    if 2 * len(tx.inputs) + 1 != len(tx.outputs):
        return False
    for i in range(1, len(tx.outputs)):
        s = tx.outputs[i].script
        if i % 2 == 0:                               # change output
            if not is_tagged(0, s, OP_SSTXCHANGE):
                return False
        else:                                        # commitment output
            if not is_null_data(0, s):
                return False
            if len(s) < 32 or len(s) > 77:
                return False
            if s[0] != OP_RETURN or not (0x1e <= s[1] <= 0x4b):
                return False
    return True
```

dcrd: blockchain/stake/staketx.go · CheckSStx

Algorithm 8.1 implies that a commitment script is exactly $2 + m$ bytes with a direct push of $m \in [30, 75]$ bytes, so the 30 bytes of Table 8.7 are always present. A transaction with no inputs is excluded by the transaction sanity rules (Chapter 7), not by this predicate.

> **NOTE**
> The value fields of commitment outputs are not constrained by any stake rule. They are ordinary, unspendable outputs: any value placed in them counts towards the ticket's total output value (and so reduces its fee) and is lost.

8.3.3 Inputs, commitments and price

RULE 8.4 Ticket input script forms

For every input of a ticket, the referenced output **MUST** exist and be unspent, **MUST** have script version 0, and its public key script **MUST** be one of: an untagged P2PKH script (25 bytes) or untagged P2SH script (23 bytes) of the exact forms in Rule 8.2, or a script tagged with `OP_SSGEN`, `OP_SSRTX` or `OP_SSTXCHANGE`. In particular a ticket cannot be funded by an `OP_SSTX` output, a treasury (`OP_TGEN`) output, or any other script type.

dcrd: internal/blockchain/validate.go · checkTicketPurchaseInputs, isAllowedTicketInputScriptForm

RULE 8.5 Ticket commitment amounts

For every input i of a ticket, let v_i be the amount of the output it spends, c_i the committed amount of output $2i + 1$ (bytes 22–29 decoded as a little-endian `uint64` with bit 63 cleared, taken as `int64`), and r_i the value of output $2i + 2$. Then $c_i + r_i$ **MUST** equal v_i exactly.

dcrd: internal/blockchain/validate.go · checkTicketPurchaseInputs, extractTicketCommitAmount

RULE 8.6 Ticket price

The value of output 0 of every ticket in a block's stake tree **MUST** be at least the `SBits` field of that block's header (`ErrNotEnoughStake`), and **MUST** be at least `MinimumStakeDiff` (`ErrStakeBelowMinimum`). Both comparisons are "not less than": a ticket may pay more than the stake difficulty. The check uses the block's own claimed `SBits` and is context free; Rule 9.6 separately requires the claim to equal the stake difficulty computed by Chapter 9.

dcrd: internal/blockchain/validate.go · checkProofOfStake

The general input rules (Chapter 7) also apply to tickets: total input value at least total output value, no zero-value inputs, fraud proofs, maturity of spent outputs, script validation, and the prohibition on spending outputs created in the same block (Rule 8.23).

NOTE

Write P for the ticket price (output 0), $C = \sum c_i$ for the total commitment and $R = \sum r_i$. The input rules give $\sum v_i \geq P + R +$ (values of the commitment outputs), and Rule 8.5 gives $\sum v_i = C + R$, so $C \geq P > 0$. The difference $C - P$ is the ticket's transaction fee plus any value placed in commitment outputs. Since $C > 0$, the division in Algorithm 8.4 never divides by zero.

TEST VECTOR A ticket purchase (dcrd regression-network test chain, block 19)

```
ticket 2f278527eea8d2e6663c547f9b669a6c1aaa8f330bac43f16bb8bb93e0d21df5 (display)
input 0 spends 320000 atoms
out 0  value 20000  ba76a9149bc83f416940190002e6bdded468709d87ffd36088ac
       = OP_SSTX-tagged P2PKH, hash 9bc83f416940190002e6bdded468709d87ffd360
out 1  value 0      6a1e9bc83f416940190002e6bdded468709d87ffd36000e20400000000000058
       push 0x1e (30 bytes)
       hash     9bc83f416940190002e6bdded468709d87ffd360
       amount   00e2040000000000 -> 0x000000000004e200 = 320000, bit 63 clear: P2PKH
       limits   0058 -> 0x5800: vote flag (0x0040) clear -> votes pay exact amounts
                revocation flag (0x4000) set, k_rev = (0x5800 & 0x3f00) >> 8 = 24
out 2  value 0      bd76a914000000000000000000000000000000000000000088ac
       = OP_SSTXCHANGE-tagged P2PKH (change of 0 atoms)
check  c_0 + r_0 = 320000 + 0 = v_0   ticket fee = 320000 - 20000 = 300000
```

8.4 Vote Transactions

A vote spends a winning ticket. Its first input, the stakebase, creates the vote subsidy; its second input spends output 0 of the ticket. Its first two outputs are data: the block being voted on and the vote bits. Then follows one payment per ticket commitment, in commitment order. Once the treasury agenda is active a vote may end with one more data output carrying votes on treasury spends.

Table 8.9 Layout of a vote for a ticket with n commitments

Position	Content
Input 0 (stakebase)	Previous outpoint: hash all zero, index `0xffffffff` , tree 0. Fraud proof: block height 0, block index `0xffffffff` , value in = vote subsidy. Signature script = `StakeBaseSigScript`.
Input 1	Outpoint (ticket hash, index 0, tree 1): output 0 of the ticket. Signature script satisfies the `OP_SSTX` script.
Output 0	Block reference: `6a 24` ‖ block hash (32 bytes) ‖ LE_{32}(block height). Exactly 38 bytes.
Output 1	Vote bits: `6a` ‖ push(2…75 bytes) beginning with LE_{16}(vote bits).
Outputs 2…$n+1$	Payment j (output $j+2$) pays commitment j (ticket output $2j+1$); script tagged `OP_SSGEN` .
Output $n+2$ (optional)	Treasury votes: `6a` ‖ push(`"TV"` ‖ 1 to 7 entries of 33 bytes). Only with the treasury agenda active (Section 8.4.3).

8.4.1 Block reference and vote bits outputs

Table 8.10 Vote output 0: block reference script (exactly 38 bytes)

Field	Type	Bytes	Description
[0]	uint8	1	`0x6a` (`OP_RETURN`).
[1]	uint8	1	`0x24` (`OP_DATA_36`).
[2:34]	hash	32	Hash of the block voted on, in internal byte order (the order in which a block hash is serialized, not display order).
[34:38]	uint32 LE	4	Height of the block voted on.

Table 8.11 Vote output 1: vote bits script ($4 \le |s| \le 77$)

Field	Type	Bytes	Description
[0]	uint8	1	`0x6a` (`OP_RETURN`).
[1]	uint8	1	Push opcode `0x02` … `0x4b` ; equals $|s| - 2$.
[2:4]	uint16 LE	2	Vote bits (Section 8.10).
[4:8]	uint32 LE	0 or 4	Vote version. Present only when $|s| \ge 8$; otherwise the vote version is 0.
[8:]	bytes	0–69	Remaining extended vote bits; no consensus meaning.

dcrd calls bytes $[4 : |s|)$ the **extended vote bits** (at most $75 - 2 = 73$ bytes); the vote version occupies the first four of them.

8.4.2 Structure

ALGORITHM 8.2 **Vote predicate**

```
NOT_A_VOTE = None

def is_vote(tx):          # returns the list of treasury votes, or NOT_A_VOTE
    if len(tx.inputs) != 2:                       # NumInputsPerSSGen
        return NOT_A_VOTE
    if len(tx.outputs) > 66:                      # MaxOutputsPerSSGen
        return NOT_A_VOTE
    if len(tx.outputs) < 2:
        return NOT_A_VOTE
    sb = tx.inputs[0]                             # stakebase
    if not (sb.prev.hash == bytes(32) and sb.prev.index == 0xffffffff
            and sb.prev.tree == 0):
        return NOT_A_VOTE
    if not (sb.block_height == 0 and sb.block_index == 0xffffffff):
        return NOT_A_VOTE
    if tx.inputs[1].prev.index != 0:
        return NOT_A_VOTE
    if tx.inputs[1].prev.tree != 1:               # stake tree
        return NOT_A_VOTE
    for out in tx.outputs:
        if out.script_version != 0:
            return NOT_A_VOTE
    s0 = tx.outputs[0].script
    if not is_null_data(0, s0) or len(s0) != 38 or s0[0:2] != b"\x6a\x24":
        return NOT_A_VOTE
    s1 = tx.outputs[1].script
    if not is_null_data(0, s1) or not (4 <= len(s1) <= 77):
        return NOT_A_VOTE
    if s1[0] != 0x6a or not (0x02 <= s1[1] <= 0x4b):
        return NOT_A_VOTE
    end = len(tx.outputs)
    tvotes = []
```

```
    last = tx.outputs[-1]
    if is_null_data(last.script_version, last.script):
        end = end - 1
        tvotes = parse_treasury_votes(last.script)
        if tvotes is NOT_A_VOTE:
            return NOT_A_VOTE
        if not (len(tvotes) > 0 and tx.version == 3):   # TxVersionTreasury
            return NOT_A_VOTE
    for i in range(2, end):
        if not is_tagged(0, tx.outputs[i].script, OP_SSGEN):
            return NOT_A_VOTE
    return tvotes
```

dcrd: blockchain/stake/staketx.go · CheckSSGenVotes, IsStakeBase

IMPLEMENTATION PITFALL
The test for a treasury vote output looks only at the *last* output and applies whenever it is a null data script, whatever the transaction version and whatever the agenda state. In a two-output transaction the last output is the vote bits output itself, so such a transaction is a vote only if its vote bits push happens to parse as treasury votes (and its version is 3). Conversely, a transaction whose last output is any null data script that does not parse as treasury votes is not a vote at all.

8.4.3 Treasury votes

The format of the treasury vote output and the function `parse_treasury_votes` used by Algorithm 8.2 are specified in Chapter 12 (Algorithm 12.6). Wherever that algorithm rejects, `parse_treasury_votes` yields `NOT_A_VOTE` here: the transaction is not a vote at all.

Because a null data script pushes at most 256 bytes and an `OP_PUSHDATA1` push at most 255, a vote carries between 1 and 7 treasury votes ($2 + 7 \times 33 = 233$; $2 + 8 \times 33 = 266$). The hashes are treasury spend transaction hashes in internal byte order. How treasury votes are tallied is specified in Chapter 12.

RULE 8.7 Treasury votes require the treasury agenda
When computing the number of payment outputs of a vote (Rule 8.16) and when matching outputs to commitments (Rule 8.17), the final output is excluded if and only if the treasury agenda is active for the block and the vote has a treasury vote output. Before the agenda is active the treasury vote output, if any, is therefore counted and matched as a payment, which always fails (Rule 8.16 or Rule 8.17). In effect, votes **MUST NOT** carry treasury votes before the treasury agenda is active.

dcrd: internal/blockchain/validate.go · checkVoteInputs, checkTicketRedeemerCommitments

8.4.4 Contextual checks

RULE 8.8 Stakebase signature script
The signature script of input 0 of a vote **MUST** be between 2 and 100 bytes long inclusive (`MinCoinbaseScriptLen`, `MaxCoinbaseScriptLen`) and **MUST** be byte-for-byte equal to `StakeBaseSigScript`.

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 8.9 Ticket input not null
The previous outpoint of input 1 of a vote, and of input 0 of a revocation, **MUST NOT** be the null outpoint (all-zero hash, index `0xffffffff`, tree 0). (This is already implied by the predicates, which require index 0 and tree 1 for a vote and tree 1 for a revocation.)

dcrd: internal/blockchain/validate.go · checkTransactionContext

The stakebase input is not a real spend: it is skipped by the UTXO lookup, fraud-proof, maturity and script checks of Chapter 7, and its signature script is excluded from signature operation counting (Chapter 6). Its value-in contributes the vote subsidy to the vote's inputs (Rule 8.19).

The vote's version is otherwise bounded only by the general maximum transaction version (Chapter 7); a vote with a treasury vote output must have version exactly 3 by Algorithm 8.2.

TEST VECTOR A vote (dcrd regression-network test chain, block 144)

```
vote 86f999e29e60384f60c4bc8309f022ef2641e51b4c1f416954cd63a636c60344 (display)
in 0  stakebase: prev 00..00:ffffffff tree 0, height 0, index ffffffff,
      value in 2970297029 (the vote subsidy)
in 1  prev 49d9a5021ea29435e45c4a1342d687102d7b307bfd42903fb16a952fd23d912a:0
      tree 1
out 0 6a24bf7895d75e5da09e707eab994c05a30bf7c5c7e39ace6aed779609b36f0100008f000000
      hash, internal order:
        bf7895d75e5da09e707eab994c05a30bf7c5c7e39ace6aed779609b36f010000
      = block (display order):
        0000016fb3099677ed6ace9ae3c7c5f70ba3054c99ab7e709ea05d5ed79578bf
      height 8f000000 -> 143
out 1 6a020100   push of 2 bytes: vote bits 0x0001 (approve), vote version 0
out 2 value 3252437701
      bb76a91463ac6e0158d456f34131bca6eb8b471638daa5ec88ac (OP_SSGEN P2PKH)
ticket: price 282140672, one commitment of 282440672, vote fee flag clear
expected payment = floor(282440672 * (282140672 + 2970297029) / 282440672)
                 = 3252437701  (exact, since the vote fee flag is clear)
```

NOTE
The stakebase signature script of this test-chain vote (`deadbeef`) is not today's: the test chain (dcrd's `blockchain/stake/testdata/testexpiry.bz2`) was generated with a different genesis block, block-one ledger, organization script and `StakeBaseSigScript`. Its proof-of-work limit and difficulty rules are those of today's regression network.

8.5 Revocation Transactions

A revocation spends a missed or expired ticket and returns the ticket price to the commitments, one output per commitment, in commitment order. Before DCP-0009 [33] the revocation had to be signed by the holder of the voting rights; since DCP-0009 anyone, typically the miner, may create it, it carries no signature and pays no fee, and its outputs are fully determined by the ticket and the parent block.

Table 8.12 Layout of a revocation for a ticket with n commitments

Position	Content
Input 0	Outpoint (ticket hash, index 0, tree 1). Signature script empty if the version is at least 2.
Outputs $0 \ldots n-1$	Payment j pays commitment j (ticket output $2j+1$); script tagged `OP_SSRTX`.

ALGORITHM 8.3 **Revocation predicate**

```
def is_revocation(tx) -> bool:
    if len(tx.inputs) != 1:                          # NumInputsPerSSRtx
        return False
    if len(tx.outputs) > 64:                         # MaxOutputsPerSSRtx
        return False
    if len(tx.outputs) == 0:
        return False
    for out in tx.outputs:
        if out.script_version != 0:
```

```
            return False
    if tx.inputs[0].prev.tree != 1:                # stake tree
        return False
    for out in tx.outputs:
        if not is_tagged(0, out.script, OP_SSRTX):
            return False
    if tx.version >= 2:                            # TxVersionAutoRevocations
        if len(tx.inputs[0].sig_script) != 0:
            return False
        if sum(out.value for out in tx.outputs) < tx.inputs[0].value_in:
            return False                           # non-zero fee
    return True
```

dcrd: blockchain/stake/staketx.go · CheckSSRtx

The version-2 conditions are part of the predicate itself and apply regardless of agenda state. The `value_in` they use is the fraud-proof amount of the input, which the general input rules require to equal the amount of the spent ticket output (Chapter 7); with that, the condition says the fee is zero. The predicate does not check the input's outpoint index; Rule 8.12 does.

RULE 8.10 Revocation version

If the `autorevocations` agenda is active for the block, every revocation **MUST** have version exactly $\texttt{TxVersionAutoRevocations} = 2$.

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 8.11 Revocation script validation

If the `autorevocations` agenda is active for the block, the input scripts of a revocation whose version is at least 2 are not executed. In every other case the signature script of a revocation's input **MUST** satisfy the ticket's `OP_SSTX` output script under the rules of Chapter 6.

dcrd: internal/blockchain/scriptval.go · checkBlockScripts, ValidateTransactionScripts

NOTE

Before DCP-0009 a revocation of version 2 or higher is required by the predicate to have an empty signature script, yet its script is executed and an empty signature script cannot satisfy a tagged P2PKH or P2SH script. Such revocations therefore never validate; before activation, valid revocations have version 0 or 1.

TEST VECTOR A revocation of an expired ticket, pre-DCP-0009 rules (regression-network test chain, block 420)

```
revocation (display hash), version 1:
  3ad0c0fd5720d0266d213815614a7dcd08047aa83f21820d22fd3921b7edb0c8
in 0  prev acc72577d3c3efac89975fddbe22d18eca143a6882d7a49714da81a98f6ed43f:0
      signed; the ticket has price 20000 and one commitment of 320000
      to 9bc83f41...d360 with fee limits 0x5800
out 0 value 20000  bc76a9149bc83f416940190002e6bdded468709d87ffd36088ac
      = OP_SSRTX-tagged P2PKH to the committed hash 9bc83f41...d360
expected amount = floor(320000 * 20000 / 320000) = 20000
revocation fee flag set, k_rev = 24, 2^24 >= 20000:
      allowed range [0, 20000]
```

8.6 Input and Amount Validation

This section specifies the checks that need the UTXO set: that a vote or revocation spends a real, mature ticket; that it pays the ticket's commitments exactly the amounts they are owed; and the maturity rules for outputs created by stake transactions. They are applied to each transaction of the stake tree, in order, as part

of connecting the block (Chapter 7 gives the surrounding procedure). Throughout, h is the height of the block containing the transaction being checked.

8.6.1 Spending a ticket

RULE 8.12 Ticket outpoint index
The ticket input of a vote (input 1) and of a revocation (input 0) **MUST** reference output index 0 of the ticket.
dcrd: internal/blockchain/validate.go · checkVoteInputs, checkRevocationInputs

RULE 8.13 Ticket output available
The output referenced by the ticket input of a vote or revocation **MUST** exist in the UTXO set as seen by the transaction (that is, after the effects of the preceding transactions of the block) and be unspent.
dcrd: internal/blockchain/validate.go · checkVoteInputs, checkRevocationInputs

RULE 8.14 Ticket submission output
The output referenced by the ticket input of a vote or revocation **MUST** have script version 0, its public key script **MUST** be tagged with `OP_SSTX` (Rule 8.2), and the transaction that created it **MUST** have been classified as a ticket (Rule 8.1).
dcrd: internal/blockchain/validate.go · checkTicketSubmissionInput, isStakeSubmission

RULE 8.15 Ticket spend maturity
Let p be the height of the block that contains the ticket. A vote at height h **MUST** satisfy $h - p \geq \texttt{TicketMaturity} + 1$. A revocation at height h **MUST** satisfy $h - p \geq \texttt{TicketMaturity} + 2$ if the `autorevocations` agenda is not active for the block, and $h - p \geq \texttt{TicketMaturity} + 1$ if it is.
dcrd: internal/blockchain/validate.go · checkVoteInputs, checkRevocationInputs

RATIONALE
A ticket mined at p enters the live pool while block $p + \texttt{TicketMaturity}$ is connected and can first be drawn for the following block, so $+1$ is the earliest possible vote. Before DCP-0009 a revocation required the ticket to be in the missed set of the parent block, which cannot happen before $+2$. With DCP-0009 a ticket may be revoked in the very block in which it is missed (code comment in `internal/blockchain/validate.go · checkRevocationInputs`).

These maturity rules are necessary but far from sufficient: which tickets may be voted or revoked in a block is decided by the ticket database (Rule 8.42). A ticket that never entered the live pool (Section 8.8.3) can be neither voted nor revoked.

The general input checks of Chapter 7 also apply to the ticket input, in particular the fraud proof (the input's value in, block height and block index must match the ticket output) and script validation (subject to Rule 8.11).

8.6.2 Number of payments

RULE 8.16 Number of payments
Let $n = (|\text{ticket outputs}| - 1)/2$ be the number of commitments of the spent ticket. For a vote, let $e = 1$ if the treasury agenda is active for the block and the vote has a treasury vote output (Algorithm 8.2 returned a non-empty list), and $e = 0$ otherwise; the vote **MUST** have exactly $2 + n + e$ outputs. A revocation **MUST** have exactly n outputs.
dcrd: internal/blockchain/validate.go · checkVoteInputs, checkRevocationInputs

8.6.3 Return amounts

When a ticket is spent, the total to be returned is the ticket price P, plus the vote subsidy S for a vote. The total is divided among the commitments in proportion to the amounts they committed, which include each contributor's share of the ticket purchase fee. The division rounds every share down; the rounding remainder, between 0 and $n - 1$ atoms, is left as transaction fee for votes and (before DCP-0009) for revocations. Under DCP-0009 the remainder of a revocation is distributed one atom at a time to outputs chosen by the lottery's pseudorandom generator, seeded with the serialized header of the parent block, so that the fee can be exactly zero.

ALGORITHM 8.4 **Ticket return amounts**

```
def commit_amount(script) -> int64:
    return int64(le64_decode(script[22:30]) & (2**63 - 1))

def return_amounts(ticket_outs, price, vote_subsidy, prev_header_bytes,
                   is_vote, auto_revocations_active) -> list[int64]:
    commits = [commit_amount(ticket_outs[i].script)
               for i in range(1, len(ticket_outs), 2)]
    contribution_sum = sum(commits)          # > 0 for every valid ticket
    total = price + vote_subsidy             # vote_subsidy = 0 for revocations
    returns = []
    for c in commits:                        # unbounded (big) integers here
        returns.append(((c * total) << 32) // contribution_sum >> 32)
    if is_vote or not auto_revocations_active:
        return returns
    remainder = total - sum(returns)
    if remainder > 0:
        prng = Hash256PRNG(prev_header_bytes)          # Algorithm "PRNG"
        for _ in range(remainder):
            returns[prng.uniform(uint32(len(returns)))] += 1
    return returns
```

dcrd: internal/blockchain/validate.go · calcTicketReturnAmounts

The intermediate product $c \times \text{total} \times 2^{32}$ exceeds 64 bits and must be computed with arbitrary precision (dcrd uses `big.Int`); all operands are non-negative, so the floor division and right shift are exact floors. The formula equals $\lfloor c \times \text{total} \,/\, \text{contribution_sum} \rfloor$ exactly, but an implementation that evaluates it differently must take care not to overflow. Each result fits in `int64` because it does not exceed total.

The pseudorandom generator is the one of Algorithm 8.6, with the 180-byte serialized header of the parent of the block being validated (the block at height $h - 1$) as seed. A fresh generator is created for every revocation, so all revocations in one block draw from the same sequence. That sequence is also the one the lottery used to select the winners for block h.

TEST VECTOR Return amounts (computed with dcrd's `calcTicketReturnAmounts`)

```
prev header (180 bytes, from dcrd validate_test.go):
  07000000dc02335daa073d293e1b150648f0444a60b9c97604abd01e0000000000000003c449b23
  21c4bd0d1fa76ed59f80ebaf46f16cfb2d17ba46948f09f21861095566482410a463ed49473c2727
  8cd7a2a3712a3b19ff1f6225717d3eb71cc2b5590100012c7312a3c30500050095a100000cf42418
  f1820a870300000020a10700091600005b32a55f5bcce3107883210000746994395800 2e00000000
  0000000000000000000000000000000000007000000
PRNG IV  fd99cb2f0ea29b91d6d4f3366299a9969598ce5c2ff275bd5ed034a23fe993db
first rand32 values 4254714671 245537681 3604280118 1654237590
  -> uniform(3): 2 2 0 0      uniform(5): 1 1 3 0       (2^32 mod 3 = 2^32 mod 5 = 1)

commitments [100000000, 200000000, 300000007], price 600000005
  vote, subsidy 0           -> [99999999, 199999999, 300000005]   sum 600000003
  vote, subsidy 123456789   -> [120576130, 241152261, 361728401]  sum 723456792
```

```
  revocation, pre-DCP-0009  -> [99999999, 199999999, 300000005]   sum 600000003
  revocation, DCP-0009      -> [99999999, 199999999, 300000007]   sum 600000005
                               (remainder 2, both atoms to index 2)
commitments [100000000] x 5, price 400000004
  revocation, DCP-0009      -> [80000001, 80000002, 80000000, 80000001, 80000000]
                               (remainder 4 to indexes 1, 1, 3, 0)
```

8.6.4 Matching outputs to commitments

Payment j of a vote is output $j+2$; payment j of a revocation is output j. Payment j corresponds to commitment j, which is output $2j+1$ of the ticket, and to the j-th return amount A_j of Algorithm 8.4 (computed with P = value of ticket output 0 and, for votes, S = the vote subsidy of Rule 8.19).

RULE 8.17 Payment script

For every payment j (every output from the first payment position up to, but excluding, the treasury vote output when it is excluded by Rule 8.16):

1. its script version **MUST** be 0;
2. if bit 63 of commitment j's encoded amount is set (byte 29 of the commitment script has its top bit set), its script **MUST** be a P2SH script tagged with `OP_SSGEN` (vote) or `OP_SSRTX` (revocation), in the exact 24-byte form of Rule 8.2; otherwise it **MUST** be a P2PKH script with the same tag in the exact 26-byte form;
3. the 20-byte hash in that script **MUST** equal bytes 2–21 of commitment j's script.

`dcrd: internal/blockchain/validate.go · checkTicketRedeemerCommitments`

RULE 8.18 Payment value and fee limits

Let f be the fee limits field of commitment j (Table 8.8). Payment j is **fee-limited** if

- the transaction is a vote and bit `0x0040` of f is set, or
- the transaction is a revocation, bit `0x4000` of f is set, and it is not the case that the `autorevocations` agenda is active for the block and the revocation's version is at least 2.

If payment j is not fee-limited its value **MUST** equal A_j exactly. If it is fee-limited, let $k = f \text{ AND } 0\text{x}003\text{f}$ for a vote or $k = (f \text{ AND } 0\text{x}3\text{f}00) \gg 8$ for a revocation, and let

$$L = \begin{cases} A_j - 2^k & \text{if } k < 63 \text{ and } 2^k < A_j \\ 0 & \text{otherwise} \end{cases} \tag{8.1}$$

Then its value v **MUST** satisfy $L \le v \le A_j$.

`dcrd: internal/blockchain/validate.go · checkTicketRedeemerCommitments`

NOTE

The fee limit applies to each payment separately, not to the transaction fee as a whole. Since k has six bits, $k = 63$ is the only value that always permits the whole payment to be taken as fee; a smaller k does so only when $2^k \ge A_j$. The computation 2^k never overflows `int64` because it is only evaluated for $k < 63$. Once DCP-0009 is active every revocation is version 2 (Rule 8.10) and therefore not fee-limited: each payment is exactly A_j, and because the remainder has been distributed the payments sum to exactly P, which is consistent with the zero-fee condition of Algorithm 8.3.

NOTE

The values of a vote's two data outputs (and of a treasury vote output) are not constrained by any stake rule; value placed there is unspendable and counts against the general requirement that outputs not exceed inputs.

8.6.5 The vote subsidy

RULE 8.19 Stakebase amount
Let H_v be the height encoded in bytes 34–37 of the vote's output 0 (the height of the block voted on). The value-in field of the vote's input 0 **MUST** equal the vote subsidy for height H_v, computed by the vote subsidy function of Chapter 11 (dcrd `CalcStakeVoteSubsidyV3`) with the subsidy split variant selected by the agendas active for the block: the DCP-0012 variant if `changesubsidysplitr2` is active, otherwise the DCP-0010 variant if `changesubsidysplit` is active, otherwise the original. This value is the subsidy S used in Algorithm 8.4.

dcrd: `internal/blockchain/validate.go · checkVoteInputs`

NOTE
A dcrd source comment records that the subsidy "really should" be computed from the height of the block containing the vote rather than the block voted on, and that this is now part of consensus. Since votes must vote on the parent block (Rule 8.40), $H_v = h - 1$ in every valid block.

RULE 8.20 Stakebase totals
Let S_{h-1} be the vote subsidy for height $h - 1$ (same function and variant as in Rule 8.19). For every vote in the block, the sum of all its output values minus the amount of the ticket output it spends **MUST NOT** exceed S_{h-1}. Summed over all votes in the block, the total output value minus the total spent ticket amount **MUST NOT** exceed $S_{h-1} \times$ Voters if $h \geq$ `StakeValidationHeight`, and **MUST NOT** exceed 0 otherwise, where `Voters` is the header field.

dcrd: `internal/blockchain/validate.go · checkStakeBaseAmounts, getStakeBaseAmounts, checkTransactionsAndConnect`

These two checks are implied by the preceding rules in any block that satisfies them, but they are separate checks in dcrd and are stated for completeness. The stake tree is processed before the regular tree; transaction fees of stake transactions are added to the fees available to the coinbase as described in Chapter 11.

8.6.6 Summary: checking a vote or revocation

Algorithm 8.5 collects the transaction-level checks of this section for one vote or revocation. It adds nothing to the rules above; it shows which data each check needs. The block-level eligibility of the ticket (Rule 8.42) is checked separately.

ALGORITHM 8.5 Input checks for a vote or revocation at height h

```
def check_redeemer_inputs(tx, h, utxo, parent_header, flags, params):
    is_vote = tx_type(tx) == VOTE
    if is_vote:
        tin = tx.inputs[1]
        voted_height = le32_decode(tx.outputs[0].script[34:38])
        subsidy = vote_subsidy(voted_height, flags.subsidy_variant)
        if tx.inputs[0].value_in != subsidy:
            reject("bad stakebase amount")
    else:
        tin = tx.inputs[0]
        subsidy = 0
    if tin.prev.index != 0:
        reject("ticket input must reference output 0")
    entry = utxo.lookup(tin.prev)
    if entry is None or entry.spent:
        reject("missing ticket output")
    if (entry.script_version != 0 or not is_tagged(0, entry.script, OP_SSTX)
            or entry.tx_type != TICKET):
        reject("not a ticket submission output")
```

```
    extra = 1 if is_vote or flags.auto_revocations else 2
    if h - entry.height < params.TicketMaturity + extra:
        reject("immature ticket spend")
    ticket_outs = entry.ticket_outputs        # every output of the ticket
    n = (len(ticket_outs) - 1) // 2
    if is_vote:
        e = 1 if flags.treasury and len(is_vote_tvotes(tx)) > 0 else 0
        first, payments = 2, len(tx.outputs) - 2 - e
    else:
        first, payments = 0, len(tx.outputs)
    if payments != n:
        reject("wrong number of payments")
    amounts = return_amounts(ticket_outs, ticket_outs[0].value, subsidy,
                             serialize(parent_header), is_vote,
                             flags.auto_revocations)
    for j in range(payments):
        check_payment(tx, first + j, ticket_outs[2 * j + 1].script,
                      amounts[j], is_vote, flags)   # payment script and value rules
```

dcrd: internal/blockchain/validate.go · checkVoteInputs, checkRevocationInputs, checkTicketRedeemerCommitments

Here `is_vote_tvotes(tx)` is the list returned by Algorithm 8.2.

IMPLEMENTATION PITFALL

Validating a vote or revocation needs **all** outputs of the spent ticket: the ticket price from output 0 and the hashes, amounts and fee limits of the commitment outputs, which are unspendable null data outputs that a UTXO set would normally discard. dcrd stores them with the ticket's output 0 in its UTXO entry for as long as that output is unspent (`internal/blockchain/utxoentry.go · TicketMinimalOutputs`). An implementation must likewise retain every output (value, script version and script) of each unspent ticket.

8.6.7 Outputs created by stake transactions

RULE 8.21 Ticket outputs are spent only by votes and revocations

An input of any transaction that is neither a vote nor a revocation **MUST NOT** spend an output whose script (with its script version) is tagged with `OP_SSTX`.

dcrd: internal/blockchain/validate.go · CheckTransactionInputs

RULE 8.22 Maturity of stake outputs

Let o be the height of the block containing the spent output and h the height of the spending transaction's block.

- An output tagged with `OP_SSGEN` or `OP_SSRTX` **MUST NOT** be spent unless $h - o \geq$ `SStxChangeMaturity` when the treasury agenda is not active for the block, or $h - o \geq$ `CoinbaseMaturity` when it is.
- An output tagged with `OP_SSTXCHANGE` **MUST NOT** be spent unless $h - o \geq$ `SStxChangeMaturity`.

dcrd: internal/blockchain/validate.go · CheckTransactionInputs

IMPLEMENTATION PITFALL

`SStxChangeMaturity` is 1 on every network, so before the treasury agenda vote rewards and revocation refunds could be spent in the very next block. Informative documentation that describes a 256-block wait for vote rewards matches the consensus rule only once the treasury agenda is active. The general coinbase and expiry maturity rules of Chapter 7 still apply independently.

RULE 8.23 Stake transactions do not spend same-block outputs

No input of a ticket, vote, revocation, treasury add, treasury spend or treasurybase (other than input 0 of a vote, treasury spend or treasurybase, which spends nothing) may spend an output created in the same block (in either tree), except the single historical outputs listed in Table 8.13: on the given network, at the given height, output 0 of the regular-tree transaction with the given hash may be spent by a stake transaction of the same block.

dcrd: internal/blockchain/validate.go · CheckTransactionInputs, isSBSSViolation

Table 8.13 Same-block stake spends grandfathered into the chain (hashes in display order)

Network	Height	Transaction
main	1106817	0875ecdc5f12c8aa6ee0d54331fd5b5e786639882a3adb09882de5e3b939613b
main	1106831	714e71358937cd33480fd2900853d43ea3440ca67b38bacdfeaeb0c37c80ee27
main	1106837	aeba09a4efa334e3c13e68f27f40bd37cda7643c263f88d69f38a33c2cdacd18
main	1106854	17228266d9d6dd7084aa2e479319aca6d755c2a9d2dbb3cf28c8151470c7a439
main	1106860	20d6d3ed1b609f03969474dca1f452c25e035766457dfd58061f9b9270fc4e6d
main	1107195	5d2e1898fe0c631cb795b1906fb3cf8d9772f1826be8dea64dd3ab85ac0ab2d3
testnet3	1980161	451b6eed1a777bc5bbb5c1dbe10a9e444cbeb7d863adb75fc6b7a676b3dcbb58

The mechanics of this exception (dcrd re-adds output 0 of the listed transaction to the view before connecting the stake tree) belong to the UTXO model of Chapter 7.

8.7 The Lottery

After each block at height $N \geq$ `StakeValidationHeight` $- 1$ is connected, every node draws `TicketsPerBlock` distinct winners from the live ticket pool as it stands after that block. The draw is a deterministic function of two things only: the serialized header of block N and the set of live tickets. Because the header commits to the block's transactions and to its proof of work, a miner cannot cheaply bias the draw, and every node computes the same winners. The winners of block N may vote on block N in block $N + 1$. A six-byte digest of the draw, the **final state**, is committed in the header of block $N + 1$ so that a node can check it has the same live pool and the same winners before validating the votes.

All hashing in this section is BLAKE-256 (Chapter 3), written $\mathrm{BLAKE256}(x)$, producing 32 bytes.

8.7.1 The pseudorandom generator

The generator, `Hash256PRNG`, turns a seed of arbitrary length into a stream of 32-bit integers. The seed is hashed together with a constant to a 32-byte initialization vector (IV). The first eight outputs are the eight big-endian 32-bit words of the IV itself; every following group of eight is the words of $\mathrm{BLAKE256}(\mathrm{IV} \parallel \mathrm{BE}_{32}(i))$ for $i = 0, 1, 2, \ldots$.

ALGORITHM 8.6 Hash256PRNG

```
SEED_CONST = bytes.fromhex("243f6a8885a308d3")

class Hash256PRNG:
    def __init__(self, seed: bytes):
        self.seed = BLAKE256(seed + SEED_CONST)   # the IV
        self.last_hash = self.seed
        self.hash_idx = 0      # next 4-byte word of last_hash to use, 0..7
        self.idx = 0           # uint64: number of hashes derived from seed

    def rand32(self) -> uint32:
        i = 4 * self.hash_idx
```

```
        r = be32_decode(self.last_hash[i:i + 4])
        self.hash_idx += 1
        if self.hash_idx > 7:
            self.last_hash = BLAKE256(self.seed + be32(uint32(self.idx)))
            self.idx += 1
            self.hash_idx = 0
        if self.idx > 0xffffffff:                # unreachable in practice
            self.seed = BLAKE256(self.seed)
            self.last_hash = self.seed
            self.idx = 0
        return r

    def state_hash(self) -> bytes:              # 32 bytes
        return BLAKE256(self.last_hash + be32(uint32(self.idx))
                        + bytes([self.hash_idx]))
```

dcrd: blockchain/stake/lottery.go · CalcHash256PRNGIV, NewHash256PRNGFromIV, Hash256Rand, StateHash

Note the order of operations in `rand32`: the word is read first, and the next hash is computed immediately after the eighth word of a hash has been read, not lazily before the ninth. This matters for `state_hash`, which depends on `last_hash`, `idx` and `hash_idx` exactly as they stand after the last call. The rollover of `idx` past $2^{32} - 1$ would require more than 2^{35} calls and cannot occur in consensus use; it is included for completeness.

IMPLEMENTATION PITFALL
Both 32-bit encodings in this generator are **big-endian**, unlike almost every other integer in the protocol: the words read from the hash and the counter appended to the IV. The counter appended for the first derived hash is 0, so the ninth output comes from BLAKE256(IV ‖ `00000000`).

8.7.2 Uniform integers

ALGORITHM 8.7 Uniform random integer in [0, n)

```
def uniform(prng: Hash256PRNG, n: uint32) -> uint32:    # prng.uniform(n)
    if n < 2:
        return 0                            # consumes no output
    if n > 0x80000000:
        lo = uint32(2**32 - n)
    else:
        lo = uint32(0xffffffff - 2 * n + 1) % n
    while True:                             # rejection sampling
        r = prng.rand32()
        if r >= lo:
            return r % n
```

dcrd: blockchain/stake/lottery.go · UniformRandom

In both branches `lo` equals $2^{32} \bmod n$ (for $n = 2^{31}$ the expression $2^{32} - 2n$ is 0), so the accepted values $[2^{32} \bmod n, 2^{32})$ form a range whose length is a multiple of n and the result is exactly uniform. Values below `lo` are discarded and another word is drawn.

TEST VECTOR Hash256PRNG and uniform, seed = ASCII “Decred” (446563726564)

```
IV = d48f4175da6800a5e3557cb4205f8be944483480935c0f2f30d7d4eaabd2efcf
rand32 #0..#11:
  3566158197 3664248997 3814030516  543132649 1145582720 2472283951
   819451114 2882727887 2142381687 1037079434  122677752 3963859140
(#0..#7 are the words of the IV; #8 = 0x7fb22677 is the first word
 of BLAKE256(IV || 00000000))
```

```
state_hash after 12 calls (idx = 1, hash_idx = 4):
  84b9d4f546995a04697b695feb9497d66e46a23b82c339cfdd03e4282c0d6ce5

fresh generator, successive calls:
  uniform(1) = 0                (no output consumed)
  uniform(2) = 1                (r = 3566158197, lo = 0)
  uniform(5) = 2                (r = 3664248997, lo = 1)
  uniform(40960) = 40116        (r = 3814030516, lo = 0)
  uniform(2147483649) = 324800302
                                (lo = 2147483647: r = 543132649 and 1145582720
                                 rejected, r = 2472283951 accepted)
  uniform(4294967295) = 819451114  (lo = 1)

dcrd TestBasicPRNG: seed = BLAKE256(01), 100000 calls, state_hash =
  e4c445ebe0815aaf9944f94b63d388566132b72e1ee2d3a985fcfbae72cdf124
```

8.7.3 Order of the live ticket pool

dcrd stores the live tickets in a treap keyed by ticket hash, with the ticket's live height as heap priority. Only the key order matters for consensus: a winner index i denotes the ticket at position i of the pool in ascending key order.

RULE 8.24 Live pool order

The live ticket pool is ordered by ticket hash, ascending, where two hashes are compared as 32-byte strings in **internal byte order** (the byte order in which the hash is produced by BLAKE-256 and serialized in an outpoint), lexicographically from byte 0, each byte as an unsigned value. Position 0 is the smallest hash. The heights of the tickets play no part in the order.

dcrd: blockchain/stake/internal/tickettreap/immutable.go · get, Put, GetByIndex
blockchain/stake/lottery.go · fetchWinners

IMPLEMENTATION PITFALL

Display-form hashes are byte-reversed (Chapter 2). Sorting display strings, or sorting hashes interpreted as little-endian 256-bit integers, gives a different order. In terms of display strings the comparison starts at the *last* two hexadecimal digits. Any data structure may be used; a sorted array of 32-byte keys reproduces dcrd exactly.

8.7.4 Drawing the winners and the final state

In the algorithms of this section `fail(reason)` means that the block cannot be connected. It is distinct from `reject` because dcrd does not report the condition as a blockchain rule violation (the draw failure is an untyped error, the others are stake-package errors), so the block is not marked invalid; it is refused either way.

ALGORITHM 8.8 Winner selection and final state

```
def draw_winners(header_bytes: bytes, pool: list[bytes], k: int):
    # header_bytes: the 180-byte serialized header of the block just connected
    # pool: live ticket hashes after connecting it, sorted as in the pool order rule
    # k: TicketsPerBlock
    prng = Hash256PRNG(header_bytes)
    size = len(pool)
    if size < k:
        fail("cannot draw winners")          # block cannot be connected
    idxs = []
    while len(idxs) < k:
        r = prng.uniform(uint32(size))
        if r not in idxs:                    # draw again on a repeat
```

```
            idxs.append(r)
    winners = [pool[i] for i in idxs]        # in draw order, not sorted
    final_state = BLAKE256(b"".join(winners) + prng.state_hash())[0:6]
    return winners, final_state
```

dcrd: blockchain/stake/tickets.go · connectNode
blockchain/stake/lottery.go · findTicketIdxs, fetchWinners
internal/blockchain/blockindex.go · lotteryIV

The winners keep the order in which they were drawn; the final state hashes their concatenation in that order (each hash in internal byte order) followed by the generator's state hash after the last draw. A pool larger than $2^{32} - 1$ tickets is an error in dcrd and cannot occur. The case `size < k` is discussed in Section 8.8.3.

TEST VECTOR Lottery on a ten-ticket pool (computed with dcrd's `connectNode`)

```
seed (stands in for a header): ASCII "block one header"
    = 626c6f636b206f6e6520686561646572
IV = 4295eeba3103039d7adb28dee716bbceb7bce96ee13f5089d43daea53a43c1fd
pool = { BLAKE256(byte i) : i = 0..9 }, sorted (internal byte order):
  [0] 0ce8d4ef4dd7cd8d62dfded9d4edb0a774ae6a41929a74da23109e8f11139c87  BLAKE256(00)
  [1] 19b7506ad9c189a9f8b063d2aee15953d335f5c88480f8515d7d848e7771c4ae  BLAKE256(06)
  [2] 49af37ab5270015fe25276ea5a3bb159d852943df23919522a202205fb7d175c  BLAKE256(02)
  [3] 4a6c419a1e25c85327115c4ace586decddfe2990ed8f3d4d801871158338501d  BLAKE256(01)
  [4] 4c2231813064f8500edae05b40195416bd543fd3e76c16d6efb10c816d92e8b6  BLAKE256(09)
  [5] 4c6eb9e38415034f4c93d3304d10bef38bf0ad420eefd0f72f940f11c5857786  BLAKE256(04)
  [6] 53d661e71e47a0a7e416591200175122d83f8af31be6a70af7417ad6f54d0038  BLAKE256(07)
  [7] 9bff7982eab6f7883322edf7bdc86a23c87ca1c07906fbb1584f57b197dc6253  BLAKE256(08)
  [8] b706d561742ad3671703c247eb927ee8a386369c79644131cdeb2c5c26bf6c5d  BLAKE256(03)
  [9] bdd15db13448905791a70b68137445e607cca06cc71c7a58b9b2e84a06c54d08  BLAKE256(05)
k = 5: indexes drawn [4, 7, 8, 6, 5]  (6 draws: idx = 0, hash_idx = 6 afterwards)
state_hash  = 45a5fba5bbb9c9058645d09514427a53722ec3f2ab4076f650741f06a6bd4229
final state = 27a0cf022cbd

next block (seed "block two header"): winners [4] [7] [8] vote, [6] and [5] are
missed; the pool is now the 5 remaining tickets and all 5 are drawn:
indexes [3, 1, 0, 4, 2] over the new pool, final state = 27c864c436ea
```

TEST VECTOR Duplicate rejection: seed "Decred", pool size 7, k = 5

```
rand32 #0..#6 mod 7 = 0 3 2 3 2 4 6   (lo = 2^32 mod 7 = 4; no rejections)
indexes = [0, 3, 2, 4, 6]             (draws #3 and #4 repeat and are discarded)
state_hash = 95e00d629ec81c0e67a570021bf081437385ce01db184af466c7773bbdbb1e97
```

TEST VECTOR A real draw: dcrd regression-network test chain, lottery on block 143

```
header of block 143 (180 bytes):
  000000008379bcce660662c28735988b54d193bd9386c501acae677fe002326c66b10000a1c527ea
  1a1abceb6a145579ba739a0fb0de3ba3594f8717c62e3fc1d88b25fe0000000000000000000000
  0000000000000000000000000000000000000000000010000000000000000000000080070000f7e4001f
  000048340400000008f0000007e0100006d5cec57306f010000000000000000000000000000000000
  00000000000000000000000000000000000000000000
IV = f680c8ab1cac33b1b9fd72165d34adaeaca23a564bfe16f55976b6953a338e72
live pool after block 143: 1940 tickets
indexes [1315, 1193, 1774, 302, 1558]
winners (internal byte order):
  adad145e4bb898ea4e2e4ab4eac39cad1d628cead91e245df48d1851cc8001ff
  9f2fcb5eb5dd8710c219db9899eb1bd8c60163eaa25abecbfeeca0c5c8df4cc2
  eb97034120808b03bf4155183094e37cd723de0f1d2e470d5b0482adc7e9f530
  2a913dd22f956ab13f9042fd7b307b2d1087d642134a5ce43594a21e02a5d949
```

```
  cd615d92df7eb0f042c36899b4dbafa9935b8fc320c32cd689e21054acae63b5
state_hash = 0ca9f8c7ca073e2f9e4e2303b2a632407975d6c10dcb4e0e106233c671ceb980
final state = ff00007be270  = FinalState in the header of block 144 (PoolSize 1940)
```

The fourth winner of the last vector is the ticket spent by the vote shown in Section 8.4.4 (whose display-form hash `49d9a5021ea29435e45c4a1342d687102d7b307bfd42903fb16a952fd23d912a` is the reverse of the internal form above).

8.8 The Ticket Database

Every node maintains, for each block, the set of live tickets, the set of missed tickets and the set of revoked tickets as of that block, together with the winners drawn on that block and its final state. dcrd calls this per-block state a **stake node** (`blockchain/stake/tickets.go · Node`). The state of a block is a function of the state of its parent and of the block itself; the genesis block's state is empty.

8.8.1 Ticket states

Table 8.14 Ticket states. Each ticket is in at most one of the three sets.

State	Set	Leaves the state when
Immature	none	Block $p + $ `TicketMaturity` is connected: the ticket becomes live (if that height is at least `StakeEnabledHeight`).
Live	live, with live height L	It is a winner of the parent of the block being connected (then voted or missed), or that block's height reaches $L + $ `TicketExpiry` (then expired).
Voted	none (removed)	Final. The vote is the ticket's only spend.
Missed	missed, flag *expired* = false	Revoked.
Expired	missed, flag *expired* = true	Revoked.
Revoked	revoked	Final.

The **live height** L of a ticket is the height of the block whose connection made it live, $L = p +$ `TicketMaturity`, not the height of its purchase. Expired tickets are kept in the missed set; the expired flag only distinguishes them for reporting. In consensus terms "missed set" always means missed or expired, not yet revoked.

8.8.2 Connecting a block

ALGORITHM 8.9 **Stake state transition when connecting the block at height h**

```
def connect(parent: StakeState, block, h: int, params) -> StakeState:
    st = parent.copy()          # live: hash -> L; missed: hash -> (L, expired); revoked
    # Tickets spent in this block, in stake-tree order:
    voted   = [tx.inputs[1].prev.hash for tx in block.stake_txs
               if is_vote(tx) is not NOT_A_VOTE]
    revoked = [tx.inputs[0].prev.hash for tx in block.stake_txs if is_revocation(tx)]
    if h >= params.StakeEnabledHeight:
        # 1. Every voted ticket must be a winner drawn on the parent.
        for t in voted:
            if t not in parent.winners:
                fail("unknown ticket spent")
        # 2. Each parent winner is now either voted or missed.
        for w in parent.winners:
            L = st.live.pop(w)                    # must be present
            if w not in voted:
                st.missed[w] = (L, False)
```

```
        # 3. Expire live tickets whose live height is old enough.
        expire_h = h - params.TicketExpiry if h > params.TicketExpiry else 0
        for t, L in list(st.live.items()):
            if L <= expire_h:
                del st.live[t]
                st.missed[t] = (L, True)
        # 4. Revocations move tickets from missed to revoked.
        for t in revoked:
            if t not in st.missed:
                fail("revoked ticket not missed")
            st.revoked[t] = st.missed.pop(t)
        # 5. Tickets bought TicketMaturity blocks ago become live.
        for t in tickets_in(ancestor(block, h - params.TicketMaturity)):
            if t in st.live:
                fail("duplicate ticket")
            st.live[t] = h                          # live height L = h
    # 6. Draw the winners for block h + 1.
    st.winners, st.final_state = [], bytes(6)
    if h >= params.StakeValidationHeight - 1:
        st.winners, st.final_state = draw_winners(serialize(block.header),
                                                  sorted(st.live),
                                                  params.TicketsPerBlock)
    return st
```

dcrd: blockchain/stake/tickets.go · connectNode
internal/blockchain/stakenode.go · maybeFetchNewTickets, fetchStakeNode
blockchain/stake/staketx.go · FindSpentTicketsInBlock

Here `tickets_in(B)` is the list of hashes of the stake-tree transactions of block B that are tickets (Rule 8.1), and `ancestor(block, x)` is the ancestor of `block` at height x on its own chain. The order of steps 1 to 5 is significant: winners are processed before expiry, so a winner whose expiry height has also arrived becomes voted or missed, never expired; expiry precedes revocations, which (with DCP-0009) lets a block revoke a ticket that expires in that same block; and new tickets are added before the draw, so a ticket that becomes live at height h can be drawn for block $h + 1$. The order of tickets within each step affects only dcrd's undo records, not the resulting sets.

RULE 8.25 Stake state transition

A block can only be connected if Algorithm 8.9 succeeds for it: every vote spends a winner drawn on its parent, every revoked ticket is in the missed set after steps 2 and 3, no ticket becomes live twice, and, at heights $h \geq$ `StakeValidationHeight` $- 1$, the live pool after step 5 contains at least `TicketsPerBlock` tickets. A block for which it fails **MUST NOT** become part of the chain.

dcrd: blockchain/stake/tickets.go · connectNode | internal/blockchain/chain.go · connectBlock

NOTE

In dcrd a failure of Algorithm 8.9 is reported as an ordinary error rather than a rule violation, because in any block that has passed the checks of Section 8.9 the first two conditions are already guaranteed (by Rule 8.42) and the third by unique transaction hashes. The pool-size condition is a genuine consensus constraint: a block at height $h \geq$ `StakeValidationHeight` $- 1$ after which the live pool would hold fewer than `TicketsPerBlock` tickets cannot be connected, so a chain whose pool is about to run out cannot be extended past that point. Mining software avoids producing such a chain (Section 8.11).

TEST VECTOR State transitions (dcrd's `connectNode` with `TicketsPerBlock` $= 2$, `StakeEnabledHeight` $= 1$, `StakeValidationHeight` $= 2$, `TicketExpiry` $= 3$)

```
Ti = BLAKE256(byte i); hashes abbreviated to their first 4 bytes (internal order).
Block headers are replaced by the seeds "h1".."h4". Only the ticket database
transition is exercised; the block-level rules are not applied.
```

```
block 1: T0..T9 become live (L = 1)
  live    T0 0ce8d4ef  T6 19b7506a  T2 49af37ab  T1 4a6c419a  T9 4c223181
          T4 4c6eb9e3  T7 53d661e7  T8 9bff7982  T3 b706d561  T5 bdd15db1
  winners [T6, T1]                    final state 3aced47dce07
block 2: votes T6; revokes T1 (missed in step 2, revoked in step 4)
  live    T0 T2 T9 T4 T7 T8 T3 T5     revoked T1
  winners [T7, T5]                    final state 95ea5ea60e92
block 3: votes T7; T5 is missed and not revoked
  live    T0 T2 T9 T4 T8 T3           missed T5     revoked T1
  winners [T4, T9]                    final state 93df9dcae44a
  expiring in block 4 (e = 4 - 3 = 1, winners excluded): T0 T2 T8 T3
block 4: votes T4; T9 missed (step 2); T0 T2 T8 T3 expire (step 3);
         revokes T9 T0 T2 T8 T3 (step 4); T10 T11 T12 become live (L = 4)
  live    T12 61a3403f  T11 7b527886  T10 e81db4f0
  missed  T5            revoked T0 T2 T1 T9 T8 T3
  winners [T11, T10]                  final state 52c4fe648924
```

8.8.3 Early heights

The two activation parameters divide the chain into three phases.

Before `StakeEnabledHeight` Steps 1 to 5 are skipped entirely. No ticket becomes live, nothing expires, and no winners exist. Tickets may be bought (subject to the price rules) from the first block that can fund them.

From `StakeEnabledHeight` to `StakeValidationHeight` −2 Tickets become live and can expire; the parent's winner list is empty, so step 2 does nothing. Blocks may not contain votes or revocations (Rule 8.27, Rule 8.36).

From `StakeValidationHeight` −1 The draw runs after every block. The first winners are drawn on block `StakeValidationHeight` −1, and block `StakeValidationHeight` is the first block that must carry votes.

IMPLEMENTATION PITFALL

The block at height h adds the tickets of block $h -$ `TicketMaturity` only when $h \geq$ `StakeEnabledHeight`. A ticket mined in a block below `StakeEnabledHeight` − `TicketMaturity` would never enter the live pool and could never be voted or revoked. On all four networks `StakeEnabledHeight` equals `CoinbaseMaturity` + `TicketMaturity` (256 + 256 on the main network, 16 + 16 on the others), and every spendable output descends from a coinbase of block 1 or later, which cannot be spent before height 1 + `CoinbaseMaturity`; so no ticket can be mined that early and every ticket does enter the pool. An implementation must nevertheless gate step 5 on $h \geq$ `StakeEnabledHeight` exactly as stated.

The header fields `PoolSize` and `FinalState` of block h commit to the state of block $h - 1$ (Rule 8.32, Rule 8.33). For every block below `StakeValidationHeight` the final state is therefore the all-zero value, matching Rule 8.27; the first non-zero final state is in the header of block `StakeValidationHeight`.

8.8.4 Tickets expiring in the next block

The block-level revocation rules of DCP-0009 need to know, when validating block h, which tickets block h will expire. This set is computed from the parent's state.

RULE 8.26 Tickets expiring in block h

Let $e = h -$ `TicketExpiry` if $h >$ `TicketExpiry` and $e = 0$ otherwise. The tickets **expiring in block** h are the tickets in the parent's live set whose live height equals e, excluding the parent's winners. (Every live ticket with a smaller live height has already expired, so this is the same set that step 3 of Algorithm 8.9 expires.)

dcrd: blockchain/stake/tickets.go · ExpiringNextBlock

8.8.5 Disconnecting a block (informative)

A reorganization requires undoing blocks. Since the state of a block is a function of its parent's state and the block, an implementation may simply recompute or cache states. dcrd instead records, for every block, the list of ticket transitions it performed (each with the ticket's live height and its flags) and applies them in reverse order to restore the parent's live, missed and revoked sets. The parent's winners are recovered from the transitions that removed winners (spent ones and missed non-expired ones), restored to their original draw order, and the parent's final state is recomputed by re-running the draw on the parent's header. Reverse order matters because, with DCP-0009, one ticket can be missed (or expired) and revoked in the same block. None of this affects consensus.

8.9 Block-Level Stake Rules

The rules in this section apply to a whole block. Together with the rules above they determine which stake transactions a block may contain and tie the stake fields of the header to the block's contents and to the ticket database. Table 8.15 lists the header fields involved; their serialization is specified in Chapter 4.

Table 8.15 Stake-related block header fields

Field	Type	Must equal
`VoteBits`	`uint16`	`0x0001` below `StakeValidationHeight`; afterwards bit 0 must match the majority of the votes (Rule 8.41). Other bits are unconstrained here.
`FinalState`	`[6]byte`	Final state of the lottery drawn on the parent (Rule 8.33).
`Voters`	`uint16`	Number of votes in the stake tree (Rule 8.38).
`FreshStake`	`uint8`	Number of tickets in the stake tree (Rule 8.31).
`Revocations`	`uint8`	Number of revocations in the stake tree (Rule 8.39).
`PoolSize`	`uint32`	Number of live tickets after the parent (Rule 8.32).
`SBits`	`int64`	Stake difficulty (Chapter 9); minimum ticket price (Rule 8.6).

8.9.1 Context-free header rules

RULE 8.27 Stake header fields below stake validation height

If the header's `Height` is less than `StakeValidationHeight`, its `Voters` **MUST** be 0, its `Revocations` **MUST** be 0, its `VoteBits` **MUST** equal `0x0001`, and its `FinalState` **MUST** be six zero bytes.

dcrd: internal/blockchain/validate.go · checkBlockHeaderSanity

RULE 8.28 Minimum number of votes

If the header's `Height` is at least `StakeValidationHeight`, its `Voters` **MUST** be at least $\lfloor \texttt{TicketsPerBlock}/2 \rfloor + 1$ (3 on every network, where $\texttt{TicketsPerBlock} = 5$).

dcrd: internal/blockchain/validate.go · checkBlockHeaderSanity

RULE 8.29 Maximum number of votes

The header's `Voters` **MUST NOT** exceed `TicketsPerBlock`.

dcrd: internal/blockchain/validate.go · checkBlockHeaderSanity

RULE 8.30 Maximum number of ticket purchases

The header's `FreshStake` **MUST NOT** exceed `MaxFreshStakePerBlock`.

dcrd: internal/blockchain/validate.go · checkBlockHeaderSanity

RULE 8.31 Ticket purchase count
The header's `FreshStake` **MUST** equal the number of transactions in the block's stake tree that are tickets.
dcrd: internal/blockchain/validate.go · checkBlockDataSanity

Rule 8.6, which checks every ticket in the stake tree against the header's `SBits` and `MinimumStakeDiff`, is also a context-free block check in dcrd.

8.9.2 Header commitments to the ticket database

RULE 8.32 Pool size commitment
The header's `PoolSize` **MUST** equal the number of tickets in the live set of the parent block's stake state (the live pool from which the winners for this block were drawn).
dcrd: internal/blockchain/validate.go · checkBlockHeaderContext

RULE 8.33 Final state commitment
The header's `FinalState` **MUST** equal the final state of the parent block's stake state: the value computed by Algorithm 8.8 on the parent, or six zero bytes if no draw was made on the parent.
dcrd: internal/blockchain/validate.go · checkBlockHeaderContext

NOTE
dcrd skips these two checks, together with the difficulty, stake difficulty and stake version checks and the ticket redeemer check of Rule 8.42, when it adds a block that is an ancestor of its assumed-valid block or while in bulk-import mode (`BFFastAdd`). That is an optimization for blocks presumed valid; the rules hold for every block of the chain.

8.9.3 Stake tree composition

RULE 8.34 Tree membership
Every transaction in a block's regular tree **MUST** be of type *regular* (Rule 8.1), and every transaction in its stake tree **MUST NOT** be of type *regular*.
dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 8.35 Allowed stake transaction types
Every transaction in the stake tree **MUST** be a ticket, a vote, a revocation, or, only if the treasury agenda is active for the block, a treasury add, a treasury spend or a treasurybase. A stake-tree transaction classified as a treasury type while the treasury agenda is not active makes the block invalid.
dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 8.36 No votes or revocations below stake validation height
If the block's height is less than `StakeValidationHeight`, every transaction in its stake tree **MUST** be a ticket, or, if the treasury agenda is active, a treasury add or treasurybase. (Votes, revocations and treasury spends are excluded.)
dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 8.37 Non-empty stake tree
If the block's height is at least `StakeValidationHeight`, its stake tree **MUST NOT** be empty. (This is implied by Rule 8.28 and Rule 8.38.)
dcrd: internal/blockchain/validate.go · checkTransactionsAndConnect

RULE 8.38 Vote count
The header's `Voters` **MUST** equal the number of votes in the stake tree.
dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 8.39 Revocation count and limit
The number of revocations in the stake tree **MUST NOT** exceed 255 (`maxRevocationsPerBlock`), and the header's `Revocations` **MUST** equal that number.
dcrd: internal/blockchain/validate.go · checkBlockContext

There is no rule on the order of tickets, votes and revocations within the stake tree. The only positional requirement in the stake tree is that a treasurybase, when the treasury agenda is active, is its first transaction (Chapter 12). Because stake transactions cannot spend outputs of the same block (Rule 8.23), their relative order cannot affect the validity of their inputs.

8.9.4 Votes

RULE 8.40 Votes vote on the parent
If the block's height h is at least `StakeValidationHeight`, the block reference of every vote in the stake tree (Table 8.10) **MUST** name the block's parent: its 32-byte hash **MUST** equal the header's `PrevBlock` field and its height **MUST** equal $h - 1$.
dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 8.41 Parent approval
If the block's height is at least `StakeValidationHeight`, let Y be the number of votes in the stake tree whose vote bits have bit 0 set and V the total number of votes. The votes **approve** the parent if $Y > V - Y$ (a strict majority; a tie disapproves). Bit 0 of the header's `VoteBits` **MUST** be set if the votes approve the parent and **MUST** be clear otherwise.
dcrd: internal/blockchain/validate.go · checkBlockContext, voteBitsApproveParent

What happens to the parent's regular tree when it is disapproved is specified in Chapter 10: from height 2 on, a block whose header has bit 0 of `VoteBits` clear undoes the effects of its parent's regular tree before its own transactions are applied. Below `StakeValidationHeight` the header value is fixed at `0x0001` (approve) by Rule 8.27.

8.9.5 Votes and revocations against the ticket database

ALGORITHM 8.10 Ticket redeemer check for the block at height h

```
def check_ticket_redeemers(block, h, parent: StakeState, auto_revocations_active):
    # h: height of block; parent: stake state of block h - 1
    vote_tickets = [tx.inputs[1].prev.hash for tx in block.stake_txs
                    if is_vote(tx) is not NOT_A_VOTE]
    revoke_tickets = [tx.inputs[0].prev.hash for tx in block.stake_txs
                      if is_revocation(tx)]
    has_vote = {w: False for w in parent.winners}
    for t in vote_tickets:
        if t not in has_vote:
            reject("vote for ineligible ticket")
        has_vote[t] = True
    newly_missed = {w for w, voted in has_vote.items() if not voted}
    expiring = set(expiring_in_block(parent, h))      # expiring-tickets rule
    revoked = set()
    for t in revoke_tickets:
        eligible = ((auto_revocations_active
```

```
                    and (t in newly_missed or t in expiring))
                  or t in parent.missed)
        if not eligible:
            reject("revocation of ineligible ticket")
        revoked.add(t)
    if auto_revocations_active:
        for t in newly_missed:
            if t not in revoked:
                reject("missing revocation of missed ticket")
        for t in expiring:
            if t not in revoked:
                reject("missing revocation of expired ticket")
```

dcrd: internal/blockchain/validate.go · checkTicketRedeemers, checkBlockContext

RULE 8.42 Votes and revocations spend eligible tickets

If the block's height is at least StakeValidationHeight, Algorithm 8.10 **MUST NOT** reject it. That is:

1. every vote **MUST** spend one of the winners drawn on the parent;
2. if the `autorevocations` agenda is not active for the block, every revocation **MUST** spend a ticket in the parent's missed set;
3. if it is active, every revocation **MUST** spend a ticket that is in the parent's missed set, or is a winner of the parent without a vote in this block, or expires in this block (Rule 8.26); and the block **MUST** contain a revocation for every winner of the parent that has no vote in this block and for every ticket that expires in this block.

dcrd: internal/blockchain/validate.go · checkTicketRedeemers, checkBlockContext

Duplicate spends of one ticket (two votes, two revocations, or a vote and a revocation) are excluded by the ordinary double-spend rule, since all of them spend output 0 of the ticket.

RATIONALE

DCP-0009 makes revocation automatic: as soon as a ticket is missed or expires, the block in which that happens must return its funds. Because a version-2 revocation needs no signature and its outputs are fully determined (Rule 8.18), the miner can always construct it.

NOTE

Tickets that became missed before DCP-0009 activated and were never revoked remain in the missed set and may still be revoked at any later height. The required revocations of a block number at most $\left\lfloor \frac{\text{TicketsPerBlock}}{2} \right\rfloor$ missed winners (a valid block carries at least a majority of the votes) plus the tickets that became live together TicketExpiry blocks earlier (at most MaxFreshStakePerBlock), well below the limit of 255.

8.10 Vote Bits and Vote Version

Each vote carries 16 **vote bits** (bytes 2–3 of output 1, little-endian) and, optionally, a 32-bit **vote version** (bytes 4–7). At the transaction level only two things are defined here:

- **Bit 0** (`0x0001`, dcrd `BlockValid`) is the voter's verdict on the regular tree of the block voted on: set means approve, clear means disapprove. The verdicts of the votes in a block are combined by Rule 8.41.
- The **vote version** is the little-endian `uint32` at bytes 4–7 of the vote bits script if that script is at least 8 bytes long, and 0 otherwise.

Table 8.16 Extraction of the vote fields from vote output 1 (script s)

Field	Value
vote bits	$\mathrm{LE}_{16}(s[2:4])$

Field	Value
approves parent	(vote bits AND 0x0001) $\neq 0$
vote version	$\mathrm{LE}_{32}(s[4:8])$ if $\lvert s\rvert \geq 8$, else 0
voted block hash	$s_0[2:34]$ of output 0 (internal byte order)
voted block height	$\mathrm{LE}_{32}(s_0[34:38])$ of output 0

The meaning of bits 1–15 (choices on consensus rule changes, each agenda occupying a masked group of bits), how the vote version of the votes in a block feeds the stake version and the rule-change tallies, and which votes are counted, are specified in Chapter 13. No stake rule constrains bits 1–15 or the vote version; a vote with any values is valid as far as this chapter is concerned.

8.11 Non-Consensus Behavior

The following is policy: it describes what dcrd's mempool, block template generator and wallets do, not what a block must satisfy. Details are in Chapter 17.

- **Automatic revocations.** Once DCP-0009 is active, dcrd's block template generator creates a version-2, zero-fee revocation for every ticket that the block must revoke (`blockchain/stake/staketx.go · CreateRevocationFromTicket`), computing outputs with the same return amount algorithm.
- **Ticket exhaustion.** The template generator refuses to produce a block template whose ticket purchases are too few to keep at least `TicketsPerBlock` live tickets after the ticket maturity period, counting all immature tickets and the votes that will consume tickets (`internal/blockchain/validate.go · checkTicketExhaustion`). This is not a consensus rule; it prevents miners from producing a chain that Rule 8.25 would make impossible to extend.
- **Fee limits.** The fee limits are chosen by the wallet that builds the ticket. The tickets in dcrd's regression-network test data, such as the one in the vector of Section 8.3.3, set the revocation fee flag with $k_{\mathrm{rev}} = 24$ and leave the vote fee flag clear, so their votes pay no fee beyond the rounding remainder.
- **Stake tree order.** Apart from the treasurybase, which must come first when the treasury agenda is active, the order of stake transactions is left to the block producer; no order is required by consensus.

8.12 Error Kinds (informative)

dcrd reports each violation with an error kind. Table 8.17 maps the error kinds whose subject is a ticket, vote or revocation to the rules of this chapter, as a cross-check for implementers comparing behavior with dcrd. The kind reported for an invalid block depends on the order of dcrd's checks and is not part of consensus.

Table 8.17 Stake-related dcrd error kinds and the rules that cover them

Error kind(s)	Rule
`ErrSStxTooManyInputs`, `ErrSStxTooManyOutputs`, `ErrSStxNoOutputs`, `ErrSStxInvalidOutputs`, `ErrSStxInOutProportions`	Algorithm 8.1 (the transaction is not a ticket)
`ErrSSGenWrongNumInputs`, `ErrSSGenTooManyOutputs`, `ErrSSGenNoOutputs`, `ErrSSGenNoStakebase`, `ErrSSGenWrongIndex`, `ErrSSGenWrongTxTree`, `ErrSSGenBadGenOuts`, `ErrSSGenNoReference`, `ErrSSGenBadReference`, `ErrSSGenNoVotePush`, `ErrSSGenBadVotePush`, `ErrSSGenInvalidTxVersion`	Algorithm 8.2 (not a vote)
`ErrSSGenInvalidDiscriminatorLength`, `ErrSSGenInvalidNullScript`, `ErrSSGenUnknownDiscriminator`, `ErrSSGenInvalidTVLength`, `ErrSSGenInvalidTreasuryVote`, `ErrSSGenDuplicateTreasuryVote`	Algorithm 12.6 (not a vote)
`ErrSSRtxWrongNumInputs`, `ErrSSRtxTooManyOutputs`, `ErrSSRtxNoOutputs`, `ErrSSRtxBadOuts`, `ErrSSRtxWrongTxTree`, `ErrSSRtxInputHasSigScript`, `ErrSSRtxInvalidFee`	Algorithm 8.3 (not a revocation)
`ErrStakeTxInRegularTree`, `ErrRegTxInStakeTree`	Rule 8.34
`ErrNonstandardStakeTx`	Rule 8.35

Error kind(s)	Rule
`ErrNotEnoughStake`, `ErrStakeBelowMinimum`	Rule 8.6
`ErrTicketInputScript`; `ErrMissingTxOut` for a ticket input	Rule 8.4
`ErrTicketCommitment`	Rule 8.5
`ErrBadStakebaseScriptLen`, `ErrBadStakebaseScrVal`	Rule 8.8
`ErrBadTxInput` (null ticket input)	Rule 8.9
`ErrInvalidRevocationTxVersion`	Rule 8.10
`ErrScriptValidation` (revocation input)	Rule 8.11
`ErrInvalidVoteInput`, `ErrInvalidRevokeInput`	Rule 8.12, Rule 8.14
`ErrMissingTxOut` for a ticket input of a vote or revocation	Rule 8.13
`ErrImmatureTicketSpend`	Rule 8.15
`ErrBadNumPayees`	Rule 8.16, Rule 8.7
`ErrBadPayeeScriptVersion`, `ErrBadPayeeScriptType`, `ErrMismatchedPayeeHash`	Rule 8.17
`ErrBadPayeeValue`	Rule 8.18
`ErrBadStakebaseAmountIn`	Rule 8.19
`ErrSSGenSubsidy`, `ErrBadStakebaseValue`	Rule 8.20
`ErrTxSStxOutSpend`	Rule 8.21
`ErrImmatureSpend` for a stake-tagged output	Rule 8.22
`ErrMissingTxOut` for a same-block spend	Rule 8.23
`ErrUnknownTicketSpent`, `ErrMissingTicket`, `ErrDuplicateTicket`, and the untyped “cannot pick unique ticket indexes” error	Rule 8.25
`ErrInvalidEarlyStakeTx`	Rule 8.27, Rule 8.36
`ErrInvalidEarlyVoteBits`, `ErrInvalidEarlyFinalState`	Rule 8.27
`ErrNotEnoughVotes`	Rule 8.28
`ErrTooManyVotes`	Rule 8.29
`ErrTooManySStxs`	Rule 8.30
`ErrFreshStakeMismatch`	Rule 8.31
`ErrPoolSize`	Rule 8.32
`ErrInvalidFinalState`	Rule 8.33
`ErrNoStakeTx`	Rule 8.37
`ErrVotesMismatch`	Rule 8.38
`ErrTooManyRevocations`, `ErrRevocationsMismatch`	Rule 8.39
`ErrVotesOnWrongBlock`	Rule 8.40
`ErrIncongruentVotebit`	Rule 8.41
`ErrTicketUnavailable` (ineligible vote), `ErrInvalidSSRtx`, `ErrNoMissedTicketRevocation`, `ErrNoExpiredTicketRevocation`	Rule 8.42
`ErrTicketUnavailable` (in the stakebase total checks)	Unreachable: the ticket output was loaded by the earlier input checks.
`ErrSerializeHeader`	Unreachable: header serialization cannot fail.
`ErrTicketExhaustion`	Policy only (Section 8.11).
`ErrSStxBadCommitAmount`, `ErrSStxBadChangeAmts`, `ErrSSRtxInvalidTxVersion`, `ErrDatabaseCorrupt`, `ErrMissingDatabaseTx`, `ErrMemoryCorruption`	Not consensus: raised only by helper, wallet, mining or database code.

Error kind(s)	Rule
`ErrSStxInvalidInputs`, `ErrSStxVerifyCalcAmts`, `ErrVerSStxAmts`, `ErrVerifyInput`, `ErrVerifyOutType`, `ErrVerifyTooMuchFees`, `ErrVerifySpendTooMuch`, `ErrVerifyOutputAmt`, `ErrVerifyOutPkhs`, `ErrFindTicketIdxs`	Defined but never raised by dcrd at this commit.

CHAPTER 9

Proof of Work and Difficulty

Every Decred block header carries two difficulty values. The **difficulty bits** (header field `Bits`) encode the proof-of-work **target**: the hash of the header, computed with the network's proof-of-work hash function and read as a number, must not exceed it. The **stake difficulty** (header field `SBits`), also called the **ticket price**, is the minimum amount, in atoms, that a ticket purchased in the block must pay. Neither value is chosen by the miner. Each is a deterministic function of the chain that the block extends, and a block whose header states any other value is invalid.

This chapter specifies how the proof of work is checked, how targets are encoded, how the work of a chain is measured, and how both difficulty values are computed. Each of the three mechanisms has changed once by stakeholder vote, and both forms of each are normative because a node validates the whole chain from the genesis block:

- The proof-of-work hash was BLAKE-256 until the `blake3pow` agenda (DCP-0011 [34]) and is BLAKE3 afterwards.
- The proof-of-work target was retargeted every 144 blocks by an exponentially weighted moving average (here called the **EMA algorithm**) until the same agenda, and is computed for every block by ASERT afterwards.
- The stake difficulty was computed by a windowed moving average (**stake difficulty version 1**) until the `sdiffalgorithm` agenda (DCP-0001 [35]), and by the pool-size-based algorithm of that DCP (**version 2**) afterwards.

9.1 Overview and Selection of Algorithms

Table 9.1 lists the mechanisms and the condition that selects between them. "Active for the block" refers to the state of an agenda for the block being validated, computed from its ancestors as described in Chapter 13. Throughout this chapter, **the block** is the block being validated and **the parent** is the block it builds on; the parent always exists, because the genesis block is not validated (it is valid by definition, see Section 9.2.3).

Table 9.1 Difficulty mechanisms and their selection

Quantity	Original rule	Replacement	Selected by
Proof-of-work hash	BLAKE-256 of the header (identical to the block hash)	BLAKE3 of the header	`blake3pow` active for the block
`Bits`	EMA algorithm (Section 9.6)	ASERT (Section 9.7)	`blake3pow` active for the block
`SBits`	Version 1 (Section 9.9.1)	Version 2 (Section 9.9.2)	`sdiffalgorithm` active for the block

Table 9.2 summarizes how the two agendas resolved on each network defined by dcrd. An agenda that is **forced active** is active for every block, including block 1, without a vote. The activation heights restate the historical anchors of Table 13.8, which are part of the activity predicate (Rule 13.6): dcrd consults them before any tally of votes.

Table 9.2 Status of the difficulty-related agendas per network (informative)

Network	`blake3pow` (DCP-0011)	`sdiffalgorithm` (DCP-0001)
mainnet	Voted; active from block 794368	Voted; active from block 149248
testnet3	Voted; active from block 1170048	Forced active
simnet	Forced active	Forced active
regnet	Decided by vote on each regnet chain	Decided by vote on each regnet chain

The parameters used in this chapter are collected in Table 9.3. They are fields of the network parameters (Chapter 14); values are given for all four networks because the special cases below differ between them.

Table 9.3 Difficulty parameters

Parameter	mainnet	testnet3	simnet	regnet
`PowLimit`	$2^{224} - 1$	$2^{232} - 1$	$2^{255} - 1$	$2^{255} - 1$
`PowLimitBits`	`0x1d00ffff`	`0x1e00ffff`	`0x207fffff`	`0x207fffff`
`TargetTimePerBlock`	300 s	120 s	1 s	1 s
`WorkDiffWindowSize`	144	144	8	8
`WorkDiffWindows`	20	20	4	4
`WorkDiffAlpha`	1	1	1	1
`TargetTimespan`	43200 s	17280 s	8 s	8 s
`RetargetAdjustmentFactor`	4	4	4	4
`ReduceMinDifficulty`	false	true	false	false
`MinDiffReductionTime`	(unused)	600 s	(unused)	(unused)
`WorkDiffV2Blake3StartBits`	`0x1b00a5a6`	`0x1e00ffff`	`0x207fffff`	`0x207fffff`
`WorkDiffV2HalfLifeSecs`	43200	720	6	6
`MinimumStakeDiff`	200000000	20000000	20000	20000
`StakeDiffWindowSize`	144	144	8	8
`StakeDiffWindows`	20	20	8	8
`StakeDiffAlpha`	1	1	1	1
`TicketPoolSizeWeight`	4	4	4	4
`TicketPoolSize`	8192	1024	64	64
`TicketsPerBlock`	5	5	5	5
`TicketMaturity`	256	16	16	16
`CoinbaseMaturity`	256	16	16	16
`BaseSubsidy`	3119582664	2500000000	50000000000	50000000000
`MulSubsidy` / `DivSubsidy`	100 / 101	100 / 101	100 / 101	100 / 101
`SubsidyReductionInterval`	6144	2048	128	128
Block-one subsidy (sum of `BlockOneLedger`)	168000000000000	10000000000000	30000000000000	30000000000000
Genesis `Bits`	`0x1b01ffff`	`0x1e00ffff`	`0x207fffff`	`0x207fffff`
Genesis `SBits`	200000000	20000000	0	0

> **NOTE**
> `TargetTimespan` equals `WorkDiffWindowSize` × `TargetTimePerBlock` on every network, but the EMA algorithm uses the parameter as given and an implementation should not derive it. `RetargetAdjustmentFactor` is shared by the EMA algorithm and stake difficulty version 1. Time durations are used as whole seconds.

9.1.1 Notation

The pseudocode follows the conventions of Section 2.7 in Chapter 2. In addition:

- A block is represented by an object with the fields `height`, `timestamp`, `bits`, `sbits`, `pool_size` and `fresh_stake`, which are the header fields `Height`, `Timestamp`, `Bits`, `SBits`, `PoolSize` and `FreshStake` (Chapter 4), and `parent`, which is `None` for the genesis block. Timestamps are used as signed integers.
- `ancestor(b, h)` is the block at height `h` on the chain ending at `b` (`b` itself if `h = b.height`), or `None` if `h < 0`.
- `back(b, d)` is `ancestor(b, max(b.height - d, 0))`: the block `d` blocks before `b`, or the genesis block if `b` has fewer than `d` ancestors.
- `fdiv(a, b)` is **floor** division for `b > 0`. It is used where dcrd applies the Euclidean division of Go's `big.Int.Div` to a dividend that may be negative; for a positive divisor Euclidean division and floor division coincide. Where both operands are known to be non-negative, `//` is used.
- `>>` on a negative integer is an arithmetic shift, rounding toward negative infinity; dcrd's `big.Int.Rsh` behaves this way.
- `blake3pow_active_for(b)` is true if the `blake3pow` agenda is active for block `b`, and `blake3pow_active_for_child_of(p)` if it is active for a block whose parent is `p` (similarly for `sdiffalgorithm`). `blake3pow_forced_active` is true on a network where the agenda is forced active, and `is_testnet3` on testnet3.

9.2 The Proof-of-Work Check

9.2.1 Targets and the range rule

The target T of a block is obtained by decoding its `Bits` field with the compact decoding of Section 9.3. A valid target is a positive integer no greater than the network's `PowLimit`. The larger the target, the easier the block is to mine, so `PowLimit` defines the **minimum** difficulty.

> **RULE 9.1 Target range**
> Let $T = \text{compact_to_big}(\text{Bits})$ (Algorithm 9.1), evaluated with unbounded signed integers. The block **MUST** be rejected (`ErrUnexpectedDifficulty`) if $T \leq 0$ or $T >$ `PowLimit`. This check is context free and applies to every block other than the genesis block, independently of the proof-of-work hash check.
>
> dcrd: internal/blockchain/validate.go · checkProofOfWorkSanity
> blockchain/standalone/pow.go · checkProofOfWorkRange

A `Bits` value whose sign bit is set and whose mantissa decodes to a nonzero magnitude yields $T < 0$; one whose decoded magnitude is zero (for example `0x00000000`, `0x01003456`, or `0x01800000`) yields $T = 0$. Both are rejected by this rule.

9.2.2 The proof-of-work hash

The proof-of-work hash is computed over the serialized 180-byte block header (Chapter 4): the same byte string from which the block hash is computed. Two hash functions are used:

- **Version 1** is BLAKE-256 with 14 rounds [9], as specified in Chapter 3. Its output is exactly the block hash.
- **Version 2** is BLAKE3 [10] with a 32-byte output and no key (Chapter 3). It is used only for the proof of work. The block hash remains the BLAKE-256 hash in every role (identifying the block, linking it to its parent, merkle and commitment structures) after the change.

The 32-byte digest is interpreted as an unsigned 256-bit integer in **little-endian** byte order: byte 0 of the digest is the least significant byte. Read this way, the human-readable display form of a hash (Chapter 2) is the big-endian hexadecimal representation of the integer, which is why solved hashes are displayed with leading zeros.

> **RULE 9.2 Proof-of-work hash**
> Let H be the BLAKE3 hash of the serialized header if the `blake3pow` agenda is active for the block, and the BLAKE-256 hash of the serialized header otherwise. Let h be H read as a little-endian unsigned

256-bit integer and T the target of Rule 9.1. The block **MUST** be rejected (`ErrHighHash`) if $h > T$. A hash equal to the target is valid.

dcrd: `internal/blockchain/validate.go` · `checkProofOfWorkSanity`, `checkProofOfWorkContext`
`blockchain/standalone/pow.go` · `checkProofOfWorkHash`
`wire/blockheader.go` · `PowHashV1`, `PowHashV2`

NOTE
dcrd applies this rule in two stages. The context-free header check, which cannot evaluate the agenda, accepts a header whose BLAKE-256 hash **or** whose BLAKE3 hash satisfies the target (and rejects one for which neither does). The contextual check later requires the hash selected by the agenda, as stated in the rule. The combined effect is exactly Rule 9.2. A block that satisfies the target only with the other hash function is invalid, even though it passes the first stage.

NOTE
dcrd has a processing flag that skips the hash comparison (but not the range check) when it validates a block template that it built itself for mining. It has no effect on which blocks are valid.

9.2.3 The genesis block

The genesis block of each network is hard-coded and is not subjected to any of the rules in this chapter. Its `Bits`, `SBits` and `Timestamp` serve only as inputs to the difficulty calculations for its descendants. In particular the main-network genesis header does not satisfy its own target: its hash `298e5cc3d985bfe7f81dc135f360abe089edd4396b86d2de66b0cef42b21d980` (display form) is far above the target `0x1b01ffff` encodes.

TEST VECTOR BLAKE3 proof of work (testnet3 block 1170048)
This is the first BLAKE3 block of testnet3, from the reference vectors of DCP-0011, recomputed with dcrd. Hashes are in display form except where marked.

Item	Value
Serialized header (180 bytes)	`0b00000027d7ae5c9872a879b9a9da0b63915121d741e4cee3a96faea6eabf96b30000003271fd283c087219f14c7b26389fa1b22a639632d59e02e4f8e0456f36cbf740e9bba84160577d03ca9d858f07d6c78426421887cd6dc518d13fa06cfa000b5a01000b4e00a52a7705000400d4130000ffff001e9d7152e80000000080da1100e7100000085fb464f9a00f009cb73ed0429619d10000000000000000000000000000000000000000000000000000000000b000000`
`Bits`	`0x1e00ffff` (`PowLimitBits` of testnet3)
Target T	`000000ffff0000000000000000000000000000000000000000000000000000000000`
Block hash = BLAKE-256 PoW hash	`c7da7b548a2a9463dc97adb48433c4ffff18c3873f7e2ae99338a990dae039f0`
BLAKE3 PoW hash, digest bytes in order	`331e3590bd3988f55d1e69cc0163ff98e23824686ff19853e9db0530ce000000`
BLAKE3 PoW hash as integer h (= display form)	`000000ce3005dbe95398f16f682438e298ff6301cc691e5df58839bd90351e33`
Result	$h \leq T$ with BLAKE3: valid. The BLAKE-256 value exceeds T, so the header would fail Rule 9.2 before activation.

9.3 The Compact Target Encoding

Targets are 256-bit quantities but are carried in headers as a 32-bit **compact** value, a base-256 floating-point format inherited from Bitcoin. Bits 31 to 24 hold an unsigned exponent e, bit 23 a sign flag, and bits 22 to 0 an unsigned mantissa m. The encoded number is

$$N = (-1)^{\text{sign}} \cdot \left\lfloor m \cdot 256^{e-3} \right\rfloor \tag{9.1}$$

so that the exponent is the length in bytes of the magnitude and the mantissa holds its three most significant bytes. The floor matters only for $e < 3$, where low-order mantissa bytes are discarded.

ALGORITHM 9.1 Compact decoding (CompactToBig)

```
def compact_to_big(c: uint32) -> int:
    m = c & 0x007fffff
    negative = (c & 0x00800000) != 0
    e = c >> 24
    if e <= 3:
        n = m >> (8 * (3 - e))
    else:
        n = m << (8 * (e - 3))          # may exceed 256 bits; no overflow
    return -n if negative else n
```

dcrd: blockchain/standalone/pow.go · CompactToBig

Decoding is defined for every 32-bit value; the result can be negative, zero, or far larger than 2^{256} (up to $(2^{23} - 1) \cdot 2^{2016}$). An implementation that decodes into a fixed-width unsigned 256-bit type **MUST** still reject exactly the values Rule 9.1 rejects: those with the sign bit set and a nonzero decoded magnitude, those whose decoded magnitude is zero, and those whose value exceeds `PowLimit` (which includes every value that overflows 256 bits: a nonzero mantissa with exponent ≥ 35, exponent 34 with $m >$ `0xff`, or exponent 33 with $m >$ `0xffff`).

Encoding produces the canonical compact form of a non-negative integer by truncating it to its three most significant bytes. It is used by consensus code only on non-negative values (every computed target is positive).

ALGORITHM 9.2 Compact encoding (BigToCompact) of a non-negative integer

```
def big_to_compact(n: int) -> uint32:      # n >= 0
    if n == 0:
        return 0
    e = (n.bit_length() + 7) // 8          # length of n in bytes
    if e <= 3:
        m = n << (8 * (3 - e))
    else:
        m = n >> (8 * (e - 3))             # truncates: discards low bytes
    if m & 0x00800000:                     # would collide with the sign bit
        m = m >> 8
        e = e + 1
    return uint32((e << 24) | m)
```

dcrd: blockchain/standalone/pow.go · BigToCompact

The final normalization means a canonical mantissa never has bit 23 set; for $e > 3$ it lies in the range `0x008000` to `0x7fffff`. As a consequence, for canonical encodings of positive values the order of the 32-bit compact values as unsigned integers is the same as the order of the targets they encode, which some testnet rules rely on (Section 9.8).

IMPLEMENTATION PITFALL

The encoding is lossy and not unique. Many 32-bit values decode to the same target (`0x04001234` and `0x03123400` both decode to `0x123400`), and encoding then decoding a target generally yields a smaller number (`0x12345678` encodes to `0x04123456`, which decodes to `0x12345600`). Consensus compares **compact values**, never decoded targets: the `Bits` field must equal, as a 32-bit integer, the canonical encoding of the required target (Rule 9.3). An implementation that compares decoded targets instead would wrongly accept a header carrying a non-canonical encoding of the required target. Likewise, a computed target is always passed through `big_to_compact` before it is compared or reused; for example the decoded value of `PowLimitBits` is smaller than `PowLimit` on every network.

> **NOTE**
> dcrd's encoder also accepts negative inputs, setting the sign bit. For a negative input with more than three bytes it rounds the magnitude of the mantissa **up** rather than down (it shifts the signed value arithmetically), so `-0x1234567` encodes as `0x04812346`. No consensus computation encodes a negative value, so this has no effect on validity.

TEST VECTOR Compact decoding

Bits	Decoded value (hexadecimal)	Note
`0x1d00ffff`	`ffff` followed by 52 zero digits (= ffff · 2^{208})	mainnet `PowLimitBits`
`0x1b01ffff`	`1ffff` followed by 58 zero digits	mainnet genesis
`0x1b00a5a6`	`a5a6` followed by 58 zero digits	mainnet BLAKE3 start
`0x1e00ffff`	`ffff` followed by 54 zero digits	testnet3 limit
`0x207fffff`	`7fffff` followed by 58 zero digits	simnet/regnet limit
`0x00000000`, `0x00123456`, `0x01003456`	0	rejected
`0x01123456`	`0x12`	low bytes dropped
`0x02123456`	`0x1234`	
`0x03123456`	`0x123456`	
`0x04123456`	`0x12345600`	
`0x05009234`	`0x92340000`	
`0x04923456`	`-0x12345600`	sign bit; rejected
`0x01810000`	`-1`	sign bit; rejected
`0x01010000`	`1`	hardest valid target
`0x20123456`	123456 followed by 58 zero digits	above every `PowLimit`

TEST VECTOR Compact encoding

Input (hexadecimal)	Encoding	Decodes back to
`0`	`0x00000000`	`0`
`1`	`0x01010000`	`1`
`7f`	`0x017f0000`	`7f`
`80`	`0x02008000`	`80`
`ff`	`0x0200ff00`	`ff`
`1234`	`0x02123400`	`1234`
`123456`	`0x03123456`	`123456`
`800000`	`0x04008000`	`800000`
`12345678`	`0x04123456`	`12345600`
`ffffffff`	`0x0500ffff`	`ffff0000`
$2^{224} - 1$	`0x1d00ffff`	ffff · 2^{208}
$2^{232} - 1$	`0x1e00ffff`	ffff · 2^{216}
$2^{255} - 1$	`0x207fffff`	7fffff · 2^{232}
$2^{256} - 1$	`0x2100ffff`	ffff · 2^{240}
$2^{226} - 1$	`0x1d03ffff`	3ffff · 2^{208}

9.4 Work and Chain Work

The **work** of a block measures the expected number of hash evaluations needed to find a header that meets its target. It is computed from the `Bits` field alone:

ALGORITHM 9.3 Work of a block (CalcWork)

```
def calc_work(bits: uint32) -> int:
    t = compact_to_big(bits)
    if t <= 0:
        return 0
    return (1 << 256) // (t + 1)
```

dcrd: internal/staging/primitives/pow.go · CalcWork | blockchain/standalone/pow.go · CalcWork

For any target that overflows 256 bits the quotient is zero, so the result is zero for every `Bits` value outside the valid range except targets between `PowLimit` and $2^{256} - 1$, which Rule 9.1 excludes anyway. (dcrd's block index uses a fixed-width 256-bit implementation that returns zero for negative, zero, and overflowing values and is otherwise equal to this formula.)

The **cumulative work** (chain work) of a block is the sum of `calc_work` over the block and all of its ancestors, including the genesis block. It is the quantity by which competing chains are compared; chain selection is specified in Chapter 10.

TEST VECTOR Work

`Bits`	`calc_work` (hexadecimal)
`0x1d00ffff`	`100010001`
`0x1b01ffff`	`800040002000`
`0x1b00a5a6`	`18ba210ed6b21`
`0x1e00ffff`	`1000100`
`0x207fffff`	`2`
`0x01010000`	`8000000000000000000000000000000000000000000000000000000000000000`
`0x04923456`, `0x00000000`	`0`

9.5 Required Difficulty Bits

RULE 9.3 Difficulty bits

The `Bits` field of the block **MUST** equal, as a 32-bit unsigned integer, the value returned by `next_required_bits(parent, timestamp)` (Algorithm 9.4), where `parent` is the parent block and `timestamp` is the block's own `Timestamp` field. Otherwise the block **MUST** be rejected (`ErrUnexpectedDifficulty`).

dcrd: internal/blockchain/validate.go · checkBlockHeaderContext, checkDifficultyPositional
internal/blockchain/difficulty.go · calcNextRequiredDifficulty

ALGORITHM 9.4 Required difficulty bits

```
def next_required_bits(parent, timestamp: int64) -> uint32:
    if blake3pow_active_for_child_of(parent):
        return next_bits_asert(parent)              # Algorithm "ASERT required bits"
    return next_bits_ema(parent, timestamp)         # Algorithm "EMA required bits"
```

dcrd: internal/blockchain/difficulty.go · calcNextRequiredDifficulty

The block's own timestamp is an input only in the EMA algorithm, and only through the testnet minimum-difficulty rule (Section 9.8). In all other cases the required bits are a function of the parent chain alone.

> **NOTE**
> Before the full contextual checks, dcrd checks `Bits` while accepting every new header, with only the ancestor headers available, where the agenda state is not yet known in general. On a network where `blake3pow` is forced active that earlier check computes the ASERT result directly. Otherwise, if an anchor that dcrd has already determined from full block data is an ancestor of the header, it requires the ASERT result for that anchor. Failing that, it requires the EMA result, except that a header that is solved under BLAKE3, has a version of at least 10 (mainnet) or 11 (other networks), and is at a height of at least `StakeValidationHeight` $+2\times$ `RuleChangeActivationInterval` is also accepted if its `Bits` matches the ASERT result for a candidate anchor: the final block of an earlier rule change interval, tried from the most recent backwards while the candidate's version is at least that same minimum and its height is at least `StakeValidationHeight` $+2\times$ `RuleChangeActivationInterval` -1. This check never rejects a block that satisfies Rule 9.3, Rule 9.2 and the block version rule of Chapter 10: the `blake3pow` deployment has version 10 on mainnet and 11 on the other networks, it can leave the defined state only once a majority of recent blocks has at least that version (Chapter 13), and from then on the block version rule rejects every block with a lower version. A conforming implementation needs only Rule 9.3.

When dcrd connects the full data of a block that is an ancestor of its assumed-valid block (Chapter 10), or runs in bulk-import mode, it skips this comparison together with the stake difficulty comparison of Rule 9.6 (the header-acceptance check of the note above still applies to `Bits`); this is an optimization that presumes those blocks satisfy the rules, not a relaxation of them.

9.6 The EMA Difficulty Algorithm

The EMA algorithm is the original proof-of-work retargeting rule. It applies to every block for which `blake3pow` is not active: on mainnet, blocks 1 to 794367.

The target changes only at **retarget blocks**, those whose height is a multiple of `WorkDiffWindowSize` (144 on mainnet); every other block must repeat its parent's `Bits` (except under the testnet rule of Section 9.8). At a retarget block, the algorithm looks back over `WorkDiffWindows` (20) consecutive **windows** of `WorkDiffWindowSize` blocks each, ending at the parent. For each window it measures the time the window took relative to `TargetTimespan`, as a 32.32 fixed-point ratio. The ratios are averaged with weights that double from each window to the next more recent one, and the old target is multiplied by the average. The result is clamped to between one quarter and four times the old target (`RetargetAdjustmentFactor` $= 4$) and to at most `PowLimit`.

9.6.1 Windows

Let the block be at height H, a multiple of $W =$ `WorkDiffWindowSize`, so the parent P is at height $H-1$. Window k (for $k = 0, \ldots, N-1$, $N =$ `WorkDiffWindows`) is delimited by the two blocks `back(P, k W)` (its newer end) and `back(P, (k+1) W)` (its older end). Its time difference is the difference of their timestamps, except that **if the older end is the genesis block** the window is taken to have lasted exactly `TargetTimespan`. With `back` clamping at the genesis block, this covers every window that reaches back to or beyond the genesis block, including a window whose newer end is itself the genesis block. Window k has weight $2^{(N-k)\alpha}$ with $\alpha =$ `WorkDiffAlpha`, so with mainnet parameters window 0 (the most recent) has weight 2^{20} and window 19 has weight 2^1; the weights sum to $2^{21}-2$.

Table 9.4 EMA windows for the mainnet retarget block at height $H = 144m$

Window k	Newer end (height)	Older end (height)	Weight
0	$144m-1$	$144(m-1)-1$	2^{20}
1	$144(m-1)-1$	$144(m-2)-1$	2^{19}
…	…	…	…
19	$144(m-19)-1$	$144(m-20)-1$	2^1

Heights below 0 in Table 9.4 mean the genesis block. A window's time difference may be negative if timestamps decrease; the arithmetic below is defined for that case and uses floor division.

9.6.2 Algorithm

ALGORITHM 9.5 EMA required bits

```
TESTNET3_MAX_DIFF_HEIGHT = 962928

def next_bits_ema(parent, timestamp: int64) -> uint32:
    W, N, ALPHA = WorkDiffWindowSize, WorkDiffWindows, WorkDiffAlpha
    SPAN = TargetTimespan                       # in seconds
    RAF = RetargetAdjustmentFactor
    next_height = parent.height + 1

    if next_height % W != 0:                    # not a retarget block
        if ReduceMinDifficulty and (not is_testnet3 or
                                    next_height < TESTNET3_MAX_DIFF_HEIGHT):
            if timestamp > parent.timestamp + MinDiffReductionTime:
                return PowLimitBits
            return last_non_min_diff_bits(parent)
        return parent.bits

    old = compact_to_big(parent.bits)
    lower = fdiv(old, RAF)
    upper = old * RAF

    weighted_sum = 0
    weights = 0
    for k in range(N):                          # k = 0 is the most recent window
        newer = back(parent, k * W)
        older = back(parent, (k + 1) * W)
        elapsed = newer.timestamp - older.timestamp
        if older.height == 0:
            elapsed = SPAN
        ratio = fdiv(elapsed << 32, SPAN)       # 32.32 fixed point, may be < 0
        weighted_sum += ratio << ((N - k) * ALPHA)
        weights += 1 << ((N - k) * ALPHA)

    target = (fdiv(weighted_sum, weights) * old) >> 32

    if old == 0:
        pass                                    # unreachable for valid chains
    elif target == 0:
        target = PowLimit
    elif target > upper:
        target = upper
    elif target < lower:
        target = lower
    if target > PowLimit:
        target = PowLimit
    if is_testnet3 and target < MIN_TESTNET_TARGET and \
            next_height >= TESTNET3_MAX_DIFF_HEIGHT:
        target = MIN_TESTNET_TARGET             # 2**226 - 1, see testnet rules
    return big_to_compact(target)

def last_non_min_diff_bits(b) -> uint32:
    period = WorkDiffWindowSize * WorkDiffWindows
    while b is not None and b.height % period != 0 and b.bits == PowLimitBits:
        b = b.parent
    return b.bits if b is not None else PowLimitBits
```

dcrd: internal/blockchain/difficulty.go · calcNextBlake256Diff, findPrevTestNetDifficulty

The order of operations is significant and must be reproduced exactly: each window ratio is floored before it is weighted; the weighted sum is floor-divided by the sum of weights **before** multiplication by the old target; and the product is shifted right by 32 (flooring) only at the end. The clamping is a chain of alternatives evaluated in the order shown, of which at most one applies, followed by the `PowLimit` clamp. A computed target of exactly zero (when the floored average ratio times the old target is below 2^{32}) becomes `PowLimit`, whereas a negative computed target (possible only with decreasing timestamps) falls into the `lower` branch.

IMPLEMENTATION PITFALL
Every intermediate quantity is unbounded: `elapsed << 32` exceeds 64 bits for elapsed times above 2^{31} seconds, and the weighted terms grow with the window weights. An implementation using 64-bit integers for the ratio sum overflows on realistic inputs. Division of a negative weighted sum must floor (not truncate).

IMPLEMENTATION PITFALL
The retarget condition uses `WorkDiffWindowSize` alone (retargets every 144 blocks), while `last_non_min_diff_bits` stops at multiples of `WorkDiffWindowSize` × `WorkDiffWindows` (2880 blocks). This asymmetry is part of the testnet rule as implemented.

RATIONALE
Each window ratio is the factor by which the target would need to change to make that window last exactly `TargetTimespan` at the same hash rate. Weighting recent windows more heavily makes the estimate an exponentially weighted moving average over roughly the last 2880 blocks (10 days on mainnet), smoothing short-term hash-rate fluctuations while still tracking trends. Treating windows that extend before the genesis block as on-target lets the algorithm run unchanged from block 144.

TEST VECTOR EMA retargets with mainnet parameters
Chains of `n` blocks after the mainnet genesis block (timestamp 1454954400), with the stated block spacing and every block carrying the stated `Bits`. The result is `next_bits_ema` for the block at height `n + 1`, which is a retarget block in every case. For each case the block at height `n` (not a retarget block) required the parent's `Bits` unchanged.

`n`	Block spacing	Parent `Bits`	Required `Bits`
2879	300 s	`0x1b01ffff`	`0x1b01ffff`
2879	600 s	`0x1b01ffff`	`0x1b03fffd`
5759	600 s	`0x1b01ffff`	`0x1b03fffe`
5759	150 s	`0x1b01ffff`	`0x1b00ffff`
5759	60 s	`0x1b01ffff`	`0x1a7fffc0` (clamped to old/4)
5759	3000 s	`0x1b01ffff`	`0x1b07fffc` (clamped to old × 4)
143	150 s	`0x1b01ffff`	`0x1b01ffff` (all windows reach genesis)
287	150 s	`0x1b01ffff`	`0x1b017fff`
5759	300 s; 600 s for the last 144 blocks	`0x1b01ffff`	`0x1b02fffe`
5759	600 s for blocks 1 to 3023; 300 s after	`0x1b01ffff`	`0x1b01ffff`
5759	$200 + 50 \cdot (h \bmod 7)$ s before block h	`0x1a0f00ff`	`0x1a1184d6`

In the 600 s case with `n` = 2879, windows 0 to 18 each lasted 86400 s (ratio $2 \cdot 2^{32}$) and window 19 reaches the genesis block and counts as on target (ratio 2^{32}). The weighted sum is 18014372739678208; divided by the weight sum $2^{21} - 2$ it gives 8589930495 (`0x1fffefff`), slightly below $2 \cdot 2^{32}$; multiplying by the old target `0x1ffff` $\cdot 2^{192}$ and shifting right by 32 gives `3fffddffe100100000000000000000000000000000000000000000`, which encodes to `0x1b03fffd`.

9.7 ASERT

Once `blake3pow` is active, the required target is computed for every block by ASERT ("absolutely scheduled exponentially rising targets", [36], DCP-0011). ASERT fixes an **anchor block** and an ideal schedule from it: one block every `TargetTimePerBlock` seconds. If the parent is Δt seconds after the anchor and Δh blocks above it, it is $\Delta t - \Delta h \cdot$ `TargetTimePerBlock` seconds behind schedule, and the target is set to the starting target `WorkDiffV2Blake3StartBits` multiplied by 2 raised to that delay divided by the half-life `WorkDiffV2HalfLifeSecs`:

$$T = \min\left(\max\left(T_s \cdot 2^{(\Delta t - \Delta h \cdot I_b)/\tau}, 1\right), \text{PowLimit}\right) \tag{9.2}$$

with T_s the decoded starting target, $I_b =$ `TargetTimePerBlock` and $\tau =$ `WorkDiffV2HalfLifeSecs`. The target doubles (difficulty halves) for every half-life the chain is behind schedule and halves for every half-life it is ahead. Eq. (9.2) is the design goal; the normative computation is the integer approximation of Algorithm 9.6, which uses a 16-bit fixed-point exponent and a cubic polynomial for the fractional power of two.

9.7.1 The anchor block and the difficulty reset

The anchor is determined as follows:

- **If `blake3pow` was activated by a vote** (mainnet, testnet3, and regnet once voted): the anchor is the last block, on the chain of the block being validated, for which `blake3pow` is **not** active; equivalently, the parent of the first block for which it is active. Because an agenda's state changes only at the start of a rule change interval (Chapter 13), the anchor is the final block of a rule change interval. The first BLAKE3 block is the block immediately after the anchor; its parent is the anchor itself, so $\Delta t = \Delta h = 0$ and its required bits are exactly `WorkDiffV2Blake3StartBits` (see the vectors). The proof-of-work hash switches to BLAKE3 at the same block (Rule 9.2).
- **If `blake3pow` is forced active** (simnet): block 1 must carry `WorkDiffV2Blake3StartBits`, and block 1 is the anchor for every later block. The genesis block is never the anchor, because its hard-coded timestamp is typically far in the past when mining starts.

Table 9.5 ASERT anchors of the networks where the agenda was voted (informative)

Network	Anchor height	Anchor block hash (display form)
mainnet	794367	`0000000000000000c293d8c67409d05e960447ea25cdaf770e864d995c764ef0`
testnet3	1170047	`000000b396bfeaa6ae6fa9e3cee441d7215191630bdaa9b979a872985caed727`

Note that the anchor's own `Bits` are not used: the difficulty is **reset** to `WorkDiffV2Blake3StartBits` at activation, because a target appropriate for specialized BLAKE-256 hardware would be unattainable with BLAKE3. Only the anchor's height and timestamp enter the calculation.

9.7.2 Algorithm

ALGORITHM 9.6 **ASERT target calculation (CalcASERTDiff)**

```
POLY_C1 = 195766423245049     # ceil(0.695502049712533 * 2**48)
POLY_C2 = 971821376           # ceil(0.2262697964 * 2**32)
POLY_C3 = 5127                # ceil(0.0782318 * 2**16)

def asert(start_bits: uint32, pow_limit: int, block_secs: int,
          dt: int, dh: int, half_life: int) -> uint32:
    start = compact_to_big(start_bits)
    assert 0 < start <= pow_limit and dh >= 0       # guaranteed by callers

    # Exponent in 16.16 fixed point, truncated toward zero.
    x = tdiv((dt - dh * block_secs) << 16, half_life)
    n = x >> 16                  # integer part: arithmetic shift (floor)
```

```
    f = x & 0xffff               # fractional part, 0 <= f < 2**16

    # 2**16 * 2**(f / 2**16), cubic approximation, 16.48 fixed point inside.
    factor = (1 << 16) + ((POLY_C1 * f + POLY_C2 * f * f +
                           POLY_C3 * f * f * f + (1 << 47)) >> 48)

    target = start * factor
    s = n - 16
    if s >= 0:
        target = target << s
    else:
        target = target >> -s    # floor
    if target == 0:
        target = 1
    elif target > pow_limit:
        target = pow_limit
    return big_to_compact(target)
```

dcrd: blockchain/standalone/pow.go · CalcASERTDiff

ALGORITHM 9.7 **ASERT required bits**

```
MIN_TESTNET_BITS = 0x1d03ffff            # big_to_compact(MIN_TESTNET_TARGET)

def next_bits_asert(parent) -> uint32:
    if blake3pow_forced_active:
        if parent.height == 0:
            return WorkDiffV2Blake3StartBits
        anchor = ancestor(parent, 1)
    else:
        # Last block on this chain for which blake3pow is not active.
        anchor = parent
        while blake3pow_active_for(anchor):
            anchor = anchor.parent
    dt = parent.timestamp - anchor.timestamp      # may be negative
    dh = parent.height - anchor.height            # >= 0
    bits = asert(WorkDiffV2Blake3StartBits, PowLimit, TargetTimePerBlock,
                 dt, dh, WorkDiffV2HalfLifeSecs)
    if is_testnet3 and bits < MIN_TESTNET_BITS:   # unsigned 32-bit comparison
        bits = MIN_TESTNET_BITS
    return bits
```

dcrd: internal/blockchain/difficulty.go · calcNextBlake3Diff, calcNextBlake3DiffFromAnchor, blake3WorkDiffAnchor

dcrd locates the anchor by stepping back one rule change interval at a time from the final block of the interval preceding the block, testing only those interval-final blocks, and caches the result; because the agenda state can change only at interval boundaries, the block-by-block loop in Algorithm 9.7 finds the same block. If `blake3pow` is active for the block but not for its parent, the anchor is the parent. The loop always terminates before the genesis block, since an agenda activated by vote is never active for block 1.

IMPLEMENTATION PITFALL

The exponent uses **truncating** division (toward zero), but its integer part uses **flooring** (an arithmetic shift) and its fractional part is the non-negative remainder modulo 2^{16}, so that $x = 2^{16}n + f$ exactly. For example $\Delta t = 299$, $\Delta h = 1$ on mainnet gives $x = \mathrm{tdiv}(-65536, 43200) = -1$, hence $n = -1$ and $f = 65535$; floor division would give $x = -2$ and a different result. A logical (unsigned) shift for n is also wrong.

IMPLEMENTATION PITFALL

The polynomial numerator needs the full unsigned 64-bit range: at $f = 65535$ it equals 18446563080438344768, just below 2^{64}. It is always positive and the resulting factor is at most 17 bits ($2^{16} \le \text{factor} < 2^{17}$). In the

final step the shift amount $n - 16$ is unbounded in principle; for consensus inputs ($\Delta h < 2^{32}$, timestamps 32-bit) the exponent stays far inside the 64-bit range, but the target before clamping can exceed 256 bits (on testnet3, whose half-life is 720 seconds, by more than a thousand bits for a chain stalled for two weeks). An implementation may clamp early, provided the result is identical: any left shift by at least the bit length of `PowLimit` yields `PowLimit`, and any right shift by at least the bit length of `start * factor` yields 1.

IMPLEMENTATION PITFALL
The testnet3 lower bound in Algorithm 9.7 compares the 32-bit compact values, not targets. Because `asert` returns a canonical encoding of a positive target, this ordering agrees with the ordering of the targets (Section 9.3).

RATIONALE
An absolutely scheduled algorithm measures the cumulative deviation from the ideal schedule since a fixed point, so errors do not accumulate and there are no windows whose boundaries a miner could exploit. Using a fixed anchor makes each target an exact function of two numbers, which is cheaper to compute and to verify than a windowed average. The polynomial approximates 2^x on $[0, 1)$ with a relative error below 0.013%, and integer arithmetic keeps the result identical across implementations.

TEST VECTOR ASERT with mainnet parameters
Start bits `0x1b00a5a6`, `PowLimit` $= 2^{224} - 1$, block time 300, half-life 43200. `dt` and `dh` are the deltas of the parent from the anchor.

dt	dh	Result	Comment
0	0	`0x1b00a5a6`	first BLAKE3 block
300	1	`0x1b00a5a6`	on schedule
299	1	`0x1b00a5a5`	$x = -1$, $n = -1$, $f = 65535$, factor 131071
301	1	`0x1b00a5a6`	$x = 1$, $n = 0$, $f = 1$, factor 65537
1	0	`0x1b00a5a6`	
−1	0	`0x1b00a5a5`	
43200	0	`0x1b014b4c`	one half-life behind: target doubles
−43200	0	`0x1a52d300`	one half-life ahead: target halves
21600	0	`0x1b00ea3e`	$x = 32768$, factor 92674
−21600	0	`0x1a751f03`	$x = -32768$, $n = -1$, $f = 32768$
86400	144	`0x1b014b4c`	
86400	0	`0x1b029698`	
0	144	`0x1a52d300`	
0	288	`0x1a296980`	
172800	288	`0x1b029698`	
864000	0	`0x1d00ffff`	clamped to `PowLimit`
−1728000	0	`0x1600a5a6`	40 half-lives ahead
−4320000	0	`0x0e0a5a60`	100 half-lives ahead
−8942400	0	`0x01010000`	207 half-lives ahead: the shift leaves exactly 1
−8985600	0	`0x01010000`	208 half-lives ahead: the shift leaves 0, clamped to 1

TEST VECTOR ASERT reference vectors of DCP-0011 (excerpt)
From the test vector file shipped with dcrd and DCP-0011; mainnet parameters, anchor height 0 and anchor time as stated. dcrd reproduces all of them.

Start bits	Anchor time	Height	Timestamp	Result
`0x1d00ffff`	0	1	691200	`0x1d00ffff`
`0x1d00ffff`	0	6	11793600	`0x1d00ffff`
`0x1b00a5a6`	0	4	1199	`0x1b00a5a5`
`0x1b00a5a6`	300	1	299	`0x1b00a4da`
`0x1b00a5a6`	300	6	294	`0x1b00a0ed`
`0x01010000`	0	1	43500	`0x01020000`
`0x01010000`	0	7	304500	`0x02008000`

TEST VECTOR ASERT with testnet3 parameters and the testnet3 bound
Start bits `0x1e00ffff`, `PowLimit` $= 2^{232} - 1$, block time 120, half-life 720.

`dt`	`dh`	`asert` result	Required bits
0	0	`0x1e00ffff`	`0x1e00ffff`
1200	10	`0x1e00ffff`	`0x1e00ffff`
600	10	`0x1e008faf`	`0x1e008faf`
−720	0	`0x1d7fff80`	`0x1d7fff80`
−7200	0	`0x1c3fffc0`	`0x1d03ffff` (testnet3 bound)

9.8 Test Network Rules

Version 3 of the test network (testnet3) has rules that keep it usable with ordinary hardware. They are selected by the network identity (the `TestNet3` network magic), not by a parameter, except for the minimum-difficulty rule, which is selected by `ReduceMinDifficulty` and limited on testnet3 by height. On testnet3, `MIN_TESTNET_TARGET` is `PowLimit` shifted right by 6 bits, $2^{226} - 1$ (a maximum difficulty of $2^6 = 64$ relative to the limit), and `MIN_TESTNET_BITS` = `big_to_compact(MIN_TESTNET_TARGET)` = `0x1d03ffff`. The threshold height is 962928.

Minimum-difficulty blocks On a network with `ReduceMinDifficulty` set (only testnet3 among dcrd's networks), for a block that is not an EMA retarget block and, on testnet3, whose height is below 962928: if its timestamp is more than `MinDiffReductionTime` (600 s) after its parent's timestamp, the required bits are `PowLimitBits`; otherwise they are the bits of the most recent ancestor (starting with the parent) that does not carry `PowLimitBits`, stopping at the first block whose height is a multiple of `WorkDiffWindowSize` × `WorkDiffWindows` (Algorithm 9.5). Retarget blocks are never minimum-difficulty blocks, and the retarget calculation uses the parent's `Bits` as the old target even if it is `PowLimitBits`. The rule never applies to ASERT.

Maximum difficulty On testnet3, from height 962928 on, an EMA retarget result below `MIN_TESTNET_TARGET` is replaced by `MIN_TESTNET_TARGET` before encoding, and an ASERT result whose compact value is below `MIN_TESTNET_BITS` is replaced by `MIN_TESTNET_BITS` (at all heights; ASERT begins above 962928 on testnet3). Both forms produce the same compact result for every target, because every target between the decoded value of `0x1d03ffff` and $2^{226} - 1$ encodes to `0x1d03ffff`.

Block spacing and checkpoint Rule 9.4 and Rule 9.5 below.

RULE 9.4 Testnet3 maximum-difficulty checkpoint
On testnet3, a block at height 962928 whose block hash is not the hash below (display form) **MUST** be rejected (`ErrBadMaxDiffCheckpoint`): `0000004fd1b267fd39111d456ff557137824538e6f6776168600e56002e23b93`

dcrd: internal/blockchain/validate.go · checkBlockHeaderPositional

RULE 9.5 Testnet3 minimum block spacing at maximum difficulty
On testnet3, a block at height 962928 or above whose `Bits`, compared as an unsigned 32-bit integer, is less than or equal to `0x1d03ffff` **MUST** have a timestamp at least 60 seconds after its parent's timestamp (equality allowed); otherwise it **MUST** be rejected (`ErrTimeTooOld`).

dcrd: internal/blockchain/validate.go · checkBlockHeaderPositional

HISTORY
The testnet3 maximum-difficulty rules were introduced at height 962928. dcrd invalidates, at startup, any stored testnet3 chain whose block at that height is not the checkpoint block, which has the same effect as Rule 9.4 for chains received before the rule existed.

9.9 Stake Difficulty

The stake difficulty is the ticket price: the minimum value, in atoms, of the stake submission output (output 0) of every ticket purchase in a block. It is carried in the header field `SBits`, an `int64`. Like the proof-of-work target it is recomputed only at **stake retarget blocks**, those whose height is a multiple of `StakeDiffWindowSize` (144 blocks on mainnet), and every other block repeats its parent's value. The ticket structure, the ticket pool and the header fields `PoolSize` (the number of live tickets) and `FreshStake` (the number of ticket purchases in the block) are specified in Chapter 8.

RULE 9.6 Stake difficulty
The `SBits` field of the block **MUST** equal `next_required_sbits(parent)` (Algorithm 9.8), where `parent` is the parent block. Otherwise the block **MUST** be rejected (`ErrUnexpectedDifficulty`).

dcrd: internal/blockchain/validate.go · checkBlockHeaderContext
internal/blockchain/difficulty.go · calcNextRequiredStakeDifficulty

The `SBits` field is in turn the minimum price of every ticket in the block: Rule 8.6 requires output 0 of each ticket purchase in the stake tree to be worth at least the block's `SBits` and at least `MinimumStakeDiff`.

ALGORITHM 9.8 Required stake difficulty

```
def next_required_sbits(parent) -> int64:
    if sdiffalgorithm_active_for_child_of(parent):
        return next_sbits_v2(parent)
    return next_sbits_v1(parent)
```

dcrd: internal/blockchain/difficulty.go · calcNextRequiredStakeDifficulty

Both versions share the retarget schedule and carry the parent's value forward between retargets. They differ by one block in their start-up rule: version 1 returns `MinimumStakeDiff` whenever the **parent's** height is at most `CoinbaseMaturity`, version 2 whenever the **block's** height is at most `CoinbaseMaturity`. Before `CoinbaseMaturity` +1 no output exists that could fund a ticket.

NOTE
On mainnet the `sdiffalgorithm` agenda became active at block 149248, which is not a stake retarget block ($149248 = 1036 \times 144 + 64$). Blocks 149248 to 149327 therefore carry forward the value computed by version 1 for block 149184, and block 149328 is the first to be computed by version 2, from that carried-forward value. No other transition step exists.

9.9.1 Version 1

Version 1 computes two independent estimates of the new price, each a weighted moving average over `StakeDiffWindows` windows of `StakeDiffWindowSize` blocks in the style of the EMA algorithm, and combines them:

- The **pool-size estimate** compares the live ticket pool size, sampled once per window, with the target pool size TicketsPerBlock × TicketPoolSize (40960 on mainnet). Deviations are amplified by `TicketPoolSizeWeight`.
- The **fresh-stake estimate** compares the number of tickets bought in each window with StakeDiffWindowSize × TicketsPerBlock (720 on mainnet).

Each estimate is clamped to a factor of `RetargetAdjustmentFactor` around the old price, the two are combined multiplicatively, the result is clamped again, and finally raised to at least `MinimumStakeDiff`.

For a stake retarget block at height $H = 144m$ with mainnet parameters, the pool-size estimate uses the `PoolSize` of the blocks at heights $144(m-1-k)$ for $k = 0, \ldots, 19$ (the first block of each of the last 20 intervals), and the fresh-stake estimate sums `FreshStake` over the complete intervals $[144(m-1-k), 144(m-k)-1]$. Window k has weight $2^{(20-k)\alpha}$ with $\alpha =$ `StakeDiffAlpha` $= 1$. Blocks before the genesis block are replaced by the genesis block (whose `PoolSize` and `FreshStake` are 0 on every network); in the fresh-stake sum, a window extending before the genesis block therefore counts the genesis block repeatedly.

ALGORITHM 9.9 Stake difficulty version 1

```
def next_sbits_v1(parent) -> int64:
    SW, SN, ALPHA = StakeDiffWindowSize, StakeDiffWindows, StakeDiffAlpha
    RAF = RetargetAdjustmentFactor
    if parent.height < CoinbaseMaturity + 1:
        return MinimumStakeDiff
    old = parent.sbits
    if (parent.height + 1) % SW != 0:
        return old
    if old == 0:
        return 0                                  # unreachable for valid chains

    # Pool-size estimate.
    target_pool = TicketsPerBlock * TicketPoolSize
    weighted_sum, weights = 0, 0
    for k in range(SN):
        b = back(parent, (k + 1) * SW - 1)
        skew = (b.pool_size - target_pool) * TicketPoolSizeWeight + target_pool
        if skew <= 0:
            skew = 1
        ratio = (skew << 32) // target_pool
        weighted_sum += ratio << ((SN - k) * ALPHA)
        weights += 1 << ((SN - k) * ALPHA)
    by_pool = int64(((weighted_sum // weights) * old) >> 32)
    by_pool = clamp_v1(old, by_pool)

    # Fresh-stake estimate.
    target_fresh = SW * TicketsPerBlock
    weighted_sum, weights = 0, 0
    for k in range(SN):
        fresh = 0
        for d in range(k * SW, (k + 1) * SW):
            fresh += back(parent, d).fresh_stake
        if fresh <= 0:
            fresh = 1
        ratio = (fresh << 32) // target_fresh
        weighted_sum += ratio << ((SN - k) * ALPHA)
        weights += 1 << ((SN - k) * ALPHA)
    by_fresh = int64(((weighted_sum // weights) * old) >> 32)
    by_fresh = clamp_v1(old, by_fresh)

    nxt = clamp_v1(old, merge(old, by_pool, by_fresh))
    if nxt < MinimumStakeDiff:
        return MinimumStakeDiff
```

```
    return nxt

def clamp_v1(old: int64, x: int64) -> int64:   # old > 0, x >= 0
    if x == 0:
        return old // RetargetAdjustmentFactor
    if x // old > RetargetAdjustmentFactor - 1:
        return old * RetargetAdjustmentFactor
    if old // x > RetargetAdjustmentFactor - 1:
        return old // RetargetAdjustmentFactor
    return x

def merge(old: int64, d1: int64, d2: int64) -> int64:   # all > 0
    a = (old << 32) // d1
    b = (d2 << 32) // old
    return int64((((b << 32) // a) * old) >> 32)
```

dcrd: internal/blockchain/difficulty.go · calcNextRequiredStakeDifficultyV1, mergeDifficulty

`merge` computes approximately $d_1 d_2 /$ old: the product of the two estimated change factors applied to the old price. All divisions in version 1 are of non-negative operands. The `int64` conversions take the value modulo 2^{64} as two's complement; for every reachable input the values fit, so the conversion has no effect. (Wrapping requires a price far above the total supply. dcrd also evaluates `clamp_v1` in 64-bit arithmetic, and the pseudocode above does not describe its results in that unreachable range, where the estimates can wrap to negative values.) The clamp conditions are integer comparisons of truncated quotients: `x // old > RAF - 1` holds exactly when $x \geq \mathrm{RAF} \cdot \mathrm{old}$, and `old // x > RAF - 1` exactly when $\mathrm{old} \geq \mathrm{RAF} \cdot x$.

IMPLEMENTATION PITFALL
The early return for `old == 0` reproduces dcrd, which returns the (necessarily zero) pool-size estimate when the old price is zero. `merge` divides by `d1`, `old` and `a`; these are nonzero because a valid chain's price is at least `MinimumStakeDiff`, which exceeds `RetargetAdjustmentFactor` on every network. dcrd would fail on a zero divisor rather than reject the block.

TEST VECTOR Stake difficulty version 1, mainnet parameters
Chains of 5759 blocks in which every block has the stated `PoolSize` and `FreshStake` and `SBits` 5000000000; the result is the required `SBits` of block 5760.

PoolSize	FreshStake	Result	Comment
40960	5	5000000000	both estimates on target
45056	5	6999999999	skew $= 4096 \times 4 + 40960$: factor 1.4
36864	5	2999999998	factor 0.6
40960	20	20000000000	fresh estimate clamped to $\times 4$
40960	0	1250000000	fresh estimate clamped to $\div 4$
0	0	1250000000	
41779	10	10799804684	

`merge(5000000000, 6000000000, 4000000000)` $= 4799999998$;
`merge(200000000, 800000000, 800000000)` $= 3200000000$.

9.9.2 Version 2 (DCP-0001)

Version 2 adjusts the price in proportion to two ratios: the relative change of the pool size (including immature tickets) since the previous retarget, and the ratio of the current pool size to its target. With S the current price, P_N and P_{N-1} the current and previous pool sizes including immature tickets, and $T_P = \mathrm{TicketsPerBlock} \times (\mathrm{TicketPoolSize} + \mathrm{TicketMaturity})$ (42240 on mainnet):

$$S' = \max\left(\min\left(\left\lfloor \frac{S \cdot P_N^2}{P_{N-1} \cdot T_P} \right\rfloor, S_{\text{ub}}\right), S_{\text{lb}}\right) \tag{9.3}$$

where $S_{\text{lb}} =$ `MinimumStakeDiff` and S_{ub} is an estimate of the total supply divided by `TicketPoolSize`. The normative computation, including the order in which the bounds are applied, is Algorithm 9.10.

The pool size including immature tickets "as of" a block b is b.pool_size plus the `FreshStake` of b and of its `TicketMaturity` -1 nearest ancestors (`TicketMaturity` blocks in total, fewer if the genesis block is reached). For a retarget block at height H, the current value is taken as of the parent (height $H-1$) and the previous value as of the block at height $H - \text{StakeDiffWindowSize} - 1$, the parent of the previous retarget block. The `PoolSize` header field counts live tickets only and does not yet include tickets that mature in that very block (Chapter 8); the formula uses the header values as they are.

ALGORITHM 9.10 Stake difficulty version 2

```
def next_sbits_v2(parent) -> int64:
    next_height = parent.height + 1
    if next_height < CoinbaseMaturity + 1:
        return MinimumStakeDiff
    cur = parent.sbits
    if next_height % StakeDiffWindowSize != 0:
        return cur

    prev_node = ancestor(parent, next_height - StakeDiffWindowSize - 1)
    prev_all = 0
    if prev_node is not None:
        prev_all = prev_node.pool_size + sum_fresh(prev_node, TicketMaturity)
    if prev_all == 0:
        return cur                      # avoids division by zero at start-up
    cur_all = parent.pool_size + sum_fresh(parent, TicketMaturity)

    target_all = TicketsPerBlock * (TicketPoolSize + TicketMaturity)
    nxt = int64((cur * cur_all * cur_all) // prev_all // target_all)
    maximum = estimate_supply(next_height) // TicketPoolSize
    if nxt > maximum:
        nxt = maximum
    if nxt < MinimumStakeDiff:
        nxt = MinimumStakeDiff
    return nxt

def sum_fresh(b, count: int) -> int:
    total = 0
    while b is not None and count > 0:
        total += b.fresh_stake
        b = b.parent
        count -= 1
    return total

def estimate_supply(height: int64) -> int64:
    if height <= 0:
        return 0
    I = SubsidyReductionInterval
    supply = BLOCK_ONE_SUBSIDY        # sum of the BlockOneLedger amounts
    subsidy = BaseSubsidy
    for _ in range(height // I):
        supply += I * subsidy
        subsidy = subsidy * MulSubsidy // DivSubsidy
    supply += (1 + height % I) * subsidy
    supply -= BaseSubsidy * 2         # blocks 0 and 1 were counted at full subsidy
    return supply
```

dcrd: internal/blockchain/difficulty.go · calcNextRequiredStakeDifficultyV2, calcNextStakeDiffV2, sumPurchasedTickets, estimateSupply

The numerator is formed first and divided by `prev_all` and then by `target_all`, each time with flooring; since all operands are non-negative this equals a single floored division by their product. The maximum is applied before the minimum, so when the supply-based maximum is below `MinimumStakeDiff` the minimum prevails. (DCP-0001 writes the bounds in the opposite nesting; the difference is unobservable on dcrd's networks, where the maximum exceeds the minimum at every height at which a retarget can occur, and dcrd's order is normative.) `estimate_supply` is evaluated at the height of the block being validated. It is a deliberately simplified model of issuance that assumes every block paid its full subsidy; it does not account for missing votes, disapproved blocks, or changes in the subsidy split, which do not alter the total block subsidy.

IMPLEMENTATION PITFALL
The product `cur * cur_all * cur_all` can exceed 64 bits on realistic inputs (for example $2 \times 10^{10} \times 72845^2 \approx 10^{20}$) and **MUST** be computed with wider arithmetic. The quotient is then converted to `int64` modulo 2^{64}, as with version 1; it fits for every reachable input.

TEST VECTOR Supply estimate, mainnet parameters

Height	`estimate_supply`	Maximum price (`// 8192`)
0 (and negative)	0	0
1	168000000000000	20507812500
2	168003119582664	20508193308
4096	180774691009080	22067223023
6143	187160476722288	22846737881
6144	187163565417994	22847114919
149248	583636569410475	71244698414
794368	1569076628758532	191537674409

TEST VECTOR Stake difficulty version 2, mainnet parameters
A chain from dcrd's tests: no purchases in blocks 1 to 256, then 20 ticket purchases in every block, with no votes before block 4096 and 5 per block from 4096. Each row is a retarget block; "prev" and "cur" are pool sizes and immature counts as defined above.

Height	Parent `SBits`	Prev pool	Prev imm.	Cur pool	Cur imm.	Result
2304	200000000	32920	5120	35800	5120	208418769
2448	208418769	35800	5120	38680	5120	231326567
2592	231326567	38680	5120	41560	5120	272451490
3456	1354989669	55960	5120	58840	5120	2148473276
4032	10947547379	67480	5120	70360	5120	20338554623
4176	20338554623	70360	5120	72845	5120	22097687698

For example, at height 2304: prev_all $= 32920 + 5120 = 38040$ and cur_all $= 35800 + 5120 = 40920$; $200000000 \cdot 40920^2 = 334889280000000000$; divided by 38040 this gives 8803608832807, and divided by 42240 it gives 208418769, which is within the bounds. The last row (height 4176) is the only one that is clamped to the maximum: the unclamped value 38776057578 exceeds the estimated supply divided by the target pool size, 22097687698.

NOTE
dcrd also provides an estimate of the next stake difficulty for wallets, which simulates the remaining blocks of the interval under assumed purchase counts. It is not part of consensus.

9.10 Summary of Header Rules

For every block other than the genesis block:

1. `Bits` decodes to a target in $[1, \text{PowLimit}]$ (Rule 9.1).
2. The proof-of-work hash selected by `blake3pow` does not exceed the target (Rule 9.2).
3. `Bits` equals the required bits computed from the parent chain and the block's timestamp by the EMA algorithm or ASERT (Rule 9.3).
4. `SBits` equals the required stake difficulty computed from the parent chain by version 1 or version 2 (Rule 9.6).
5. Every ticket purchase pays at least `SBits` and at least `MinimumStakeDiff` (Rule 8.6).
6. On testnet3 only: Rule 9.4 and Rule 9.5.

CHAPTER 10

Blocks and Chain Selection

A Decred block is accepted in stages. Its header is checked first, on its own and against the headers before it; then its transaction data is checked against the header that commits to it; and finally the block is *connected*: every transaction is checked against the set of unspent outputs that exists at the end of the parent block, and that set is updated. Among all the branches of blocks a node knows about, the one with the most cumulative proof of work whose blocks all pass every stage is the *main chain*.

This chapter specifies those stages and the selection of the main chain. Two features distinguish Decred from its Bitcoin ancestry and run through the whole chapter. First, a block carries two transaction trees, the *regular tree* and the *stake tree*, and they are processed separately. Second, the votes in a block decide whether the regular tree of the *previous* block takes effect at all (Section 10.5). That second rule makes the order in which a node updates its unspent-output set when connecting and disconnecting blocks part of consensus, so it is given in full in Algorithm 10.5 and Algorithm 10.7.

Many block rules belong to other chapters and are only referenced here. The ticket, vote and revocation rules, and the meaning of the stake-related header fields, are in Chapter 8; proof of work, the difficulty bits and the ticket price in Chapter 9; the coinbase and the subsidy in Chapter 11; the treasury transactions in Chapter 12; header and stake versions and the agenda mechanism in Chapter 13; and per-transaction validation in Chapter 7. The serialization of blocks and headers is in Chapter 4.

10.1 Overview

10.1.1 Terminology

Block A header followed by the regular tree and the stake tree, each an ordered list of transactions (Chapter 4). The **block hash** is the BLAKE-256 hash of the serialized header.

Parent The block whose hash is in the `PrevBlock` field. Every block but the genesis block has exactly one parent; a block's **height** is its parent's height plus one.

Block index The set of all headers a node has accepted, which forms a tree rooted at the genesis block. A **branch** is a path from the genesis block to some block; a **chain tip** is a block with no known children.

Main chain The branch currently selected by the rules of Section 10.7. Its last block is the **best block** or **tip**.

Side chain Any branch that is not the main chain.

UTXO set The set of unspent transaction outputs as of the end of a given block of a given branch (Chapter 7). The UTXO set *as of block B* includes the effects of every block from the genesis block to B inclusive, subject to the disapproval rule of Section 10.5.

Agenda active for a block Rules that were changed by a stakeholder vote apply from a certain block on. Whether agenda A is active for block B is a function of B's ancestors only, computed from B's parent as defined in Chapter 13. No agenda is ever active for the genesis block. In this chapter "active" always means active *for the block being validated* unless the text says otherwise.

Stake validation height The parameter `StakeValidationHeight` (4096 on the main network). Blocks at or above this height must contain votes (Chapter 8).

10.1.2 Validation stages

dcrd divides block validation into the stages in Table 10.1, distinguished by how much context each one needs. The division matters to an implementation for three reasons: headers are accepted, and their proof of work verified, before any transaction data is downloaded; a failure can only be attributed to a block hash once the header is known to commit to the data that was checked; and the expensive checks need the full data of every ancestor, which a node that downloads blocks out of order does not always have.

Table 10.1 Block validation stages

Stage	Context needed and purpose	Section
Header sanity	None. Proof of work, timestamp not too far in the future, stake header fields consistent with the claimed height.	Section 10.2.1
Header positional	The headers of all ancestors. Timestamp after the median time past, height, difficulty (provisionally), checkpoints, block version.	Section 10.2.3
Data preconditions	Ancestor headers. Block size within the wire limit and the header commits to the transaction data.	Section 10.3.1
Block sanity	None. Transaction count, size field, per-transaction sanity, duplicates.	Section 10.3.2
Block positional	Ancestor headers. Transaction expiry.	Section 10.3.3
Header contextual	The full data of all ancestors (needed to tally votes). Exact proof-of-work algorithm and difficulty, ticket price, stake version, pool size, final state.	Section 10.2.5
Block contextual	Full ancestor data. Coinbase and treasurybase placement, tree composition, vote counts and approval, signature operations, consensus block size, lock times, vote and revocation eligibility.	Section 10.3.4
Connection	The UTXO set as of the parent. Inputs, fees, subsidies, scripts, relative lock times, treasury spends, header commitment.	Section 10.6

A block is **valid** if and only if it passes every stage and its parent is valid; the genesis block is valid by definition. The order of the stages, and of the checks within them, determines only which error a node reports, except where this chapter states that one check depends on another or that state changes between checks.

RULE 10.1 Validity is inherited

A block whose parent is invalid is invalid. A node **MUST NOT** accept into its main chain any block that has an invalid ancestor, regardless of the cumulative work of the branch.

`dcrd: internal/blockchain/blockindex.go · MarkBlockFailedValidation`
`internal/blockchain/process.go · maybeAcceptBlockHeader`

10.1.3 Agendas that affect block validation

Table 10.2 lists the agendas whose state changes a rule in this chapter. The main-network activation heights are given for orientation only: they are one more than the activation anchors recorded in Chapter 13, and an implementation determines activation from the chain, not from this table.

Table 10.2 Agendas referenced in this chapter

Agenda	DCP	Effect on block validation	Main net
`maxblocksize`	none	Selects the second entry of `MaximumBlockSizes` (Algorithm 10.3).	never
`sdiffalgorithm`	DCP-0001	Ticket price algorithm (Rule 10.15).	149248
`lnfeatures`	DCP-0002, DCP-0003	Absolute lock times against the parent's median time past (Rule 10.36); relative lock times (Rule 10.49); new script flags.	189568
`fixlnseqlocks`	DCP-0004	Historical. Corrected the sequence lock calculation; the reference implementation no longer consults it and applies the corrected calculation to every block (Chapter 7).	342784
`headercommitments`	DCP-0005	Combined Merkle root and commitment root (Rule 10.19, Rule 10.38).	431488

Agenda	DCP	Effect on block validation	Main net
`treasury`	DCP-0006	Coinbase version and outputs, treasurybase, treasury adds and spends (Rule 10.28, Rule 10.46, Rule 10.47).	552448
`reverttreasurypolicy`	DCP-0007	Treasury expenditure limit (Chapter 12).	657280
`explicitverupgrades`	DCP-0008	Maximum transaction and script versions (Rule 10.29).	657280
`autorevocations`	DCP-0009	Mandatory revocations (Rule 10.37); revocation scripts not executed (Rule 10.48).	657280
`changesubsidysplit`	DCP-0010	Subsidy split variant (Rule 10.46, Rule 10.47).	657280
`blake3pow`	DCP-0011	Proof-of-work hash and difficulty algorithm (Rule 10.14, Rule 10.15).	794368
`changesubsidysplitr2`	DCP-0012	Subsidy split variant, taking precedence over `changesubsidysplit`.	794368
`maxtreasuryspend`	DCP-0013	Treasury expenditure limit (Chapter 12).	1052416

10.2 Header Rules

The serialized header is 180 bytes (`MaxBlockHeaderPayload`; layout in Chapter 4). Table 10.3 lists where each field is validated.

Table 10.3 Where each header field is validated

Field	Validated by
`Version`	Old versions rejected once superseded by a majority: Chapter 13 (positional stage).
`PrevBlock`	Must identify a known header that is not known to be invalid: Rule 10.6.
`MerkleRoot`	Rule 10.19.
`StakeRoot`	Stake tree Merkle root before the `headercommitments` agenda, header commitment root after it: Rule 10.19, Rule 10.38.
`VoteBits`	Fixed to `0x0001` before stake validation height (Chapter 8); otherwise bit 0 must agree with the votes: Rule 10.33.
`FinalState`, `PoolSize`	Must match the ticket lottery state of the parent: Chapter 8 (contextual stage).
`Voters`, `FreshStake`, `Revocations`	Bounds in the sanity stage and counts in the block stages: Chapter 8.
`Bits`	Proof of work and difficulty: Chapter 9 (sanity, positional and contextual stages).
`SBits`	Ticket price: Chapter 9 (contextual stage); ticket purchases must pay it: Chapter 8.
`Height`	Rule 10.7.
`Size`	Rule 10.21.
`Timestamp`	Rule 10.3, Rule 10.4, Rule 10.8, Rule 10.9.
`Nonce`, `ExtraData`	No constraint other than through the proof of work hash, which covers them.
`StakeVersion`	Chapter 13 (contextual stage).

10.2.1 Header sanity checks

These checks need nothing but the header itself (and the local clock, for Rule 10.4). They are performed before a header is added to the block index.

RULE 10.2 Proof of work, context-free form

The target encoded by `Bits` **MUST** be within the range allowed by the network's proof-of-work limit, and the header **MUST** satisfy that target under at least one of the two proof-of-work hash functions (BLAKE-256 or BLAKE3). Both functions are accepted here because whether the `blake3pow` agenda (DCP-0011) is active cannot be determined without ancestor data; Rule 10.14 later requires the correct one. The range check, the hash functions and the target comparison are specified in Chapter 9.

`dcrd: internal/blockchain/validate.go · checkProofOfWorkSanity`

RULE 10.3 Timestamp precision
The header timestamp **MUST** be a whole number of seconds. Since the serialized field is an unsigned 32-bit count of seconds, every decoded header satisfies this rule; it constrains only implementations that carry timestamps internally at a finer resolution.

dcrd: internal/blockchain/validate.go · checkBlockHeaderSanity

RULE 10.4 Timestamp not too far in the future
Let T_{adj} be the node's network-adjusted time in whole seconds (Section 10.2.2) at the moment the header is checked. A header whose timestamp is strictly greater than $T_{\text{adj}} + \texttt{MaxTimeOffsetSeconds}$, where $\texttt{MaxTimeOffsetSeconds} = 7200$ (two hours), **MUST** be rejected. A timestamp exactly equal to $T_{\text{adj}} + 7200$ is accepted.

dcrd: internal/blockchain/validate.go · checkBlockHeaderSanity, MaxTimeOffsetSeconds

This is the only rule in the protocol that depends on the local clock. It is evaluated once, when the header is first offered; a header rejected under it is not recorded as invalid (Section 10.7.6) and is accepted if offered again once the clock has caught up. Two honest nodes whose clocks differ can therefore disagree for a while about a block near the bound, but the disagreement resolves itself as time passes; it cannot cause a permanent split.

RULE 10.5 Stake header fields in the sanity stage
The header **MUST** satisfy the context-free bounds on `Voters`, `Revocations`, `VoteBits`, `FinalState` and `FreshStake` specified in Chapter 8: before stake validation height no votes or revocations, `VoteBits` equal to `0x0001` and an all-zero `FinalState`; from stake validation height at least a majority of `TicketsPerBlock` votes; never more than `TicketsPerBlock` votes or more than `MaxFreshStakePerBlock` ticket purchases. These checks use the height claimed in the header, which Rule 10.7 later ties to the actual position.

dcrd: internal/blockchain/validate.go · checkBlockHeaderSanity

10.2.2 Network-adjusted time

The network-adjusted time used by Rule 10.4 is the local clock corrected by the median of the clock offsets reported by peers. It is local policy, not consensus: an implementation may use any reasonable clock. For interoperability with dcrd it is specified here as dcrd computes it. The samples are the timestamps in the `version` messages of peers that complete the handshake (Chapter 15).

ALGORITHM 10.1 Network-adjusted time (dcrd, non-consensus)

```
MAX_ENTRIES = 200            # maxMedianTimeEntries
MAX_OFFSET = 70 * 60         # maxAllowedOffsetSecs: 4200 seconds

known_ids = set()
offsets = []                 # int64 seconds, in arrival order
offset_secs = 0

def add_time_sample(source_id, remote_unix_seconds):
    global offset_secs
    if source_id in known_ids:          # one sample per source
        return
    known_ids.add(source_id)
    now = unix_now_seconds()            # local clock, truncated to seconds
    offsets.append(remote_unix_seconds - now)
    if len(offsets) > MAX_ENTRIES:
        offsets.pop(0)                  # drop the oldest sample
    n = len(offsets)
    if n < 5 or n % 2 == 0:             # only odd counts of at least 5
        return
```

```
    median = sorted(offsets)[n // 2]
    offset_secs = median if abs(median) < MAX_OFFSET else 0

def adjusted_time():
    return unix_now_seconds() + offset_secs
```

dcrd: internal/blockchain/mediantime.go · AddTimeSample, AdjustedTime

NOTE
The update condition reproduces a quirk inherited from Bitcoin Core: once 200 samples have been collected the count stays even, so the offset never changes again for the life of the process.

10.2.3 Header positional checks

Positional checks need the headers of all ancestors, which a node always has because it accepts a header only if its parent header is already known. They do not need any ancestor's transaction data.

RULE 10.6 Parent known and not known invalid
The header's `PrevBlock` **MUST** identify a header already in the block index; otherwise the header is not accepted (it is an *orphan*, error `ErrMissingParent`). If the parent is known to be invalid, or known to have an invalid ancestor, the header **MUST** be rejected (`ErrInvalidAncestorBlock`).

dcrd: internal/blockchain/process.go · maybeAcceptBlockHeader
internal/blockchain/process.go · checkKnownInvalidBlock

dcrd keeps no pool of orphan headers or orphan blocks. Headers must be supplied in order, which the synchronization protocol of Chapter 15 ensures by requesting headers before blocks.

RULE 10.7 Height
The `Height` field **MUST** equal the height of the parent plus one (`ErrBadBlockHeight`).

dcrd: internal/blockchain/validate.go · checkBlockHeaderPositional

RULE 10.8 Timestamp after the median time past
Let M be the median time past of the parent block, computed by Algorithm 10.2 with the parent as argument. The header timestamp **MUST** be strictly greater than M (`ErrTimeTooOld`). A timestamp equal to M is rejected.

dcrd: internal/blockchain/validate.go · checkBlockHeaderPositional

RULE 10.9 Minimum block spacing at maximum difficulty (testnet3 only)
On the version 3 test network only, let h be the height of the block. If $h \geq 962928$ and the `Bits` field, compared as an unsigned 32-bit integer, is less than or equal to `0x1d03ffff` (the compact encoding of the testnet proof-of-work limit shifted right by 6 bits, that is, a difficulty of 64), the timestamp **MUST** be greater than or equal to the parent's timestamp plus 60 seconds (`ErrTimeTooOld`). This rule does not exist on any other network. The difficulty rules that accompany it are in Chapter 9.

dcrd: internal/blockchain/validate.go · checkBlockHeaderPositional, testNet3MaxDiffActivationHeight
internal/blockchain/chain.go · New

RULE 10.10 Difficulty, positional form
The `Bits` field **MUST** equal a value permitted by the difficulty rules of Chapter 9 for this position. Because the `blake3pow` agenda state cannot always be determined from headers, this check accepts the value required by either difficulty algorithm in the circumstances described there; Rule 10.15 later requires the value of the algorithm that is actually in force.

dcrd: internal/blockchain/validate.go · checkDifficultyPositional

RULE 10.11 Block version
A header whose `Version` has been superseded by a sufficient majority of recent blocks **MUST** be rejected (`ErrBlockVersionTooOld`), with the historical exceptions listed in Chapter 13, where the rule is specified.

dcrd: internal/blockchain/validate.go · isOldBlockVersionByMajority
internal/blockchain/validate.go · isDCP0005Violation

RULE 10.12 Fork rejection checkpoint
On a network whose `AssumeValid` parameter is not the zero hash, once the header identified by `AssumeValid` is in the block index, let A be that header, let $E = \lfloor 1209600\ \text{s} \ / \ \texttt{TargetTimePerBlock} \rfloor$ be the expected number of blocks in two weeks (4032 on the main network), and let C be the ancestor of A at height $\max(0, A.\text{height} - E)$. A header that is not already in the block index and whose height is strictly less than C.height **MUST** be rejected (`ErrForkTooOld`). The rule uses the `AssumeValid` value from the network parameters, never a value supplied by the operator, and is not in effect until A is known.

dcrd: internal/blockchain/validate.go · checkBlockHeaderPositional
internal/blockchain/process.go · maybeSetForkRejectionCheckpoint
internal/blockchain/chain.go · New

Every header of the branch leading to A is necessarily in the index before A itself, so the rule rejects only headers of competing branches that fork more than about two weeks of blocks before the assumed-valid block. The outcome depends on which headers a node already has: the rule is applied only to new headers below C.height, so the headers of a competing branch that a node accepted before A became known stay in the index, and that branch can still be extended by headers at heights of C.height and above. For a node that learns the main-chain headers first, as a node synchronizing from the genesis block does, a branch that forks before C can never become the main chain, however much work it has. The rule is thus a checkpoint derived from the network parameters rather than a property of the blocks themselves; an implementation that omits it agrees with dcrd on every chain except such a deep fork. dcrd's operator option `--allowoldforks` disables the rule as a recovery mechanism; a node with the rule disabled can follow a chain that other nodes refuse. On the simulation and regression networks `AssumeValid` is zero and the rule never applies.

RULE 10.13 Maximum-difficulty checkpoint (testnet3 only)
On the version 3 test network, a header at height 962928 **MUST** have the block hash (display form) `0000004fd1b267fd39111d456ff557137824538e6f6776168600e56002e23b93` (`ErrBadMaxDiffCheckpoint`).

dcrd: internal/blockchain/validate.go · checkBlockHeaderPositional, block962928Hash

NOTE
Earlier versions of dcrd carried a list of manual checkpoints in the network parameters. The list is no longer consulted; the two rules above are the only checkpoints that remain.

10.2.4 Median time past

The **median time past** (MTP) of a block B is the median of the timestamps of B and up to ten of its ancestors. It is a time that, unlike the timestamp of a single block, no individual miner can move far, and it is used both by Rule 10.8 and, once the `lnfeatures` agenda is active, as the reference time for transaction lock times (Rule 10.36 and Chapter 7).

ALGORITHM 10.2 Median time past

```
MEDIAN_TIME_BLOCKS = 11
```

```python
def median_time_past(node):
    # node: the block whose MTP is wanted, e.g. the parent of a new block.
    timestamps = []
    n = node
    while len(timestamps) < MEDIAN_TIME_BLOCKS and n is not None:
        timestamps.append(n.timestamp)   # node itself first, then ancestors
        n = n.parent
    timestamps.sort()                    # ascending
    return timestamps[len(timestamps) // 2]
```

dcrd: internal/blockchain/blockindex.go · CalcPastMedianTime
internal/blockchain/validate.go · medianTimeBlocks

Fewer than eleven timestamps are available only when the argument is at height 9 or below, that is, when validating blocks 1 to 10. With an even number n of timestamps the algorithm returns the element at index $n/2$ of the sorted list, the *upper* of the two middle values, and does not average them. The genesis block's own timestamp is included when it is reached. Timestamps are not required to increase from block to block; they are sorted, so the order of the ancestors does not matter.

IMPLEMENTATION PITFALL
The MTP that bounds a new block's timestamp is that of the *parent*, covering the parent and its ten predecessors, not of the new block. A correct implementation computes `median_time_past(parent)`.

TEST VECTOR Median time past
Timestamps are listed from the oldest block to the block passed as argument (the last entry). Produced with dcrd's `CalcPastMedianTime`.

```
[1454954400]                                          -> 1454954400
[1000, 900]                                           -> 1000
[100, 300, 200, 400]                                  -> 300
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]        -> 60
[5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]     -> 60  (5 is ignored)
[110, 100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 1000000]
                                                      -> 60  (110 is ignored)
[1000, 1000, 1000, 999, 1, 1, 1]                      -> 999
```

10.2.5 Header contextual checks

The remaining header checks depend on which agendas are active and on the ticket lottery, both of which are derived from the votes and tickets in ancestor blocks. They therefore need the full data of every ancestor and run only when the block is otherwise ready for full validation (Section 10.7.4).

RULE 10.14 Proof of work, contextual form
The header **MUST** satisfy its target under the proof-of-work hash function in force for the block: BLAKE-256 if the `blake3pow` agenda (DCP-0011) is not active, BLAKE3 if it is (Chapter 9).

dcrd: internal/blockchain/validate.go · checkProofOfWorkContext

RULE 10.15 Difficulty and ticket price, exact
`Bits` **MUST** equal the required difficulty and `SBits` **MUST** equal the required ticket price, both computed from the parent as specified in Chapter 9 (`ErrUnexpectedDifficulty`).

dcrd: internal/blockchain/validate.go · checkBlockHeaderContext

RULE 10.16 Stake version
Under the conditions given in Chapter 13, `StakeVersion` **MUST** equal the expected stake version (`ErrBadStakeVersion`).
dcrd: internal/blockchain/validate.go · checkBlockHeaderContext

RULE 10.17 Ticket pool commitments
`PoolSize` **MUST** equal the number of live tickets and `FinalState` **MUST** equal the final state of the ticket lottery, both as of the parent block (`ErrPoolSize` , `ErrInvalidFinalState`); see Chapter 8.
dcrd: internal/blockchain/validate.go · checkBlockHeaderContext

10.3 Block Body Rules

10.3.1 Data commitment preconditions

When the transaction data of a block arrives, the node first establishes that the data is the data the header commits to. Until that is established, a failure of any later check could be the fault of whoever relayed a corrupted copy rather than of the block itself, so it must not be held against the block hash (Section 10.7.6).

RULE 10.18 Wire size limit
The serialized size of the block **MUST NOT** exceed `MaxBlockPayload` $= 1310720$ bytes (`ErrBlockTooBig`). This is a context-free bound that applies on every network, independent of the consensus limit of Rule 10.35, which is never larger.
dcrd: internal/blockchain/validate.go · checkBlockSizeSanity | wire/msgblock.go · MaxBlockPayload

RULE 10.19 Merkle roots
Let R be the Merkle root of the regular tree and S the Merkle root of the stake tree, each computed with the Merkle tree construction of Chapter 3 over the *full* transaction hashes of Chapter 4 (which cover both prefix and witness), with the root of an empty tree defined as 32 zero bytes.

- If the `headercommitments` agenda (DCP-0005) is **not** active for the block, the header **MUST** have `MerkleRoot` $= R$ and `StakeRoot` $= S$.
- If it **is** active, the header **MUST** have `MerkleRoot` $=$ BLAKE-256$(R \parallel S)$, the root of the two-leaf Merkle tree whose leaves are R and S. `StakeRoot` is then the header commitment root of Rule 10.38.

A mismatch is `ErrBadMerkleRoot` .
dcrd: internal/blockchain/validate.go · checkMerkleRoots
internal/blockchain/validate.go · checkMerkleRootsContext
blockchain/standalone/merkle.go · CalcTxTreeMerkleRoot, CalcCombinedTxTreeMerkleRoot

Whether `headercommitments` is active is in general a contextual question, yet the Merkle check must come before all other data checks. dcrd resolves the circularity as follows, and an implementation that does not process blocks out of order may simply evaluate the agenda directly.

- When the agenda state is known from the headers alone (on the main and test networks its activation is a recorded historical fact; on other networks it may be forced by the parameters), the exact form is checked immediately, and a pass definitively proves that the header commits to the data.
- Otherwise the data is provisionally accepted if it matches *either* form, and the exact form is enforced in the contextual stage. Until then the commitment is not considered proven.

HISTORY
On the main network the `headercommitments` agenda became active at block 431488 (its parent, block 431487, is the recorded anchor; see Chapter 13). Every earlier block uses separate roots. On the test network it became active at block 323328.

TEST VECTOR Combined Merkle root (DCP-0005)
Roots in display form. Produced with dcrd's `CalcMerkleRoot` over the two leaves; they agree with the vectors published in DCP-0005.

```
R = 0000000000000000000000000000000000000000000000000000000000000000
S = 0000000000000000000000000000000000000000000000000000000000000000
-> 988c02a849815a2c70d97fd613a333d766bcb250cd263663c58d4f954240996d

R = c867d085c96604812854399bf6df63d35d857484fedfd147759ed94c3cdeca35
S = 0000000000000000000000000000000000000000000000000000000000000000
-> 1ac7b08e89cf9a09aa37cbc6da5ef565387d5ab7e8a405a6bf75a31efb1a6987

R = a7bb523ac8ee4cfb39f8ccc9a02281825f7e5d88d6874d5e08724f6bb0ac5083
S = 826c95263cc3c7de75b0503796c96a0072f8e4da5adef8b95eee27654008a77b
-> f6b7bd7ac6f1d61c6e48ae1e53302ccc84da2f3b7802a09244c2657a203aa9af
```

10.3.2 Block sanity checks

RULE 10.20 At least one regular transaction
The regular tree **MUST** contain at least one transaction (`ErrNoTransactions`). (Rule 10.26 further requires the first to be the coinbase.) The stake tree may be empty except where Chapter 8 or Chapter 12 requires transactions in it.

dcrd: internal/blockchain/validate.go · checkBlockDataSanity

RULE 10.21 Header size field
The header's `Size` field **MUST** equal the serialized size of the whole block in bytes, header included, as defined in Chapter 4 (`ErrWrongBlockSize`).

dcrd: internal/blockchain/validate.go · checkBlockDataSanity

RULE 10.22 Transaction sanity
Every transaction in both trees **MUST** pass the context-free transaction sanity checks of Chapter 7, with the maximum transaction size `MaxTxSize` (393216 bytes on the main network).

dcrd: internal/blockchain/validate.go · checkBlockDataSanity
blockchain/standalone/tx.go · CheckTransactionSanity

RULE 10.23 Ticket purchases
Every ticket purchase in the stake tree **MUST** pay at least the ticket price `SBits` claimed by the header and at least `MinimumStakeDiff`, and the header's `FreshStake` **MUST** equal the number of ticket purchases in the stake tree (`ErrNotEnoughStake` , `ErrStakeBelowMinimum` , `ErrFreshStakeMismatch`); see Chapter 8.

dcrd: internal/blockchain/validate.go · checkProofOfStake
internal/blockchain/validate.go · checkBlockDataSanity

RULE 10.24 No duplicate transactions
Two transactions in the block, counting both trees together, **MUST NOT** have the same transaction hash (`ErrDuplicateTx`). The comparison uses the transaction hash that excludes witness data (the *prefix hash* of Chapter 4), not the full hash used for the Merkle roots.

dcrd: internal/blockchain/validate.go · checkBlockDataSanity

10.3.3 Block positional checks

RULE 10.25 No expired transactions
Let h be the height of the block. A transaction in either tree **MUST NOT** be expired at height h as defined in Chapter 7: a transaction with nonzero `Expiry` x is expired at h when $h \geq x$ (`ErrExpiredTx`).

dcrd: internal/blockchain/validate.go · checkBlockDataPositional
internal/blockchain/validate.go · IsExpired

10.3.4 Block contextual checks

The rules in this section need the agenda states for the block and, for some, the ticket lottery state of the parent.

RULE 10.26 Coinbase is the first regular transaction
The first transaction of the regular tree **MUST** be a coinbase as defined in Chapter 7 (with the version requirement that applies when the `treasury` agenda is active) (`ErrFirstTxNotCoinbase`), and any other transaction of the regular tree **MUST NOT** be a coinbase (`ErrMultipleCoinbases`).

dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 10.27 Coinbase height commitment
For blocks at height 2 and above, the coinbase **MUST** encode the block height in the output and format specified in Chapter 11 (`ErrFirstTxNotCoinbase` , `ErrCoinbaseHeight`). Blocks 0 and 1 are exempt.

dcrd: internal/blockchain/validate.go · checkCoinbaseUniqueHeight

RULE 10.28 Treasurybase placement
If the `treasury` agenda (DCP-0006) is active, the stake tree **MUST** be non-empty and its first transaction **MUST** be a treasurybase (`ErrFirstTxNotTreasurybase`), any other stake transaction **MUST NOT** be a treasurybase (`ErrMultipleTreasurybases`), and for blocks at height 2 and above the treasurybase **MUST** commit to the block height as specified in Chapter 12. The agenda is never active for the blocks at heights 0 and 1 (Chapter 13), so the placement requirement never applies to them.

dcrd: internal/blockchain/validate.go · checkBlockContext
internal/blockchain/validate.go · checkTreasurybaseUniqueHeight

RULE 10.29 Transaction context checks
Every transaction in both trees **MUST** pass the agenda-dependent transaction checks of Chapter 7 (maximum transaction and script versions, null-input rules for each transaction type, stake opcodes in regular outputs), evaluated with the agendas active for the block.

dcrd: internal/blockchain/validate.go · checkBlockContext
internal/blockchain/validate.go · checkTransactionContext

RULE 10.30 Tree composition
With transaction types determined as in Chapter 7:

- every transaction in the regular tree **MUST** be of the regular type (`ErrStakeTxInRegularTree`);
- a transaction in the stake tree **MUST NOT** be of the regular type (`ErrRegTxInStakeTree`);
- every transaction in the stake tree **MUST** be a ticket purchase, a vote, a revocation, or, when the `treasury` agenda is active, a treasury add, treasury spend or treasurybase (`ErrNonstandardStakeTx`).

dcrd: internal/blockchain/validate.go · checkBlockContext

Transaction types are determined independently of every agenda (Chapter 7): a version 3 transaction of treasury-add, treasury-spend or treasurybase form is a stake transaction even when the `treasury` agenda is not active. Before activation such a transaction is therefore rejected from the regular tree by the first item and from the stake tree by the third (`ErrNonstandardStakeTx`). Once the agenda is active the third item cannot be violated by any transaction that passes the second, and exists to protect against future transaction types.

RULE 10.31 Votes and revocations in the block
The header's `Voters` and `Revocations` fields **MUST** equal the numbers of votes and revocations in the stake tree; from stake validation height every vote **MUST** vote on the parent block (hash and height); before stake validation height the stake tree **MUST** contain only ticket purchases and, when the `treasury` agenda is active, treasury adds and the treasurybase; and the block **MUST NOT** contain more than 255 revocations. These rules (`ErrVotesMismatch` , `ErrVotesOnWrongBlock` , `ErrInvalidEarlyStakeTx` , `ErrTooManyRevocations` , `ErrRevocationsMismatch`) are specified in Chapter 8.
dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 10.32 Treasury transaction limits
When the `treasury` agenda is active the block **MUST NOT** contain more than $\texttt{MaxTAddsPerBlock} = 20$ treasury adds, and treasury spends **MUST** satisfy the voting-interval placement and expiry rules of Chapter 12 (`ErrTooManyTAdds` , `ErrNotTVI` , `ErrInvalidTVoteWindow`).
dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 10.33 Approval of the parent
If the block's height is at least stake validation height, let Y be the number of votes in its stake tree whose vote bits (Chapter 8) have bit 0 set and N the number of the remaining votes. Bit 0 of the header's `VoteBits` field **MUST** be set if $Y > N$ and **MUST** be clear otherwise (`ErrIncongruentVotebit`). Below stake validation height `VoteBits` is fixed at `0x0001` by Rule 10.5.
dcrd: internal/blockchain/validate.go · checkBlockContext, headerApprovesParent

The meaning of that bit is the subject of Section 10.5.

RULE 10.34 Signature operations, fast count
The sum over all transactions in the regular tree followed by the stake tree of the *fast* signature operation count of Section 10.3.5 **MUST NOT** exceed $\texttt{MaxSigOpsPerBlock} = 5000$ (`ErrTooManySigOps`).
dcrd: internal/blockchain/validate.go · checkBlockContext
internal/blockchain/validate.go · countSigOps

RULE 10.35 Consensus block size
The block's serialized size (equal to the header `Size` field by Rule 10.21) **MUST NOT** exceed the maximum block size of Section 10.3.6 for the block (`ErrBlockTooBig`).
dcrd: internal/blockchain/validate.go · checkBlockContext
internal/blockchain/chain.go · maxBlockSize

RULE 10.36 All transactions final
Let h be the block height and let t be the header timestamp if the `lnfeatures` agenda is not active, or the median time past of the *parent* (Algorithm 10.2) if it is. Every transaction in both trees **MUST** be final at height h and time t as defined in Chapter 7 (`ErrUnfinalizedTx`).
dcrd: internal/blockchain/validate.go · checkBlockContext
internal/blockchain/validate.go · IsFinalizedTransaction

HISTORY
The switch from the block's own timestamp to the parent's median time past for absolute lock times is part of the `lnfeatures` agenda (Chapter 13), which became active on the main network at block 189568. Relative lock times, which the same agenda introduced, are checked during connection (Rule 10.49).

RULE 10.37 Vote and revocation eligibility
From stake validation height, every vote **MUST** redeem a ticket selected by the lottery of the parent, every revocation **MUST** redeem a ticket eligible for revocation, and when the `autorevocations` agenda

(DCP-0009) is active the block **MUST** revoke every ticket that becomes missed or expired in it. These rules are specified in Chapter 8.

dcrd: internal/blockchain/validate.go · checkBlockContext
internal/blockchain/validate.go · checkTicketRedeemers

10.3.5 Signature operation limits

A **signature operation** is an execution of a signature-checking opcode, counted statically from scripts as specified in Chapter 6. Two counting methods exist:

Fast count For one transaction, the sum of the static counts of the signature scripts of its inputs and of the public key scripts of its outputs. The signature script of a coinbase's input and of a vote's first input (the stakebase) are not counted, and a treasurybase counts zero in total when the `treasury` agenda is active.

Precise P2SH count For each input of a transaction that spends a pay-to-script-hash output, the precise count of the redeem script pushed by the signature script (Chapter 6). Coinbases, the stakebase input of votes, and, when the `treasury` agenda is active, treasurybases and treasury spends contribute nothing.

The limit `MaxSigOpsPerBlock` $= 1000000/200 = 5000$ is the same on every network. It was derived from a one-megabyte block size and was not changed when the block size was. It is enforced three times, with different scopes:

Table 10.4 The three signature operation limits

Stage	Quantity that must not exceed 5000	Rule
Contextual	Fast count, summed over the regular tree and then the stake tree.	Rule 10.34
Connection, stake tree	Fast count plus precise P2SH count, summed over the stake tree only.	Rule 10.40
Connection, regular tree	Fast count plus precise P2SH count, summed over the regular tree only.	Rule 10.40

All three accumulations use unsigned 32-bit arithmetic, and an addition that overflows is itself a violation (`ErrTooManySigOps`); in practice the limit is reached long before an overflow is possible. The limit is checked after each transaction is added, so the error is raised at the first transaction that crosses it, but the outcome is the same as comparing the final sum.

10.3.6 Block size limits

Three size limits apply to a block:

1. the wire limit `MaxBlockPayload` $= 1310720$ bytes (Rule 10.18), applied before anything else;
2. the requirement that the header `Size` field equal the actual size (Rule 10.21);
3. the consensus maximum (Rule 10.35), selected per block by Algorithm 10.3 from the network parameter `MaximumBlockSizes`.

ALGORITHM 10.3 **Maximum block size for a block**

```
def max_block_size(block):
    sizes = params.MaximumBlockSizes          # list of one or two sizes
    if agenda_active("maxblocksize", block) and len(sizes) > 1:
        return sizes[1]
    return sizes[0]
```

dcrd: internal/blockchain/chain.go · maxBlockSize
internal/blockchain/agendas.go · isMaxBlockSizeAgendaActive

Table 10.5 Maximum block sizes by network

Network	`MaximumBlockSizes`	Effective maximum
Main	393216	393216 bytes for every block. No `maxblocksize` deployment exists, so the agenda is never active, and there is only one size.

Network	MaximumBlockSizes	Effective maximum
Test (v3)	1310720	1310720 bytes for every block. The agenda is considered active by default (Chapter 13) but there is only one size.
Simulation	1000000, 1310720	The deployment forces the choice “yes”, so 1310720 from block 1 on.
Regression	1000000, 1310720	1000000 until the `maxblocksize` agenda is voted in, 1310720 after.

HISTORY
The `maxblocksize` vote was held only on the version 2 test network, which has been retired. The main network’s limit of 393216 bytes (384 KiB) has not changed since launch.

10.4 Header Commitments

DCP-0005 repurposed the header’s `StakeRoot` field. Once the `headercommitments` agenda is active, the stake tree’s Merkle root is folded into `MerkleRoot` (Rule 10.19) and `StakeRoot` instead holds a **commitment root**: the root of a Merkle tree whose leaves are hashes of data about the block that light clients need to verify. The scheme is versioned so that leaves can be added by later agendas; version 1 is the only version defined.

Table 10.6 Version 1 header commitment

Leaf index	Name	Value
0	Filter hash	The BLAKE-256 hash of the serialized version 2 block filter of the block (Chapter 16), or 32 zero bytes if the serialized filter is empty.

The commitment root is the Merkle root (Chapter 3) of the list of leaves. A Merkle tree with a single leaf has that leaf as its root, so the version 1 commitment root is simply the filter hash.

RULE 10.38 Header commitment root
If the `headercommitments` agenda is active for the block, the header’s `StakeRoot` **MUST** equal the version 1 commitment root, that is, the filter hash of Table 10.6 for the block (`ErrBadCommitmentRoot`). The filter is built from the block and the UTXO set as it stands immediately after the block’s own transactions have been applied (step 10 of Algorithm 10.8), so that it can include the scripts of outputs spent by the block, among them outputs created earlier in the same block and, in the historical blocks of Rule 10.45, the regular-tree output spent by a stake transaction of the same block. The exact selection of scripts is specified in Chapter 16. If a script that the filter needs is unavailable, the block is invalid (`ErrMissingTxOut`).

dcrd: internal/blockchain/validate.go · checkConnectBlock
internal/blockchain/headercmt.go · CalcCommitmentRootV1

Inclusion proofs for individual commitments, which let a light client check a filter against a header, are specified in Chapter 16. Note that the commitment root can only be checked by a node that has the UTXO set, so it is checked during connection and not in the earlier stages. dcrd builds the filter for every block, before it tests the agenda; once both trees have connected the construction cannot fail, so this does not add a rule.

10.5 Stakeholder Approval of the Previous Regular Tree

10.5.1 Mechanism

Each vote in block N is cast by a ticket selected by the lottery, and bit 0 of the vote’s vote bits says whether that ticket holder approves the *regular tree of block* $N-1$, the block being voted on. The miner of block N summarizes the votes in bit 0 of the header’s `VoteBits`, and Rule 10.33 forces the summary to be the honest majority: approval requires strictly more approving than disapproving votes. With five tickets per block and at least three votes required, a block whose votes split two to two, or which receives three votes of which two disapprove, disapproves its parent’s regular tree.

If block N disapproves, the regular tree of block $N-1$ is treated, from block N onward on that branch, as if it had never been mined: none of its transactions take effect. The stake tree of block $N-1$ is never affected; the votes, tickets, revocations and treasury transactions in it stand. Block $N-1$ itself remains part of the chain, and its header, proof of work and stake tree remain valid. Disapproval is not invalidity: it is a normal, valid outcome, and both block $N-1$ and block N are valid blocks.

RATIONALE
The Decred design documentation [37] describes disapproval as a "soft" penalty on the miner of the previous block: the miner loses the block's subsidy and fees, but the proof of work that secures the chain is not discarded, and a miner who wants to keep the previous block's reward needs a majority of approving votes in the next block, which the miner does not control. Because the regular tree of a block may be disapproved later, stake transactions may never spend outputs of the regular tree of their own block (Rule 10.42), and the stake tree, which carries the votes that let the chain advance, is never disapproved.

RULE 10.39 Effect of disapproval
Let block N, at height 2 or above, have bit 0 of its header `VoteBits` clear. Then before any transaction of block N is checked or applied, every effect of the regular tree of block $N-1$ on the UTXO set **MUST** be undone: every output it created is removed, and every output it spent is restored to the unspent state with exactly the attributes it had before (amount, script, script version, height, transaction index, coinbase and expiry flags, transaction type). The transactions of block N, in both trees, are then checked and applied against the resulting set. The undo is performed by Algorithm 10.6.

`dcrd: internal/blockchain/validate.go · checkConnectBlock`
`internal/blockchain/utxoviewpoint.go · disconnectDisapprovedBlock, connectBlock`

Since the stake validation height is at least 2 on every network and `VoteBits` is fixed to `0x0001` below it, the height condition only makes explicit that block 1 can never disapprove the genesis block.

10.5.2 Consequences

The rule has the following consequences, all of which follow from the order of operations in Algorithm 10.5.

- **The coinbase of block $N-1$ is lost.** Its outputs are removed; the proof-of-work subsidy and all the transaction fees it collected, including fees of the stake tree (Section 10.6.3), cease to exist. Before the `treasury` agenda the treasury's share of the subsidy was paid by an output of that same coinbase (Chapter 11) and is lost with it. After the agenda, the treasury's share is paid by the treasurybase in the stake tree and is unaffected.
- **The stake tree of block $N-1$ is untouched.** Votes keep their rewards, tickets purchased in it remain tickets, revocations remain, treasury adds and spends and the treasurybase stand. The ticket lottery state, which depends only on stake trees, is the same whether or not $N-1$ is approved.
- **The transactions of the disapproved tree are not invalid.** Their inputs are unspent again, so the same transactions may be included in block N or any later block, where they are validated afresh at their new height (they may since have expired, or a conflicting transaction may have spent the same inputs). The coinbase of $N-1$ cannot be re-included, because a block may contain only its own coinbase.
- **No transaction of block N may spend an output of the regular tree of $N-1$ when N disapproves it,** in either tree: the output no longer exists (`ErrMissingTxOut`). When N approves, both trees of N may spend such outputs, subject to the usual maturity rules.
- **The approval of $N-1$ is decided by block N alone.** Different children of $N-1$ on different branches may decide differently, and a reorganization that replaces N can reverse the decision (Section 10.7.5).
- **The tentative state between blocks.** After block $N-1$ is connected and before block N arrives, the UTXO set of a node includes the regular tree of $N-1$, since a node cannot know how N will vote. Wallets and memory pools must treat these outputs as unconfirmed by stakeholders.

- **Restoring requires the spend data of the parent.** To undo a regular tree a node needs the full details of every output that tree spent, so it **MUST** retain, for at least the tip of each branch it may extend, the outputs spent by that block (dcrd keeps a *spend journal* per block).

10.5.3 UTXO-set operations

The following algorithms define exactly how the UTXO set changes when a block is connected to, or disconnected from, the end of a branch. They operate on a UTXO set *view* that maps an outpoint (transaction hash, output index, tree) to an entry. `spend` removes an entry from the unspent set after recording it in the block's spend record; `add_outputs` is defined in Algorithm 10.4. The tree of an outpoint is the stake tree when the creating transaction's type is a stake type and the regular tree otherwise, which by Rule 10.30 coincides with the tree the transaction was mined in.

ALGORITHM 10.4 **Adding the outputs of a transaction**

```
def add_outputs(view, tx, height, tx_index, treasury_active, only_index=None):
    tree = REGULAR if tx_type(tx) == REGULAR else STAKE
    for i, out in enumerate(tx.outputs):
        if only_index is not None and i != only_index:
            continue                # used by the SBSS exception only
        if tx_type(tx) == TREASURY_ADD and i == 0:
            continue                # the treasury add output is not a utxo
        if is_unspendable(out.value, out.script):
            continue                # e.g. zero value or OP_RETURN (script chapter)
        view[(tx.hash, i, tree)] = Entry(
            amount=out.value, script=out.script, script_version=out.version,
            height=height, tx_index=tx_index, tx_type=tx_type(tx),
            is_coinbase=is_coinbase(tx, treasury_active),
            has_expiry=(tx.expiry != 0))
        # An existing entry for the outpoint is overwritten (no BIP 30 rule).
```

dcrd: internal/blockchain/utxoviewpoint.go · addTxOuts | txscript/script.go · IsUnspendable

ALGORITHM 10.5 **Connecting a block to the UTXO set**

```
def connect_block_utxos(view, block, parent):
    # Precondition: view is the UTXO set as of `parent`, which includes the
    # effects of BOTH trees of `parent`.
    h = block.height
    trsy = agenda_active("treasury", block)            # for this block
    parent_trsy = agenda_active("treasury", parent)    # for the parent!

    # 1. Undo the parent's regular tree if this block disapproves it.
    if h > 1 and block.header.vote_bits & 0x0001 == 0:
        undo_regular_tree(view, parent, spend_record(parent), parent_trsy)

    # 2. Historical exception (main and test networks only, see table).
    sbss = SBSS_VIOLATIONS.get(h)                      # display-form tx hash
    if sbss is not None:
        for idx, tx in enumerate(block.regular_txs):
            if tx.hash == sbss:
                add_outputs(view, tx, h, idx, trsy, only_index=0)
                break

    # 3. The stake tree, in order.  Inputs may only reference outputs that
    #    existed before this block (rule "Stake transactions do not spend
    #    outputs of their own block").
    for idx, tx in enumerate(block.stake_txs):
        if trsy and idx == 0:
            continue                # treasurybase: no inputs, no utxos
        if trsy and is_treasury_spend(tx):
```

```
            add_outputs(view, tx, h, idx, trsy)       # no inputs to spend
            continue
        for i, txin in enumerate(tx.inputs):
            if is_vote(tx) and i == 0:
                continue             # stakebase has no previous output
            spend(view, txin.prev_out)
        add_outputs(view, tx, h, idx, trsy)

    # 4. The regular tree, in order.  A transaction may spend outputs of
    #    transactions earlier in the regular tree of this block.
    for idx, tx in enumerate(block.regular_txs):
        if idx > 0:                  # the coinbase (index 0) has no inputs
            for txin in tx.inputs:
                spend(view, txin.prev_out)
        add_outputs(view, tx, h, idx, trsy)

    # The spend record of `block` lists the entries spent in step 3 then
    # step 4, in transaction order and input order.  Step 1 is not recorded.
```

dcrd: internal/blockchain/utxoviewpoint.go · connectBlock, connectStakeTransaction, connectRegularTransaction
internal/blockchain/validate.go · checkConnectBlock

ALGORITHM 10.6 Undoing a regular tree

```
def undo_regular_tree(view, blk, record, trsy):
    # record: the spend record of `blk`; its regular-tree part is its last
    # count_regular_spends(blk) entries.  trsy: treasury agenda state for blk.
    j = len(record) - 1
    for idx in reversed(range(len(blk.regular_txs))):
        tx = blk.regular_txs[idx]
        for i, out in enumerate(tx.outputs):
            if not is_unspendable(out.value, out.script):
                view.remove((tx.hash, i, REGULAR))
        if idx == 0:
            continue                 # coinbase: nothing to restore
        for txin in reversed(tx.inputs):
            view[txin.prev_out] = record[j].as_entry()   # restore as unspent
            j -= 1
```

dcrd: internal/blockchain/utxoviewpoint.go · disconnectTransactions
internal/blockchain/utxoviewpoint.go · disconnectDisapprovedBlock

ALGORITHM 10.7 Disconnecting a block from the UTXO set

```
def disconnect_block_utxos(view, block, parent):
    # Precondition: view is the UTXO set as of `block`.
    # Postcondition: view is the UTXO set as of `parent`.
    h = block.height
    trsy = agenda_active("treasury", block)
    parent_trsy = agenda_active("treasury", parent)
    record = spend_record(block)

    # 1. Undo the regular tree of `block` (reverse order).
    undo_regular_tree(view, block, record, trsy)

    # 2. Undo the stake tree of `block` (reverse order).  Its spends are the
    #    first count_stake_spends(block) entries of the record.
    j = count_stake_spends(block) - 1
    for idx in reversed(range(len(block.stake_txs))):
        tx = block.stake_txs[idx]
        for i, out in enumerate(tx.outputs):
            if not is_unspendable(out.value, out.script):
```

```
                view.remove((tx.hash, i, STAKE))
        if (trsy and idx == 0) or (trsy and is_treasury_spend(tx)):
            continue                # no inputs
        for i in reversed(range(len(tx.inputs))):
            if is_vote(tx) and i == 0:
                continue
            view[tx.inputs[i].prev_out] = record[j].as_entry()
            j -= 1

    # 3. Historical exception: remove the re-added output again.
    sbss = SBSS_VIOLATIONS.get(h)
    if sbss is not None and any(tx.hash == sbss for tx in block.regular_txs):
        view.remove((sbss, 0, REGULAR))

    # 4. Redo the parent's regular tree if `block` had disapproved it.
    if block.header.vote_bits & 0x0001 == 0:
        for idx, tx in enumerate(parent.regular_txs):
            if idx > 0:
                for txin in tx.inputs:
                    spend(view, txin.prev_out)   # not recorded again
            add_outputs(view, tx, parent.height, idx, parent_trsy)
```

dcrd: internal/blockchain/utxoviewpoint.go · disconnectBlock, removeSBSSViolations
internal/blockchain/chain.go · reorganizeChainInternal

IMPLEMENTATION PITFALL

Every step that involves the parent's transactions uses the agenda states *of the parent*: in particular whether the `treasury` agenda is active for the parent determines how the parent's coinbase is recognized (Chapter 7). Using the state of the child is wrong at the activation boundary. dcrd itself carried this error until shortly before the commit this document describes.

NOTE

The tree in which a spent output lives is taken from the outpoint in the spending input, and the spend record stores the output's attributes so that it can be restored without the creating transaction. Outputs created and spent within one regular tree appear in the view only transiently but are recorded like any other spend.

Table 10.7 Example of disapproval. Block 101 contains, in its regular tree, the coinbase C_{101} and a transaction T that spends output U (created in block 50) and creates output $T:0$; its stake tree contains a ticket purchase P. The table shows the state of each output.

Output	After 101	After 102, disapproving	After 102, approving
U	spent	unspent	spent
$T:0$	unspent	does not exist	unspent
C_{101} outputs	unspent (immature)	do not exist	unspent (immature)
P outputs	unspent	unspent	unspent

A transaction of block 102, in either tree, that spends $T:0$ is valid only if block 102 approves block 101. If it disapproves, T itself may be included again in the regular tree of block 102 or of a later block.

10.6 Connecting a Block

10.6.1 Order of operations

Connection checks a block against the UTXO set as of its parent and, if it passes, produces the UTXO set as of the block. The checks are interleaved with the updates of Algorithm 10.5, and since later checks see the effects of earlier updates, the order matters. Algorithm 10.8 gives it.

ALGORITHM 10.8 **Checking and connecting a block**

```
def check_connect_block(node, block, parent, view):
    # view: UTXO set as of `parent`.  node: index entry for `block`.
    h, voters = node.height, block.header.voters
    trsy = agenda_active("treasury", block)

    # 1. Treasury payment of this block (amount and form).
    if trsy:
        check_treasurybase(block.stake_txs[0], h, voters)     # treasury chapter
    else:
        check_coinbase_pays_treasury(block.regular_txs[0], h, voters)  # subsidy
    # (both are no-ops for h <= 1)

    # 2. Disapproval of the parent (rule "Effect of disapproval").
    if h > 1 and block.header.vote_bits & 0x0001 == 0:
        undo_regular_tree(view, parent, spend_record(parent),
                          agenda_active("treasury", parent))

    # 3. Historical same-block stake spends (step 2 of the UTXO algorithm).
    add_sbss_violation_output(view, block)

    # 4. Stake tree: inputs, fees, vote subsidies; applies the tree.
    variant = subsidy_split_variant(block)                    # subsidy chapter
    stake_fees = check_and_connect_tree(view, node, block, STAKE, 0, variant)

    # 5. Treasury spends: window, uniqueness, votes, expenditure limit.
    if trsy:
        treasury_spend_checks(block)                          # treasury chapter

    # 6. Relative lock times of the stake tree.
    ln = agenda_active("lnfeatures", block)
    if ln:
        mtp = median_time_past(node.parent)
        for tx in block.stake_txs:
            require_sequence_locks(tx, h, mtp, view)          # transactions ch.

    # 7. Scripts of the stake tree.
    validate_scripts(block.stake_txs, view)

    # 8. Regular tree: inputs, fees, coinbase amount; applies the tree.
    check_and_connect_tree(view, node, block, REGULAR, stake_fees, variant)

    # 9. Relative lock times of the regular tree (coinbase excluded).
    if ln:
        for tx in block.regular_txs[1:]:
            require_sequence_locks(tx, h, mtp, view)

    # 10. Header commitment (rule "Header commitment root").
    if agenda_active("headercommitments", block):
        f = build_v2_filter(block, view)                      # filters chapter
        require(block.header.stake_root == filter_hash(f))

    # 11. Scripts of the regular tree.
    validate_scripts(block.regular_txs, view)

    # 12. Block one pays the initial ledger.
    if h == 1:
        check_block_one_ledger(block.regular_txs[0])          # subsidy chapter
```

dcrd: internal/blockchain/validate.go · checkConnectBlock
internal/blockchain/validate.go · checkTransactionsAndConnect

The scripts of a tree are validated against the view after that tree has been applied. Script execution reads only the script and version of each spent output, which a spent entry retains, so this gives the same result as validating before applying. Steps 7 and 11 may equally be run in parallel with, or after, the other steps.

ALGORITHM 10.9 Checking and applying one tree

```
MAX_SIGOPS = 5000

def check_and_connect_tree(view, node, block, tree, carried_fees, variant):
    h, voters = node.height, block.header.voters
    trsy = agenda_active("treasury", block)
    txs = block.stake_txs if tree == STAKE else block.regular_txs
    total_fees: int64 = carried_fees
    sigops: uint32 = 0
    for idx, tx in enumerate(txs):
        is_cb = tree == REGULAR and idx == 0
        is_vote = tree == STAKE and is_vote_tx(tx)
        n = checked_add_u32(fast_sigops(tx, is_cb, is_vote, trsy),
                            p2sh_sigops(tx, is_cb, is_vote, view, trsy))
        sigops = checked_add_u32(sigops, n)      # overflow: "ErrTooManySigOps"
        if sigops > MAX_SIGOPS:
            reject("ErrTooManySigOps")
        fee = check_transaction_inputs(tx, h, view, block)   # transactions ch.
        total_fees = checked_add_i64(total_fees, fee)        # "ErrBadFees"
        connect_tx(view, tx, h, idx, tree)                   # UTXO algorithm
    if tree == REGULAR:
        check_coinbase_amount(txs[0], h, voters, total_fees, trsy, variant)
        # rule "Coinbase amount"
    else:
        check_stake_amounts(txs, h, voters, trsy, variant, view)
        # rule "Vote subsidies and treasurybase amount"
    return total_fees
```

dcrd: internal/blockchain/validate.go · checkTransactionsAndConnect
internal/blockchain/validate.go · checkStakeBaseAmounts, getStakeBaseAmounts

RULE 10.40 Signature operations, per tree

For each tree separately, the sum over its transactions of the fast count plus the precise P2SH count (Section 10.3.5) **MUST NOT** exceed 5000 (`ErrTooManySigOps`). The precise count of an input requires the output it spends to exist in the view; if it does not, the block is invalid (`ErrMissingTxOut`).

dcrd: internal/blockchain/validate.go · checkTransactionsAndConnect
internal/blockchain/validate.go · CountTotalSigOps

RULE 10.41 Transaction inputs

Every transaction **MUST** pass the input checks of Chapter 7 (and, for stake and treasury transactions, those of Chapter 8 and Chapter 12) against the view as it stands immediately before the transaction is applied, at the height of the block, with the fraud-proof witness fields checked and with the agendas active for the block. The view at that point reflects the steps of Algorithm 10.8 that precede it, including every earlier transaction of the same tree.

dcrd: internal/blockchain/validate.go · checkTransactionsAndConnect
internal/blockchain/validate.go · CheckTransactionInputs

10.6.2 Spending within a block

The per-transaction availability rule (an input must reference an existing, unspent output) is in Chapter 7. Combined with the order of Algorithm 10.5 it yields these block-level rules:

RULE 10.42 Stake transactions do not spend outputs of their own block
An input of a transaction in the stake tree **MUST NOT** reference an output created by any transaction of the same block, in either tree (`ErrMissingTxOut`), except for the historical outputs of Table 10.8.

dcrd: internal/blockchain/validate.go · CheckTransactionInputs
internal/blockchain/utxoviewpoint.go · fetchInputUtxos

RULE 10.43 Regular transactions spend only earlier outputs of their own block
An input of the transaction at index i of the regular tree **MAY** reference an output of the transaction at index j of the same regular tree only if $j < i$. A reference to the same or a later transaction of the block refers to an output that does not exist yet (`ErrMissingTxOut`).

dcrd: internal/blockchain/utxoviewpoint.go · addRegularInputUtxos
internal/blockchain/validate.go · CheckTransactionInputs

RULE 10.44 Each output is spent at most once per block
Because both trees are applied to one view in sequence, an output spent by one transaction of the block is no longer available to any later transaction of the block, in either tree. In particular an output spent in the stake tree cannot also be spent in the regular tree (`ErrMissingTxOut`).

dcrd: internal/blockchain/validate.go · CheckTransactionInputs

Table 10.8 Historical same-block stake spends (SBSS). Output 0 of the listed regular-tree transaction, created in the block at the listed height, is spent by a stake transaction of the same block. Hashes in display form.

Network	Height	Transaction hash
Main	1106817	0875ecdc5f12c8aa6ee0d54331fd5b5e786639882a3adb09882de5e3b939613b
Main	1106831	714e71358937cd33480fd2900853d43ea3440ca67b38bacdfeaeb0c37c80ee27
Main	1106837	aeba09a4efa334e3c13e68f27f40bd37cda7643c263f88d69f38a33c2cdacd18
Main	1106854	17228266d9d6dd7084aa2e479319aca6d755c2a9d2dbb3cf28c8151470c7a439
Main	1106860	20d6d3ed1b609f03969474dca1f452c25e035766457dfd58061f9b9270fc4e6d
Main	1107195	5d2e1898fe0c631cb795b1906fb3cf8d9772f1826be8dea64dd3ab85ac0ab2d3
Test (v3)	1980161	451b6eed1a777bc5bbb5c1dbe10a9e444cbeb7d863adb75fc6b7a676b3dcbb58

RULE 10.45 Historical same-block stake spends
When connecting a block at a height listed in Table 10.8 for the network, if the block's regular tree contains the listed transaction T, output 0 of T **MUST** be added to the view before the stake tree is processed, and a stake transaction of the block **MAY** spend it; the same-height restriction of Rule 10.42 does not apply to that output. When the regular tree is then applied, all outputs of T, output 0 included, are added as unspent, overwriting the spent entry. After the block, output 0 of T is therefore unspent, unless a transaction later than T in the same regular tree spends it. Disconnecting the block removes it again (Algorithm 10.7, step 3).

dcrd: internal/blockchain/validate.go · sbssViolations, isSBSSViolation
internal/blockchain/utxoviewpoint.go · addSBSSViolations

IMPLEMENTATION PITFALL
The way dcrd implements this exception leaves two further block shapes that its input checks do not reject. They apply only to a block at a height listed in Table 10.8 that contains T. First, output 0 of T is in the view before *either* tree is processed, so the input checks pass a regular transaction that precedes T in the regular tree and spends it (with input witness height h and index that of T), contrary to Rule 10.43. Second, the exemption from Rule 10.42 is keyed on the hash of T, not on the output index, so the input checks pass a stake transaction that spends any output of T present in the view when the stake tree is processed, which, besides output 0, is every output of T that a transaction later than T in the regular tree spends. Neither shape is accepted, however: while connecting a block to the main chain, after validating it and applying its transactions to the

view and before committing the result, dcrd runs a consistency assertion over the view `internal/blockchain/utxoviewpoint.go · assertBlockUtxoView`, which excuses only a stake transaction spending output 0 of T, and the node aborts on either shape. Their validity is therefore undefined, not accepted. Neither occurs in the seven historical blocks, and an implementation should treat output 0 of T spent by a stake transaction as the only exception.

HISTORY
These seven blocks were accepted by earlier versions of dcrd because of a defect, since corrected, that allowed a stake transaction to spend an output of the regular tree of its own block. The rule was then made explicit, and the existing violations had to be special-cased so that the historical chain still validates. The resulting effect on the UTXO set, including the anomaly that the spent output remains unspent, is the one current dcrd produces, and an implementation must reproduce it to agree with dcrd about later blocks.

10.6.3 Fees and subsidy amounts

The **fee** of a transaction is the sum of its input values less the sum of its output values, as computed by the input checks of Chapter 7; it is never negative, and is zero for a coinbase. For a vote the stakebase input counts at its declared value, and for a treasurybase and a treasury spend the single input counts at its declared value.

The fees of the stake tree are carried into the regular tree's accumulation (step 8 of Algorithm 10.8), so the coinbase may claim the fees of both trees. Both accumulations are signed 64-bit additions in which an overflow is a violation (`ErrBadFees`).

RULE 10.46 Coinbase amount
Let F be the total fees of the stake tree and the regular tree of the block, as an `int64` . If the block's height h is at least stake validation height, replace F by $\lfloor F \cdot \text{Voters} \,/\, \texttt{TicketsPerBlock} \rfloor$, computed as `int64` multiplication followed by division (both operands are non-negative), with `Voters` the header field. Let the expected coinbase total be

- $E = \text{BlockSubsidy}(1)$ if $h = 1$;
- $E = \text{WorkSubsidy}(h, \text{Voters}) + F$ if $h > 1$ and the `treasury` agenda is active;
- $E = \text{WorkSubsidy}(h, \text{Voters}) + \text{TreasurySubsidy}(h, \text{Voters}) + F$ otherwise,

with the subsidy functions, and the variant of the work subsidy selected by the `changesubsidysplit` and `changesubsidysplitr2` agendas, as specified in Chapter 11. Then for $h > 0$ the value declared by the coinbase's input **MUST** equal $E - F$ (`ErrBadCoinbaseAmountIn`), and the sum of the coinbase's output values **MUST NOT** exceed E (`ErrBadCoinbaseValue`).

dcrd: `internal/blockchain/validate.go · checkTransactionsAndConnect`

RATIONALE
The design documentation [37] gives the reason for this linear reduction, which applies to the subsidy (Chapter 11) as well as to the fees: it discourages a miner from manipulating which votes are included, for instance by omitting votes that disapprove the miner's own previous block.

NOTE
Worked example: stake tree fees 30000 and regular tree fees 70001 atoms in a block above stake validation height with four votes give $F = 100001$ and $\lfloor 100001 \cdot 4/5 \rfloor = \lfloor 400004/5 \rfloor = 80000$. The coinbase may then pay at most $\text{WorkSubsidy}(h, 4) + 80000$ atoms, plus $\text{TreasurySubsidy}(h, 4)$ before the `treasury` agenda.

RULE 10.47 Vote subsidies and treasurybase amount
When the stake tree is checked:

1. If $h > 1$ and the `treasury` agenda is active, the stake tree **MUST** be non-empty and the value declared by the treasurybase's input **MUST** equal $\mathrm{TreasurySubsidy}(h, \mathrm{Voters})$ (`ErrNoStakeTx` , `ErrBadTreasurybaseAmountIn`).
2. If the stake tree is empty, the block **MUST** be below stake validation height (`ErrNoStakeTx`).
3. For every vote, the sum of its output values less the amount of the ticket output it spends **MUST NOT** exceed $\mathrm{VoteSubsidy}(h-1)$ (`ErrSSGenSubsidy`).
4. The sum of that quantity over all votes **MUST NOT** exceed $\mathrm{VoteSubsidy}(h-1) \cdot \mathrm{Voters}$ if h is at least stake validation height, or 0 otherwise (`ErrBadStakebaseValue`).

The vote subsidy is that of the block voted on, height $h-1$, in the variant selected by the subsidy split agendas (Chapter 11). The exact form of each vote is checked by Chapter 8.

dcrd: internal/blockchain/validate.go · checkTransactionsAndConnect
internal/blockchain/validate.go · checkStakeBaseAmounts, getStakeBaseAmounts

10.6.4 Script validation

RULE 10.48 Script validation

For every input of every transaction in both trees, the signature script of the input and the public key script of the output it spends **MUST** execute successfully under the script rules of Chapter 6, with the script version of the spent output and the verification flags that Chapter 7 specifies for the block (`ErrScriptValidation` , `ErrScriptMalformed`). The following inputs are exempt:

- any input whose previous outpoint index is `0xffffffff`: the input of a coinbase, the stakebase of a vote, the input of a treasurybase and the input of a treasury spend (whose authorization is checked by Chapter 12);
- every input of a revocation of transaction version 2 or higher when the `autorevocations` agenda is active, whose outputs are fixed by consensus (Chapter 8).

dcrd: internal/blockchain/scriptval.go · checkBlockScripts
internal/blockchain/validate.go · consensusScriptVerifyFlags

An input with index `0xffffffff` that is not one of the listed kinds cannot occur in a valid block: the transaction context checks reject a null outpoint in other transaction types, and no transaction has 2^{32} outputs, so the input check fails for any other such reference.

10.6.5 Lock times

RULE 10.49 Relative lock times

If the `lnfeatures` agenda is active, every transaction of both trees that is subject to sequence locks (Chapter 7 exempts the coinbase, votes and the treasurybase) **MUST** have its sequence locks satisfied at the block's height and at the median time past of the parent, as specified in Chapter 7 (`ErrUnfinalizedTx`).

dcrd: internal/blockchain/validate.go · checkConnectBlock
internal/blockchain/validate.go · SequenceLockActive

10.6.6 The genesis block and block one

The genesis block of each network is fixed by the network parameters (Chapter 14). It is not validated, and in particular its header fields need not satisfy the rules of this chapter: the main network's genesis header, for instance, has `Size` $= 0$ although the block serializes to 300 bytes, and `VoteBits` $= 0$. The transactions of the genesis block are not added to the UTXO set, so its outputs can never be spent. Every agenda is inactive for the genesis block.

Block 1 is special in the following ways:

- its coinbase pays the initial token distribution, the *block one ledger* of Chapter 11, and its expected coinbase total is $\mathrm{BlockSubsidy}(1)$ (Rule 10.46);

- it is exempt from the coinbase height commitment and from the treasury payment checks (step 1 of Algorithm 10.8; Chapter 11, Chapter 12);
- the `treasury` agenda is never active for it, on any network and even where the agenda's result is forced, so it has no treasurybase and no treasury spends (Chapter 13);
- it cannot disapprove the genesis block (Rule 10.39).

Since nothing is spendable at height 1, block 1 can in practice contain only its coinbase.

10.6.7 Rules that do not exist

Two checks that an implementer familiar with Bitcoin might expect are not part of Decred's consensus rules:

- **No check against overwriting unspent outputs.** dcrd contains the equivalent of Bitcoin's BIP 30 check but has it permanently disabled (`checkForDuplicateHashes` is false). The coinbase height commitment makes coinbase hashes unique, and any other transaction whose hash repeated an earlier one would have to spend the same, already spent, inputs. An implementation **MUST NOT** reject a block for creating an output that already exists in the UTXO set; Algorithm 10.4 overwrites it.
- **No limit on the number of transactions** other than the size limits.

10.7 Chain Selection

10.7.1 The block index

A node keeps every header it has accepted in its block index, together with a per-block status. The status values used by dcrd are listed in Table 10.9; they are local state, but an implementation needs equivalent information to apply the rules of this section.

Table 10.9 Block status (dcrd)

Status	Meaning
Data stored	The full block has been received and stored.
Fully linked	The block and all of its ancestors have their data; the block can be validated.
Validated	The block has passed connection (or was accepted under assume-valid, Section 10.7.8). Implies all ancestors are valid.
Validation failed	The block itself violates a consensus rule.
Invalid ancestor	Some ancestor failed validation.

10.7.2 Cumulative work

Each header contributes a **work** value that depends only on its `Bits` field, $\text{work}(b) = \lfloor 2^{256} / (\text{target}(b) + 1) \rfloor$, specified in Chapter 9. The **cumulative work** of a block is the sum of the work of the block and all its ancestors, the genesis block included, computed as an unsigned 256-bit integer. Work does not depend on the actual hash value, on the proof-of-work hash function, or on anything in the stake system.

10.7.3 The best chain

RULE 10.50 **Best chain**

Among the blocks whose own data and whose ancestors' data are all available and none of which is known to be invalid, a node **MUST** make the tip of its main chain a block with the greatest cumulative work that passes full validation together with all its ancestors. If the block with the greatest work fails validation while the node attempts to make it the tip, it and its descendants are marked invalid and the node repeats the selection.

dcrd: `internal/blockchain/blockindex.go · FindBestChainCandidate, betterCandidate`
`internal/blockchain/chain.go · reorganizeChain`

Cumulative work is the only consensus criterion. Ties between blocks of equal cumulative work are broken by local policy, which affects which of two valid branches a node follows until one of them gains more work, but never which blocks are valid. dcrd prefers one candidate over another by the first of these criteria that distinguishes them:

1. greater cumulative work;
2. having the block data available (always true for candidates);
3. the block that became fully linked first, that is, whose data and all of whose ancestors' data were first all present (a counter assigned in memory, so a miner gains no advantage by announcing a header early);
4. the smaller block hash, read as a little-endian 256-bit integer.

Consequently a node does not switch to a competing branch of equal work: the branch it already follows was linked first.

ALGORITHM 10.10 Candidate ordering (dcrd)

```
def better_candidate(a, b):          # True if a is preferred over b
    if a.cumulative_work != b.cumulative_work:
        return a.cumulative_work > b.cumulative_work
    if a.have_data != b.have_data:
        return a.have_data
    if a.linked_order != b.linked_order:
        return a.linked_order < b.linked_order   # 0 for blocks loaded from disk
    return int.from_bytes(a.hash, "little") < int.from_bytes(b.hash, "little")
```

dcrd: internal/blockchain/blockindex.go · betterCandidate, compareHashesAsUint256LE

The stake system enters chain selection only through validity: from stake validation height every block needs at least three votes from tickets chosen by the lottery of its parent, so a branch whose blocks the ticket holders do not vote on cannot be extended, regardless of hash power.

10.7.4 Processing headers and blocks

dcrd processes headers and blocks as described below. The flow is not itself consensus, but it determines when each stage of Table 10.1 runs and when a failure condemns a block hash, and an implementation that departs from it must preserve those properties.

ALGORITHM 10.11 Accepting a header

```
def accept_header(header):
    node = index.lookup(header.hash())
    if node is not None:
        if node.validate_failed: reject("ErrKnownInvalidBlock")
        if node.invalid_ancestor: reject("ErrInvalidAncestorBlock")
        return node                                  # already known
    header_sanity(header)                            # section: header sanity
    parent = index.lookup(header.prev_block)
    if parent is None:
        reject("ErrMissingParent")                   # no orphan headers
    if parent.validate_failed or parent.invalid_ancestor:
        reject("ErrInvalidAncestorBlock")
    header_positional(header, parent)                # section: positional
    node = index.add(header, parent)   # cumulative work = parent's + work(bits)
    maybe_set_fork_rejection_checkpoint()            # rule "Fork rejection checkpoint"
    return node
```

dcrd: internal/blockchain/process.go · ProcessBlockHeader, maybeAcceptBlockHeader

A header that fails any of these checks is simply not added; nothing is recorded about its hash.

ALGORITHM 10.12 Processing a block

```
def process_block(block):
    if index.has_block_data(block.hash()): reject("ErrDuplicateBlock")
    node = accept_header(block.header)      # no-op lookup if already known
    proven = data_preconditions(block, node.parent)    # size, Merkle roots
    try:
        block_sanity(block)                            # section: block sanity
        fast = bulk_import or assume_valid_ancestor(node)  # assume-valid
        if fast: node.validated = True
        block_positional(block, node.parent, fast)     # expiry (skipped if fast)
    except RuleError:
        if proven: mark_failed(node)                   # see invalid marking
        raise
    store(block); node.data_stored = True
    linked = link_descendants(node)   # node and stored descendants whose
                                      # ancestors now all have data
    for n in linked:                  # in order, parents before children
        try:
            block_context(n, fast)    # header + block contextual stages;
                                      # `fast` is that of `block`, even for
                                      # descendants (see note in assume-valid)
        except RuleError as e:
            if e.kind != "ErrBadMerkleRoot": mark_failed(n)
            break                     # the rest are checked only if a later
                                      # reorganization tries to connect them
    reorganize_to(best_candidate())   # chain selection, connects blocks
```

dcrd: internal/blockchain/process.go · ProcessBlock, maybeAcceptBlockData, maybeAcceptBlocks

Blocks whose ancestors lack data are stored and linked later, when the missing data arrives. Full validation (connection) happens only when a block becomes part of the main chain, inside the reorganization of Algorithm 10.13; a block on a side chain that never has the most work is checked through the contextual stage but never connected, and is neither known valid nor known invalid.

The contextual stage of a block on a side chain needs the agenda states, difficulty, ticket price and ticket lottery state as of its side-chain parent, but not the UTXO set. A node must therefore be able to compute the ticket lottery state for any block in its index whose ancestors have data, which dcrd does by undoing and replaying the stake effects of blocks from the main-chain tip to the fork point and along the side chain (Chapter 8).

> **NOTE**
> Block templates offered to miners are checked by the same stages, without the proof-of-work check, against the current tip or the tip's parent (the latter when the tip lacks enough votes to be extended). This is mining policy (Chapter 17).

10.7.5 Reorganization

A **reorganization** moves the tip of the main chain from the current tip to a target block. Let F be the last common ancestor of the two (the **fork point**). The blocks of the current main chain after F are disconnected, newest first, and the blocks from F to the target are connected, oldest first. The common case, extending the main chain by one block, is the special case in which F is the current tip.

ALGORITHM 10.13 Reorganizing to a target

```
def reorganize_to(target):
    while target is not None and tip != target:
        try:
            reorganize_internal(target)
        except RuleError:
```

```
            new_target = best_candidate()     # failed blocks are now excluded
            if new_target == target:
                break                         # unrecoverable, give up
            target = new_target

def reorganize_internal(target):
    fork = last_common_ancestor(tip, target)
    while tip != fork:                        # disconnect, newest first
        disconnect_block_utxos(view, tip.block, tip.parent.block)
        stake_state.disconnect(tip)           # ticket pool back to the parent
        tip = tip.parent
    for n in path(fork, target):              # connect, oldest first
        if n.validated:                       # previously validated block
            connect_block_utxos(view, n.block, n.parent.block)
        else:
            try:
                block_context(n)              # if not already done
                check_connect_block(n, n.block, n.parent.block, view)
            except RuleError as e:
                if not (in_context_stage(e) and e.kind == "ErrBadMerkleRoot"):
                    mark_failed(n)
                raise                         # tip stays at n.parent
            n.validated = True
        stake_state.connect(n)                # ticket pool as of n
        tip = n
```

dcrd: internal/blockchain/chain.go · reorganizeChain, reorganizeChainInternal
internal/blockchain/chain.go · connectBlock, disconnectBlock

The state that a reorganization rolls back and rolls forward is:

- **the UTXO set**, by Algorithm 10.7 and Algorithm 10.5, which also reverse or reapply the disapproval of parents. Disconnecting a block that disapproved its parent reinstates the parent's regular tree; connecting the blocks of the new branch may disapprove a different set of parents. Disconnecting a block needs that block's spend record;
- **the ticket lottery state** (live, missed, expired and revoked tickets, pool size, winners and final state), which is a function of the branch and is updated block by block as specified in Chapter 8;
- **agenda states, difficulty and ticket price**, which are functions of the branch and need no explicit rollback;
- **the treasury balance and the set of treasury spends already mined**, which are functions of the branch (Chapter 12).

A reorganization that fails part way leaves the tip at the last block that connected successfully, and the selection is repeated, which in general moves the tip again (possibly back to the original branch). The resulting tip satisfies Rule 10.50.

10.7.6 Invalid blocks

RULE 10.51 Recording invalidity

When a block violates a consensus rule, the block is marked as having failed validation and every descendant of it known to the node is marked as having an invalid ancestor; such blocks are never again candidates for the main chain (Rule 10.1). A failure **MUST NOT** be recorded against the block hash when it may be due to a corrupted copy of the block data rather than to the block itself, namely:

- a failure of the data preconditions (Section 10.3.1), or of any earlier deserialization of the block message;
- a failure of the block sanity or positional stage while the header's commitment to the data has only been provisionally established (Section 10.3.1);
- a Merkle root mismatch detected in the contextual stage.

A header that fails the header stages is not added to the block index, so nothing is recorded for it either.

dcrd: internal/blockchain/blockindex.go · MarkBlockFailedValidation
internal/blockchain/process.go · ProcessBlock, maybeAcceptBlocks
internal/blockchain/chain.go · reorganizeChainInternal

RATIONALE
The block hash covers only the header. Two messages with the same hash can carry different transaction data, one valid and one not; the `fullblocktests` suite checks that a node rejects a block made too large by a non-canonical length prefix and then accepts the correctly encoded block with the same hash.

10.7.7 Manual invalidation and reconsideration

dcrd lets the operator mark a block invalid (`invalidateblock`) and undo such marks (`reconsiderblock`). Both are local operations outside consensus. Invalidating a block on the main chain reorganizes to its parent, marks it and its descendants invalid, and then selects the best remaining candidate; the genesis block cannot be invalidated. Reconsidering a block clears the failure marks of the block and its ancestors and the invalid-ancestor marks of those descendants that do not descend from another failed block, so that they are validated again under the current rules, and then repeats chain selection. A related RPC forces a reorganization between two children of the same parent that have the required data and are not known invalid.

10.7.8 Assume-valid and other optimizations

dcrd skips some checks for blocks deep in the history of the chain. These are optimizations: they never cause a node to accept an invalid chain *provided* the block identified by the assume-valid hash is in fact valid, which the developers verify before each release. An independent implementation may omit them and validate everything.

RULE 10.52 Assume-valid ancestors
Let V be the header identified by the assume-valid hash (by default `AssumeValid`; the operator may change or disable it), H the header with the greatest cumulative work not known to be invalid, and E the expected number of blocks in two weeks (Rule 10.12). A block B is an **assume-valid ancestor** if V is in the index, $H.\text{height} \geq E$, B is an ancestor of (or equal to) H, and B is an ancestor of (or equal to) V', where V' is V if $V.\text{height} \leq H.\text{height} - E$ and otherwise the ancestor of V at height $H.\text{height} - E$. An implementation **MAY** omit the checks of Table 10.10 for an assume-valid ancestor, and **MUST NOT** omit them for any other block.

dcrd: internal/blockchain/process.go · isAssumeValidAncestor, maybeUpdateAssumeValid
internal/blockchain/validate.go · checkConnectBlock

For an assume-valid ancestor dcrd performs all header checks (including proof of work and positional difficulty), the data preconditions, block sanity, and the contextual checks *except* those listed in Table 10.10, and then connects it to the UTXO set by Algorithm 10.5 without Algorithm 10.8. The Merkle roots still bind every transaction to the header chain, and the header chain to V, so any alteration of the data would change V.

Table 10.10 Checks skipped for assume-valid ancestors

Stage	Skipped checks
Block positional	Transaction expiry (Rule 10.25).
Header contextual	Exact difficulty and ticket price, stake version, pool size and final state (Rule 10.15, Rule 10.16, Rule 10.17). The proof-of-work check under the correct hash function is still performed.
Block contextual	Finality (Rule 10.36), vote and revocation eligibility (Rule 10.37), treasury spend placement (part of Rule 10.32).
Connection	All of Algorithm 10.8: inputs, fees, subsidies, scripts, relative lock times, treasury spends, header commitment, block one ledger, per-tree signature operations.

> **NOTE**
> In dcrd the decision to apply the reduced contextual checks is made once, for the block being processed, and is then applied to every stored descendant that becomes fully linked by that block's arrival (Algorithm 10.12). A successful contextual stage is cached, and the cached result is used instead of repeating the stage when the block is connected. Consequently, a block that is not itself an assume-valid ancestor, but whose data was stored before the data of an assume-valid ancestor that completes its linkage, passes the contextual stage without the header and block contextual checks listed in Table 10.10, and those checks are not applied to it later unless the cached result has been evicted from the bounded cache. Unlike the assume-valid ancestor itself, such a descendant is not marked validated, so Algorithm 10.8, scripts included, still runs when it is connected. In a test with dcrd's own harness, a block one above the assume-valid block whose header committed to a wrong `PoolSize` became the tip when its data was delivered first, and was rejected with `ErrPoolSize` when delivered after its parent. The behavior depends on message arrival order and is not part of the protocol: by Rule 10.52 every block that is not an assume-valid ancestor is subject to all checks.

Independently of the above, Algorithm 10.8 skips script validation (steps 7 and 11) for a block that is an assume-valid ancestor at the time it is connected. dcrd also has a bulk-import mode, used when importing a trusted block file, with the same effect for every block.

> **NOTE**
> The **minimum known chain work** parameter `MinKnownChainWork` is not a validity rule. dcrd uses it, together with the age of the tip's timestamp (more than 24 hours behind adjusted time) and whether the main chain has reached H, only to decide whether it considers itself synchronized, which affects relay, mining and logging, not which blocks are valid.

10.7.9 Local time and consensus

The local clock enters validation only through Rule 10.4, at the moment a header is first accepted. Every other time comparison in this chapter is between timestamps recorded in headers (Rule 10.8, Rule 10.9, Rule 10.36, Rule 10.49). Once a block is more than two hours in the past by every honest clock, whether it is valid is fully determined by the chain.

10.8 Constants and Error Kinds

Table 10.11 Constants used in this chapter

Name	Value	Use
`MaxTimeOffsetSeconds`	7200	Rule 10.4
`medianTimeBlocks`	11	Algorithm 10.2
`MaxSigOpsPerBlock`	5000	Section 10.3.5
`MaxBlockPayload`	1310720	Rule 10.18
`MaxBlockHeaderPayload`	180	Header size
`MaxTAddsPerBlock`	20	Rule 10.32
`maxRevocationsPerBlock`	255	Rule 10.31
`testNet3MaxDiffActivationHeight`	962928	Rule 10.9, Rule 10.13
`minTestNetDiffBits` (testnet3)	`0x1d03ffff`	Rule 10.9
`maxAllowedOffsetSecs`	4200	Algorithm 10.1 (policy)
`maxMedianTimeEntries`	200	Algorithm 10.1 (policy)
Approval bit (`BlockValid`)	`0x0001`	Section 10.5

Table 10.12 Network parameters used in this chapter (main network values)

Parameter	Main network	Use
`MaximumBlockSizes`	393216	Algorithm 10.3
`MaxTxSize`	393216	Rule 10.22
`StakeValidationHeight`	4096	Throughout
`TicketsPerBlock`	5	Rule 10.46
`TargetTimePerBlock`	300 s	Two-week block count (4032)
`AssumeValid`	`458d6a8e11c916d4149ca8bc5c7aaaaf16cc61971b0c2076 4c07edf85df44eb6`	Rule 10.12, Rule 10.52
`MinKnownChainWork`	`0x243868232c14b8224643b6`	Section 10.7.8 (non-consensus)

TEST VECTOR Derived values
Produced from dcrd's network parameters.

```
expected blocks in two weeks: main 4032, testnet3 10080,
                              simnet 1209600, regnet 1209600
testnet3: PowLimitBits 1e00ffff, compact(PowLimit >> 6) = 1d03ffff
main network genesis:
  hash 298e5cc3d985bfe7f81dc135f360abe089edd4396b86d2de66b0cef42b21d980
  header Size 0, serialized size 300, VoteBits 0000, StakeRoot all zero
Merkle root of an empty tree: 32 zero bytes
```

Table 10.13 maps the error kinds of dcrd that arise in the stages of this chapter to the rules that produce them. Error kinds raised only by rules owned by other chapters are listed with that chapter.

Table 10.13 Error kinds and rules

Error kind	Rule
`ErrMissingParent`	Rule 10.6
`ErrInvalidAncestorBlock`, `ErrKnownInvalidBlock`	Rule 10.6, Rule 10.51
`ErrDuplicateBlock`	Algorithm 10.12 (not a validity rule)
`ErrInvalidTime`	Rule 10.3
`ErrTimeTooNew`	Rule 10.4
`ErrTimeTooOld`	Rule 10.8, Rule 10.9
`ErrBadBlockHeight`	Rule 10.7
`ErrForkTooOld`	Rule 10.12
`ErrBadMaxDiffCheckpoint`	Rule 10.13
`ErrBlockVersionTooOld`	Rule 10.11 (Chapter 13)
`ErrHighHash`, `ErrUnexpectedDifficulty`	Rule 10.2, Rule 10.10, Rule 10.14, Rule 10.15 (Chapter 9)
`ErrBadStakeVersion`	Rule 10.16 (Chapter 13)
`ErrPoolSize`, `ErrInvalidFinalState`	Rule 10.17 (Chapter 8)
`ErrBlockTooBig`	Rule 10.18, Rule 10.35
`ErrWrongBlockSize`	Rule 10.21
`ErrBadMerkleRoot`	Rule 10.19
`ErrBadCommitmentRoot`	Rule 10.38
`ErrNoTransactions`	Rule 10.20
`ErrDuplicateTx`	Rule 10.24
`ErrExpiredTx`	Rule 10.25
`ErrFirstTxNotCoinbase`, `ErrMultipleCoinbases`, `ErrCoinbaseHeight`	Rule 10.26, Rule 10.27

Error kind	Rule
`ErrFirstTxNotTreasurybase`, `ErrMultipleTreasurybases`	Rule 10.28
`ErrStakeTxInRegularTree`, `ErrRegTxInStakeTree`, `ErrNonstandardStakeTx`	Rule 10.30
`ErrIncongruentVotebit`	Rule 10.33
`ErrTooManySigOps`	Rule 10.34, Rule 10.40
`ErrUnfinalizedTx`	Rule 10.36, Rule 10.49
`ErrMissingTxOut`	Rule 10.40, Rule 10.41, Rule 10.42, Rule 10.43, Rule 10.44, Rule 10.38, and Chapter 7
`ErrBadFees`	Algorithm 10.9
`ErrBadCoinbaseAmountIn`, `ErrBadCoinbaseValue`	Rule 10.46
`ErrNoStakeTx`, `ErrBadTreasurybaseAmountIn`, `ErrSSGenSubsidy`, `ErrBadStakebaseValue`	Rule 10.47
`ErrScriptValidation`, `ErrScriptMalformed`	Rule 10.48
`ErrNotEnoughStake`, `ErrStakeBelowMinimum`, `ErrFreshStakeMismatch`, `ErrNotEnoughVotes`, `ErrTooManyVotes`, `ErrTooManySStxs`, `ErrInvalidEarlyStakeTx`, `ErrInvalidEarlyVoteBits`, `ErrInvalidEarlyFinalState`, `ErrVotesMismatch`, `ErrVotesOnWrongBlock`, `ErrRevocationsMismatch`, `ErrTooManyRevocations`, `ErrTicketUnavailable`, `ErrInvalidSSRtx`, `ErrNoMissedTicketRevocation`, `ErrNoExpiredTicketRevocation`	Rule 10.5, Rule 10.23, Rule 10.31, Rule 10.37 (Chapter 8)
`ErrTooManyTAdds`, `ErrNotTVI`, `ErrInvalidTVoteWindow`, `ErrNoTreasury`, `ErrTreasurybaseTxNotOpReturn`, `ErrTreasurybaseHeight`, treasury spend errors	Rule 10.28, Rule 10.32, Algorithm 10.8 (Chapter 12, Chapter 11)
`ErrBlockOneTx`, `ErrBlockOneInputs`, `ErrBlockOneOutputs`	Section 10.6.6 (Chapter 11)

CHAPTER 11

Issuance and Block Subsidy

New decred enters existence in exactly two ways: through the **block-one ledger**, a fixed list of payments made by the coinbase transaction of the block at height 1, and through the **block subsidy**, an amount created by every later block and divided between the proof-of-work miner, the stakeholders whose votes the block contains, and the project treasury. Transaction fees are not new money; they move existing atoms from transaction inputs to the miner.

This chapter specifies the subsidy schedule (the **full block subsidy** as a function of height and its periodic reduction), the three-way split of that subsidy and the two changes made to the split by stakeholder vote (DCP-0010, DCP-0012), the coinbase transaction that pays the proof-of-work share, the amount checks on the stake tree that pay the vote and treasury shares, the block-one ledger, and the resulting total supply. The formats of votes and of the treasurybase transaction are specified in Chapter 8 and Chapter 12; this chapter fixes only the amounts they carry.

All arithmetic in this chapter is on non-negative `int64` values, and every division truncates. The order of the multiplications and divisions is part of the consensus rules: reordering them, or replacing an iterated computation by a closed form, changes results by a few atoms and splits the chain.

11.1 Overview

Table 11.1 summarizes who receives each part of the subsidy, which transaction carries it, and how its amount depends on the block. The quantities named there are defined in Section 11.2 and Section 11.3.

Table 11.1 Where each share of the subsidy of the block at height h is paid

Share	Carried by	Amount and dependencies
Block-one ledger	Coinbase of block 1	The fixed list of outputs in `BlockOneLedger`; paid only at $h = 1$ (Section 11.6).
Proof of work	Coinbase (regular tree)	$W(h, v)$: the work proportion of $S(h)$, scaled by the number of votes v in the block from `StakeValidationHeight` on. Lost if the next block disapproves this block's regular tree.
Votes	Stakebase input of each vote (stake tree)	$V(h-1)$ per vote: the vote proportion of $S(h-1)$, divided by `TicketsPerBlock`. Based on the height of the block **voted on**, not the block containing the vote. Not scaled by v.
Treasury, before the `treasury` agenda	Output 0 of the coinbase	$T(h, v)$: one tenth of $S(h)$, scaled by v from `StakeValidationHeight` on. Lost if the block is disapproved.
Treasury, after the `treasury` agenda	Treasurybase (first stake transaction)	$T(h)$: one tenth of $S(h)$, not scaled by v. Unaffected by disapproval.

Table 11.2 lists the parameters and constants used. The full per-network tables are in Chapter 14; Table 11.3 summarizes the subsidy-related values of all four networks because the derived quantities given there (block-one subsidy, final subsidy interval) are computed in this chapter.

Table 11.2 Parameters and constants of the subsidy rules (main-network values)

Name	Value	Meaning
`BaseSubsidy`	3119582664	Full subsidy, in atoms, of every block in the first reduction interval (except blocks 0 and 1).
`MulSubsidy`	100	Reduction multiplier.
`DivSubsidy`	101	Reduction divisor.
`SubsidyReductionInterval`	6144	Length of a reduction interval in blocks.
`WorkRewardProportion`	6	Proof-of-work proportion of the original split.
`StakeRewardProportion`	3	Vote proportion of the original split.
`BlockTaxProportion`	1	Treasury proportion (all eras).
`TicketsPerBlock`	5	Maximum number of votes per block; the divisor in all vote scaling.
`StakeValidationHeight`	4096	First height whose blocks must contain votes.
`CoinbaseMaturity`	256	Blocks before coinbase outputs may be spent (Chapter 7).
`OrganizationPkScript`	`a914f5916158e3e2c4551c1796708db8367207ed13bb87`	Script that coinbase output 0 must pay before the `treasury` agenda (a pay-to-script-hash script).
`OrganizationPkScriptVersion`	0	Required script version of that output.
`BlockOneLedger`	3146 entries	Required outputs of the block-one coinbase (Section 11.6).
`MinCoinbaseScriptLen`	2	Minimum coinbase signature script length in bytes.
`MaxCoinbaseScriptLen`	100	Maximum coinbase signature script length in bytes.
`maxUniqueCoinbaseNullDataSize`	256	Maximum length of the data pushed by the coinbase height commitment.

Two derived quantities are used throughout. With N = `TicketsPerBlock`,

$$P = \texttt{WorkRewardProportion} + \texttt{StakeRewardProportion} + \texttt{BlockTaxProportion} = 10 \quad (11.1)$$

is the denominator of the original split and of the treasury share in every era, and

$$m = \left\lfloor \frac{N}{2} \right\rfloor + 1 = 3 \quad (11.2)$$

is the minimum number of votes a block at or above `StakeValidationHeight` must contain (enforced in Chapter 8).

Table 11.3 Subsidy parameters and derived quantities on each network

Network	`BaseSubsidy`	Interval	`StakeValidationHeight`	Block-one subsidy	First zero-subsidy height
mainnet	3119582664	6144	4096	168000000000000	11010048 (interval 1792)
testnet3	2500000000	2048	768	10000000000000	3622912 (interval 1769)
simnet	50000000000	128	144	30000000000000	265088 (interval 2071)
regnet	50000000000	128	144	30000000000000	265088 (interval 2071)

All four networks use `MulSubsidy` $= 100$, `DivSubsidy` $= 101$, proportions 6, 3 and 1, and `TicketsPerBlock` $= 5$.

NOTE
`chaincfg.Params` also carries fields named `WorkRewardProportionV2` (1) and `StakeRewardProportionV2` (8). They are deprecated and are not read by the consensus code: the proportions of the modified splits are fixed constants (Table 11.4), the same on every network. (The fields `WorkRewardProportion` and `StakeRewardProportion` carry the same deprecation notice in dcrd but are read: they define the original split.)

11.2 The Full Block Subsidy

The **full block subsidy** $S(h)$ is the total amount a block at height h may create before it is divided into shares. It is constant within a **reduction interval** of `SubsidyReductionInterval` consecutive blocks and is multiplied by $\frac{100}{101}$ at the start of each new interval, so that it decays geometrically, by roughly one percent every 6144 blocks (about three weeks on the main network). Interval k consists of the heights $k \cdot 6144$ through $k \cdot 6144 + 6143$, so interval 0 contains the genesis block and block 1, both of which are special.

Writing B, M, D and I for `BaseSubsidy`, `MulSubsidy`, `DivSubsidy` and `SubsidyReductionInterval`, and L for the sum of the amounts in `BlockOneLedger`,

$$S(h) = \begin{cases} 0 & \text{if } h \le 0 \\ L & \text{if } h = 1 \\ f\left(\left\lfloor \frac{h}{I} \right\rfloor\right) & \text{if } h \ge 2 \end{cases} \qquad f(k) = \begin{cases} B & \text{if } k = 0 \\ \left\lfloor \frac{f(k-1) \cdot M}{D} \right\rfloor & \text{if } k \ge 1. \end{cases} \tag{11.3}$$

RULE 11.1 Full block subsidy
The full block subsidy of height h **MUST** be computed by Algorithm 11.1. In particular the reduction **MUST** be applied once per completed interval, as a multiplication by `MulSubsidy` followed by a truncating division by `DivSubsidy`, each step operating on the result of the previous step. The genesis block has subsidy 0, and block 1 has the sum of the block-one ledger instead of `BaseSubsidy`.

dcrd: blockchain/standalone/subsidy.go · CalcBlockSubsidy

ALGORITHM 11.1 Full block subsidy

```
def block_subsidy(height: int64) -> int64:
    if height <= 0:
        return 0                        # genesis (and negative heights)
    if height == 1:
        return block_one_subsidy()      # sum of BlockOneLedger amounts; 0 if empty
    s: int64 = BaseSubsidy
    for _ in range(height // SubsidyReductionInterval):
        s = s * MulSubsidy              # never overflows for valid parameters
        s = s // DivSubsidy             # truncating; both operands non-negative
        if s == 0:
            break                       # all later intervals are also 0
    return s
```

dcrd: blockchain/standalone/subsidy.go · CalcBlockSubsidy

IMPLEMENTATION PITFALL
Iterated truncating division is not the same as the closed form $\left\lfloor B \cdot \left(\frac{M}{D}\right)^k \right\rfloor$. The two first differ in interval 2, where the iteration gives 3058114560 and the closed form 3058114561. The gap grows to 31 atoms by interval 100 and stays between 42 and 56 atoms from interval 200 on, and the closed form is still 56 atoms when the iterated value reaches zero. Exact rational or floating point evaluation of a power is therefore wrong. An implementation may cache per-interval values (dcrd does), but each cached value must be the result of the iteration.

11.2.1 When the Subsidy Reaches Zero

For $1 \le s \le 101$, $\left\lfloor 100\frac{s}{101} \right\rfloor = s - 1$. Once the full subsidy falls to 101 atoms or less it therefore decreases by exactly one atom per interval. On the main network S is 1 atom in interval 1791 (heights 11003904 to 11010047) and 0 from interval 1792 on: the first height with zero full subsidy is 11010048. The shares reach zero earlier, because each is a truncated fraction of S (Section 11.7 gives the last heights).

> **NOTE**
> The parameter documentation states that `BaseSubsidy` must not exceed $140739635871744 = \left\lfloor \frac{2^{63}-1}{2^{16}-1} \right\rfloor$ atoms, so that the products in this chapter cannot overflow `int64`. All networks are far below this bound, and no consensus check enforces it.

11.3 Splitting the Subsidy

The full subsidy of a block is divided among proof of work, votes and the treasury in fixed proportions. The proportions have changed twice by stakeholder vote; each set is called a **split variant**.

11.3.1 Split Variants and Their Selection

Table 11.4 Subsidy split variants. Each share is a fraction n/d of S.

Variant	Agenda	PoW w_n/w_d	Votes s_n/s_d	Treasury	Main-network activation
Original	none	6/10 (60%)	3/10 (30%)	1/10 (10%)	From genesis
DCP-0010	`changesubsidysplit`	10/100 (10%)	80/100 (80%)	1/10 (10%)	Block 657280
DCP-0012	`changesubsidysplitr2`	1/100 (1%)	89/100 (89%)	1/10 (10%)	Block 794368

In the original variant the work fraction is `WorkRewardProportion` $/P$ and the vote fraction is `StakeRewardProportion` $/P$, with P from Eq. (11.1); in the two modified variants they are the fixed constants shown, on every network. The treasury share is `BlockTaxProportion` $/P = 1/10$ in all three variants. The activation heights are those recorded in [38] and [39] for the main network; an implementation derives activation from the chain (Chapter 13), not from this table.

> **RULE 11.2 Selection of the split variant**
> The split variant used for every subsidy amount checked while connecting a block (its coinbase, its votes' stakebase inputs and its stake-tree totals) **MUST** be chosen from the agendas that are active **for that block**, that is, as determined from the state after its parent:
>
> 1. if `changesubsidysplitr2` (DCP-0012) is active for the block, the DCP-0012 variant;
> 2. otherwise, if `changesubsidysplit` (DCP-0010) is active for the block, the DCP-0010 variant;
> 3. otherwise, the original variant.
>
> The DCP-0012 variant takes precedence even if `changesubsidysplit` is not active. The same block's variant applies to the vote subsidy even though that subsidy is computed from the parent's height (Rule 11.3).
>
> dcrd: internal/blockchain/validate.go · checkConnectBlock
> internal/blockchain/agendas.go · isSubsidySplitAgendaActive, isSubsidySplitR2AgendaActive

Whether the `treasury` agenda (DCP-0006) is active for the block is likewise determined from the state after its parent, and that agenda is never active for blocks 0 and 1.

11.3.2 The Three Shares

Algorithm 11.2 defines the three share functions. `WORK[variant]` and `VOTE[variant]` are the pairs (n, d) of Table 11.4, and m is from Eq. (11.2). The argument `voters` is the number of votes in the block, which equals the `Voters` field of its header (Chapter 10); below `StakeValidationHeight` it is 0 and is ignored.

ALGORITHM 11.2 Proof-of-work, per-vote and treasury subsidies

```
P = WorkRewardProportion + StakeRewardProportion + BlockTaxProportion   # 10
MIN_VOTES = TicketsPerBlock // 2 + 1                                    # 3
WORK = {ORIGINAL: (WorkRewardProportion, P), DCP0010: (10, 100), DCP0012: (1, 100)}
VOTE = {ORIGINAL: (StakeRewardProportion, P), DCP0010: (80, 100), DCP0012: (89, 100)}

def work_subsidy(height: int64, voters: int, variant) -> int64:
    if height == 1:
        return block_subsidy(1)
    if height >= StakeValidationHeight and voters < MIN_VOTES:
        return 0
    n, d = WORK[variant]
    s = block_subsidy(height) * n // d                  # first truncation
    if height < StakeValidationHeight:
        return s                                        # no vote scaling yet
    return (voters * s) // TicketsPerBlock              # second truncation

def vote_subsidy(height: int64, variant) -> int64:      # amount per vote
    if height < StakeValidationHeight - 1:
        return 0
    n, d = VOTE[variant]
    return block_subsidy(height) * n // (d * TicketsPerBlock)   # one truncation

def treasury_subsidy(height: int64, voters: int, treasury_active: bool) -> int64:
    if height <= 1:
        return 0
    if height >= StakeValidationHeight and voters < MIN_VOTES:
        return 0
    s = block_subsidy(height) * BlockTaxProportion // P
    if height < StakeValidationHeight or treasury_active:
        return s
    return (voters * s) // TicketsPerBlock
```

dcrd: blockchain/standalone/subsidy.go · calcWorkSubsidy, CalcWorkSubsidyV3, calcStakeVoteSubsidy, CalcStakeVoteSubsidyV3, CalcTreasurySubsidy

In formulas, for $h \geq 2$ and v votes (at least m from `StakeValidationHeight` on, and 0 below it), with N = `TicketsPerBlock`:

$$W(h,v) = \left\lfloor \frac{v \cdot \left\lfloor S(h) \cdot \frac{w_n}{w_d} \right\rfloor}{N} \right\rfloor, \quad V(h) = \left\lfloor \frac{S(h) \cdot s_n}{s_d \cdot N} \right\rfloor, \quad T(h,v) = \left\lfloor \frac{v \cdot \left\lfloor \frac{S(h)}{10} \right\rfloor}{N} \right\rfloor, \tag{11.4}$$

where the outer scaling by $\frac{v}{N}$ is omitted below `StakeValidationHeight`, and for T also when the `treasury` agenda is active. Three details of Algorithm 11.2 are easy to miss:

- The work subsidy is truncated twice (once for the proportion, once for the vote scaling), in that order. Computing $\lfloor S \cdot w_n \cdot v/(w_d \cdot N) \rfloor$ instead gives different results: at height 6144 with three votes it gives 1111930454 instead of the correct 1111930453.
- The vote subsidy is $S \cdot s_n$ divided by $s_d \cdot N$. Because nested truncating divisions of non-negative integers compose, this equals the collective stake share $\lfloor S \cdot s_n / s_d \rfloor$ divided by N; what matters is that the collective share is never scaled by the number of votes.
- The treasury share always uses the denominator $P = 10$ derived from the original parameters; the modified variants change only the work and vote shares. Before the `treasury` agenda it is scaled by the number of votes like the work share; afterwards it is not.

RULE 11.3 The vote subsidy is based on the height voted on

The subsidy earned by each vote in the block at height h **MUST** be `vote_subsidy(h - 1, variant)`, where `variant` is the split variant of the block at height h (Rule 11.2). The proof-of-work and treasury subsidies of the same block use height h.

dcrd: internal/blockchain/validate.go · checkVoteInputs, checkStakeBaseAmounts, checkTransactionsAndConnect
blockchain/standalone/subsidy.go · calcStakeVoteSubsidy

IMPLEMENTATION PITFALL

Because of Rule 11.3, the votes in the first block of each reduction interval earn the subsidy of the **previous** interval, while the work and treasury shares of that block already use the reduced value. Under the modified variants the vote share dominates, so the total paid by such a block exceeds its own full subsidy: block 657408, the first block of interval 107, pays $107574538 + 5 \cdot 173840453 + 107574538 = 1084351341$ atoms against $S(657408) = 1075745381$. The same offset makes the votes in the first voting block, at height `StakeValidationHeight`, earn the subsidy of height `StakeValidationHeight` -1, which is why `vote_subsidy` returns a non-zero value from that height on. dcrd's source calls this basis unfortunate but it is part of consensus.

RATIONALE

The work subsidy is scaled by the number of votes so that a miner gains nothing by withholding votes from a block; each missing vote forfeits a fifth of the proof-of-work share. The vote subsidy is not scaled, since a vote that is not included simply earns nothing. DCP-0006 removed the vote scaling of the treasury share when that share moved to the treasurybase.

11.3.3 Rounding and Unissued Atoms

The shares are truncated independently, so with five votes $W + 5V + T$ is normally a few atoms less than S; those atoms are never created. Table 11.5 shows the shortfall at some heights. Two further amounts are never issued: the vote share of the blocks below `StakeValidationHeight` -1 (heights 2 to 4094 on the main network, which no vote ever votes on), and every share lost to missing votes, disapproved blocks, or coinbases that pay less than allowed.

Table 11.5 Shortfall $S - W - 5V - T$ in atoms, five votes, V evaluated at the same height

Height	$S(h)$	Original	DCP-0010	DCP-0012
4096	3119582664	5	2	2
6144	3088695706	3	6	4
614400	1153344657	3	2	6
1228800	426404419	2	2	4
6144000	148792	3	4	6

11.3.4 Test Vectors

The vectors below were produced by calling dcrd's `SubsidyCache` with the main-network parameters. Table 11.6 gives the work and treasury functions and Table 11.7 the per-vote function, each evaluated at the height in the first column (not shifted).

Table 11.6 Main network: $S(h)$, `work_subsidy` per variant and `treasury_subsidy` with the `treasury` agenda inactive and active

h	v	$S(h)$	W orig.	W 0010	W 0012	T inactive	T active
0	0	0	0	0	0	0	0
1	0	168000000000000	$=S(1)$	$=S(1)$	$=S(1)$	0	0
2	0	3119582664	1871749598	311958266	31195826	311958266	311958266
4095	0	3119582664	1871749598	311958266	31195826	311958266	311958266
4096	5	3119582664	1871749598	311958266	31195826	311958266	311958266
4096	4	3119582664	1497399678	249566612	24956660	249566612	311958266
4096	3	3119582664	1123049758	187174959	18717495	187174959	311958266
4096	2	3119582664	0	0	0	0	0

h	v	$S(h)$	W orig.	W 0010	W 0012	T inactive	T active
6144	5	3088695706	1853217423	308869570	30886957	308869570	308869570
6144	3	3088695706	1111930453	185321742	18532174	185321742	308869570
552448	4	1286750131	617640062	102940010	10294000	102940010	128675013
657280	5	1086502835	651901701	108650283	10865028	108650283	108650283
794368	3	864249748	311129908	51854984	5185498	51854984	86424974
1000000	5	622348887	373409332	62234888	6223488	62234888	62234888
5000000	4	956745	459237	76539	7653	76539	95674
10000000	3	243	87	14	1	14	24
11010047	5	1	0	0	0	0	0
11010048	5	0	0	0	0	0	0

Table 11.7 Main network: `vote_subsidy(h, variant)`, the amount per vote, at height h

h	Original	DCP-0010	DCP-0012
4094	0	0	0
4095	187174959	499133226	555285714
6143	187174959	499133226	555285714
6144	185321742	494191312	549787835
552448	77205007	205880020	229041523
657280	65190170	173840453	193397504
794368	51854984	138279959	153836455
1000000	37340933	99575821	110778101
5000000	57404	153079	170300
10000000	14	38	43

Table 11.8 combines the functions as consensus applies them to a whole block at height h: the work and treasury shares at h, the per-vote share at $h-1$, and the required coinbase input value (Rule 11.8). "Total" is $W + T + v \cdot V(h-1)$.

Table 11.8 Main network: subsidy amounts of a block at height h with v votes

h	v	Variant	Treasury agenda	Work	Per vote	Treasury	Coinbase input	Total
2	0	orig.	no	1871749598	0	311958266	2183707864	2183707864
4096	5	orig.	no	1871749598	187174959	311958266	2183707864	3119582659
4096	3	orig.	no	1123049758	187174959	187174959	1310224717	1871749594
6144	4	orig.	no	1482573938	187174959	247095656	1729669594	2478369430
552447	4	orig.	no	617640062	77205007	102940010	720580072	1029400100
552448	4	orig.	yes	617640062	77205007	128675013	617640062	1055135103
657280	5	0010	yes	108650283	173840453	108650283	108650283	1086502831
657408	5	0010	yes	107574538	173840453	107574538	107574538	1084351341
794368	3	0012	yes	5185498	153836455	86424974	5185498	553119837
798720	5	0012	yes	8556928	153836455	85569281	8556928	863308484

11.4 The Coinbase Transaction

The **coinbase** is the first transaction of a block's regular transaction tree. It has a single input that spends nothing, whose `ValueIn` field (part of the input's witness data, Chapter 4) commits to the subsidy being claimed, and its outputs pay the proof-of-work subsidy plus the block's fees to whatever scripts the miner

chooses. Before the `treasury` agenda its first output also pays the treasury share. Except in blocks 0 and 1, one output carries the block height, which makes every coinbase's transaction hash unique.

11.4.1 Identifying a Coinbase

ALGORITHM 11.3 Coinbase identification

```
def is_coinbase(tx, treasury_active: bool) -> bool:
    if treasury_active and tx.version != 3:
        return False
    if len(tx.inputs) != 1:
        return False
    prev = tx.inputs[0].prev_out
    if prev.index != 0xffffffff or prev.hash != bytes(32):
        return False                              # prev.tree is not examined here
    if treasury_active and len(tx.outputs) >= 2:  # exclude treasury-spend look-alikes
        sig = tx.inputs[0].sig_script
        s0 = tx.outputs[0].pk_script
        s1 = tx.outputs[1].pk_script
        if (len(sig) > 0 and len(s0) > 0 and len(s1) > 0 and sig[-1] == 0xc2
                and s0[0] == 0x6a and s1[0] == 0xc3):  # OP_TSPEND, OP_RETURN, OP_TGEN
            return False
    return True
```

dcrd: blockchain/standalone/tx.go · IsCoinBaseTx

Before the `treasury` agenda a coinbase may have any transaction version permitted by the general version rules (Chapter 7); once the agenda is active it must have version 3 (`TxVersionTreasury`), and a transaction of any other version is not a coinbase at all.

RULE 11.4 The first regular transaction is the only coinbase

For every block other than the genesis block, the first transaction of the regular tree **MUST** satisfy `is_coinbase(tx, treasury_active)`, where `treasury_active` tells whether the `treasury` agenda is active for the block. Every other transaction of the regular tree **MUST NOT** satisfy it.

dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 11.5 Coinbase input

For a regular-tree transaction that satisfies `is_coinbase`:

1. the previous outpoint of its input **MUST** be the null outpoint: hash of 32 zero bytes, index `0xffffffff` **and** tree 0 (regular);
2. the fraud-proof fields of its input **MUST** be null: `BlockHeight` $= 0$ (`NullBlockHeight`) and `BlockIndex` $=$ `0xffffffff` (`NullBlockIndex`);
3. the length of its signature script **MUST** be at least `MinCoinbaseScriptLen` $= 2$ and at most `MaxCoinbaseScriptLen` $= 100$ bytes.

The contents of the signature script are otherwise unconstrained. The input's `ValueIn` is constrained by Rule 11.8.

dcrd: internal/blockchain/validate.go · checkTransactionContext, isNullOutpoint, isNullFraudProof

The general transaction rules of Chapter 7 (sanity limits, version and script-version limits, lock time and expiry, the prohibition on stake opcodes in regular transactions) apply to the coinbase as to any other regular transaction. The coinbase has no real inputs, so input existence, script execution and sequence locks do not apply to it.

11.4.2 Required Outputs

The required leading outputs of the coinbase depend on whether the `treasury` agenda is active for the block (Table 11.9). Outputs after the required ones are free: any number of them, with any scripts and amounts allowed by the general rules, subject only to the total of Rule 11.9. They normally pay the miner.

Table 11.9 Leading outputs of the coinbase of a block at height $h \geq 2$

Output	`treasury` agenda not active	`treasury` agenda active
0	Treasury payment: script `OrganizationPkScript`, version `OrganizationPkScriptVersion`, amount exactly `treasury_subsidy(h, v, False)` (Rule 11.6)	Height commitment (Rule 11.7)
1	Height commitment (Rule 11.7)	Free
2, …	Free	Free

RULE 11.6 Treasury output before the treasury agenda

If the `treasury` agenda is not active for a block at height $h \geq 2$, its coinbase **MUST** have at least one output, and output 0 **MUST** have script version equal to `OrganizationPkScriptVersion`, a public key script byte-for-byte equal to `OrganizationPkScript`, and an amount equal to `treasury_subsidy(h, v, False)`, where v is the number of votes in the block. Blocks 0 and 1 are exempt.

dcrd: internal/blockchain/subsidy.go · coinbasePaysTreasuryAddress
internal/blockchain/validate.go · checkConnectBlock

Once the `treasury` agenda is active the coinbase carries no treasury payment; the treasury share is paid by the **treasurybase**, which must be the first transaction of the stake tree. Its format, its own height commitment and its effect on the treasury balance are specified in Chapter 12; its amount is fixed by Rule 11.12.

11.4.3 The Height Commitment

The height commitment is a provably unspendable output (Chapter 6) whose script is `OP_RETURN` followed by a single data push. The first four bytes of the pushed data are the block height as a little-endian `uint32`; any further bytes, up to 252 of them, are free and conventionally hold an **extra nonce** that miners vary to change the coinbase hash, and with it the block's Merkle root, without changing anything else.

RULE 11.7 Coinbase height commitment

For a block at height $h \geq 2$, let $i = 1$ if the `treasury` agenda is not active for the block and $i = 0$ if it is. The coinbase **MUST** have at least $i + 1$ outputs; output i **MUST** have script version 0; and its public key script, parsed by `commitment_data` of Algorithm 11.4, **MUST** yield between 4 and `maxUniqueCoinbaseNullDataSize` $= 256$ bytes inclusive, whose first four bytes, read as a little-endian `uint32`, **MUST** equal `uint32(h)`. The output's amount is not constrained by this rule. Blocks 0 and 1 are exempt.

dcrd: internal/blockchain/validate.go · checkCoinbaseUniqueHeight

ALGORITHM 11.4 Extracting the coinbase height commitment

```
def commitment_data(pk: bytes) -> bytes:
    # pk must be OP_RETURN followed by exactly one push opcode (0x00..0x4e)
    # and nothing else; any other form yields the empty string.
    if len(pk) < 2 or pk[0] != 0x6a:          # OP_RETURN
        return b""
    op, rest = pk[1], pk[2:]
    if op <= 0x4b:                            # OP_0 and OP_DATA_1..OP_DATA_75
        n, body = op, rest
    elif op == 0x4c and len(rest) >= 1:       # OP_PUSHDATA1
        n, body = rest[0], rest[1:]
    elif op == 0x4d and len(rest) >= 2:       # OP_PUSHDATA2
```

```
        n, body = le16_decode(rest[0:2]), rest[2:]
    elif op == 0x4e and len(rest) >= 4:       # OP_PUSHDATA4
        n, body = le32_decode(rest[0:4]), rest[4:]
    else:
        return b""
    if n != len(body):                        # truncated push or trailing bytes
        return b""
    return body

def check_height_commitment(coinbase, h: int, treasury_active: bool):
    if h < 2:
        return
    i = 0 if treasury_active else 1
    if len(coinbase.outputs) < i + 1:
        reject("missing height commitment output")
    out = coinbase.outputs[i]
    if out.script_version != 0:
        reject("height commitment script version")
    data = commitment_data(out.pk_script)
    if len(data) > 256 or len(data) < 4:
        reject("height commitment data length")
    if le32_decode(data[0:4]) != h % 2**32:
        reject("wrong height in coinbase")
```

dcrd: internal/blockchain/validate.go · checkCoinbaseUniqueHeight
txscript/tokenizer.go · ScriptTokenizer.Next

Minimal push encoding is not required: the height may be pushed with `OP_PUSHDATA1`, `OP_PUSHDATA2` or `OP_PUSHDATA4` as well as with a direct push, and the number is never interpreted with script-number semantics. Table 11.10 shows the 14-byte form that dcrd's block template generator produces (a policy choice). The treasurybase is required to use exactly this form (Chapter 12).

Table 11.10 Standard coinbase height-commitment script (policy; 14 bytes)

Field	Type	Bytes	Description
opcode	byte	1	`0x6a` (`OP_RETURN`).
push	byte	1	`0x0c` (`OP_DATA_12`).
height	uint32	4	Block height, little-endian (consensus: must equal the block height).
extra nonce	uint64	8	Random value chosen by the miner, little-endian.

11.4.4 Coinbase Amounts

The coinbase may pay out at most the subsidy it is entitled to plus the fees of the block, and its input must state the subsidy part exactly. The fees are the sum of the fees of all transactions of **both** trees of the block, as computed by the input checks of Chapter 7; the stake-tree fees (from tickets, votes, revocations, treasury adds and treasury spends) are added to those of the regular tree. A vote whose outputs total less than its inputs (the ticket amount plus the stakebase subsidy), as the fee limits of its ticket may allow (Chapter 8), therefore contributes the difference as a fee that the coinbase may claim. From `StakeValidationHeight` on, the fee total is scaled by the number of votes, so that a block with fewer than five votes can claim only part of its fees; the unclaimable part is destroyed.

ALGORITHM 11.5 Coinbase amount checks

```
def check_coinbase_amounts(block, h: int, voters: int, variant, treasury_active: bool,
                           stake_fees: int64, regular_fees: list[int64]):
    fees: int64 = stake_fees              # fees of all stake-tree transactions, summed
                                          # with the same overflow check as below
    for f in regular_fees:                # regular-tree fees (coinbase contributes 0)
```

```
        fees = fees + f
        if overflows_int64(fees):
            reject("total fees overflow")
    if h >= StakeValidationHeight:
        fees = fees * voters // TicketsPerBlock
    if h == 1:
        expected = block_subsidy(1)       # fees are not added at height 1
    else:
        work = work_subsidy(h, voters, variant)
        if treasury_active:
            expected = work + fees        # treasury share is in the treasurybase
        else:
            expected = work + treasury_subsidy(h, voters, False) + fees
    coinbase = block.regular[0]
    if coinbase.inputs[0].value_in != expected - fees:
        reject("coinbase ValueIn does not commit to the subsidy")
    if sum(out.value for out in coinbase.outputs) > expected:
        reject("coinbase pays more than subsidy plus fees")
```

dcrd: internal/blockchain/validate.go · checkTransactionsAndConnect, checkConnectBlock

RULE 11.8 Coinbase input value

The `ValueIn` of the coinbase input of a block at height $h \geq 1$ **MUST** equal `expected - fees` as computed by Algorithm 11.5. For $h \geq 2$ this is `work_subsidy(h, v, variant)` plus, while the `treasury` agenda is not active for the block, `treasury_subsidy(h, v, False)`. For $h = 1$ it is the block-one subsidy minus the block's fees.

dcrd: internal/blockchain/validate.go · checkTransactionsAndConnect

RULE 11.9 Coinbase output total

The sum of the amounts of all outputs of the coinbase of a block at height $h \geq 1$, including the treasury output and the height commitment where present, **MUST NOT** exceed `expected` as computed by Algorithm 11.5. Paying less is permitted; the difference is never created. The running fee total **MUST NOT** overflow `int64`.

dcrd: internal/blockchain/validate.go · checkTransactionsAndConnect

NOTE

At height 1 the fee total is necessarily zero: no output that a non-coinbase transaction could spend exists before block 1's coinbase outputs mature, so block 1 cannot contain fee-paying transactions. The rules are stated as implemented nonetheless. The individual coinbase outputs of block 1 are fixed by Rule 11.14.

IMPLEMENTATION PITFALL

The fee scaling uses the votes of the block whose coinbase is being checked and is applied to the combined fee total, with one truncation: for 999 atoms of fees and three votes the coinbase may claim $\lfloor 999 \cdot \frac{3}{5} \rfloor = 599$ atoms. The fees are those of the block's own two trees only; nothing from the parent block is included.

11.4.5 Disapproval and the Coinbase

When the votes of block $N+1$ disapprove block N (Chapter 10), every transaction of N's regular tree, the coinbase included, is disconnected as if it had never been mined, while N's stake tree remains in effect. Consequently:

- N's proof-of-work subsidy and all fees its coinbase claimed are never created. This includes the stake-tree fees of N, which stay spent because N's stake transactions remain valid.

- Before the `treasury` agenda, the treasury payment in output 0 of N's coinbase is lost as well. Afterwards the treasury share is paid by the treasurybase in the stake tree and survives disapproval, which DCP-0006 gives as an intended consequence.
- The vote subsidies paid by N's votes are unaffected.
- The non-coinbase transactions of N's regular tree, and their fees, return to the unconfirmed state and may be included again, by $N + 1$ or later.

The coinbase of block $N + 1$ is checked against $N + 1$'s own subsidy and fees by Algorithm 11.5 whether or not it disapproves N.

11.4.6 Maturity

Outputs of a coinbase cannot be spent until `CoinbaseMaturity` $= 256$ blocks have passed (the treasury balance and vote outputs have related maturity rules). These rules are specified in Chapter 7 and Chapter 12.

11.4.7 Coinbase Construction (Policy)

The following describes what dcrd's block template generator produces; none of it is required beyond the rules above. Its coinbase has version 1 before the `treasury` agenda and 3 afterwards; a signature script of two zero bytes followed by the ASCII text `/dcrd/`; input sequence `0xffffffff`; and the outputs, in order: the treasury payment (before the agenda), a zero-value height commitment of the form in Table 11.10 with a random extra nonce, and a single output paying the work subsidy to the miner's address. For blocks above height 1, once the block's transactions are selected, the fee total of both trees, scaled by the number of votes from `StakeValidationHeight` on exactly as in Algorithm 11.5, is added to that last output, so a dcrd-built coinbase claims its full entitlement. Its `ValueIn` is set as required by Rule 11.8 (the subsidy only, without the fees). Three cases differ: the coinbase of block 1 has only the ledger outputs; before the agenda, on a network whose `BlockTaxProportion` is zero, output 0 is a zero-value `OP_TRUE` script; and with no mining address configured the work output pays to `OP_TRUE`.

11.5 Subsidy Checks in the Stake Tree

Each vote (Chapter 8) has a **stakebase** input, its first input, which, like the coinbase input, spends nothing and whose `ValueIn` states the subsidy the vote creates. Its second input spends the ticket. The vote's outputs pay the ticket amount plus the subsidy to the addresses committed to by the ticket; how that total is divided among the outputs is specified in Chapter 8. The following rules fix the amounts.

RULE 11.10 Vote stakebase value

The `ValueIn` of the stakebase input of every vote **MUST** equal `vote_subsidy(H, variant)`, where H is the height of the block the vote votes on, as encoded in the vote itself, and `variant` is the split variant of the block containing the vote. Chapter 8 requires $H = h - 1$ for every vote in a block at height h.

dcrd: internal/blockchain/validate.go · checkVoteInputs

RULE 11.11 Vote output limits

For a block at height h containing votes, let $A =$ `vote_subsidy(h - 1, variant)`, and for each vote let its **excess** be the sum of its output amounts minus the amount of the ticket output its second input spends.

1. The excess of each vote **MUST NOT** exceed A.
2. The sum of the excesses of all votes in the block **MUST NOT** exceed $A \cdot v$ if $h \geq$ `StakeValidationHeight`, where v is the number of votes, and **MUST NOT** exceed 0 otherwise.

dcrd: internal/blockchain/validate.go · checkStakeBaseAmounts, getStakeBaseAmounts, checkTransactionsAndConnect

These two limits are implied by Rule 11.10 together with the general rule that outputs may not exceed inputs and the output commitments of Chapter 8, but dcrd checks them separately and they are consensus rules in their own right.

RULE 11.12 Treasurybase amount
If the `treasury` agenda is active for a block at height $h \geq 2$, both the amount of output 0 of its treasurybase and the `ValueIn` of the treasurybase's input **MUST** equal `treasury_subsidy(h, v, True)`, which does not depend on the number of votes v once $v \geq m$. All other properties of the treasurybase are specified in Chapter 12.

dcrd: internal/blockchain/subsidy.go · checkTreasuryBase
internal/blockchain/validate.go · checkTransactionsAndConnect

NOTE
dcrd maintains a running "total subsidy" in its chain state (reported by its RPC interface) by adding, for each block, the coinbase `ValueIn` of the parent if the block approves it, the treasurybase `ValueIn`, and the stakebase `ValueIn` of each vote. The figure counts claimed subsidy, not the amounts actually paid out, and is not used by any consensus rule.

11.6 The Block-One Ledger

Decred has no subsidy at height 1 in the usual sense. Instead, the coinbase of block 1 must pay exactly the outputs listed in the network's `BlockOneLedger`: an ordered list of entries, each a script version, a public key script and an amount. On the main network the ledger distributes 1680000 DCR, eight percent of 21 million: according to the Decred documentation [37], half to the project's founding developers in compensation for initial development and half airdropped to registered participants.

RULE 11.13 Block-one coinbase
If `BlockOneLedger` is non-empty, the coinbase of the block at height 1 **MUST** have lock time 0, expiry 0 (`NoExpiryValue`), and input sequence number `0xffffffff`.

dcrd: internal/blockchain/subsidy.go · blockOneCoinbasePaysTokens

RULE 11.14 Block-one outputs
If `BlockOneLedger` is non-empty, the coinbase of the block at height 1 **MUST** have exactly as many outputs as the ledger has entries, and for every index i, output i **MUST** have the script version, the public key script (byte for byte) and the amount of ledger entry i. The order is significant. If the ledger is empty this rule imposes nothing.

dcrd: internal/blockchain/subsidy.go · blockOneCoinbasePaysTokens

Together with Rule 11.8 and Rule 11.9 this fixes the coinbase of block 1 completely except for its transaction version and its signature script (2 to 100 bytes). Block 1 has no height commitment and no treasury output. All four networks have a non-empty ledger, and every entry on every network is a version-0 pay-to-pubkey-hash script of 25 bytes (`76 a9 14` ‖ 20-byte hash ‖ `88 ac`).

Table 11.11 Block-one ledgers. The digest is BLAKE-256 over the concatenation, in ledger order, of each entry serialized as $\mathrm{LE}_{64}(\text{amount}) \,\|\, \mathrm{LE}_{16}(\text{version}) \,\|\, \mathrm{varint}(|\text{script}|) \,\|\, \text{script}$, the serialization of the outputs in the block-one coinbase without the leading output count; it is given in natural byte order.

Network	Entries	Total (atoms)	Ledger digest
mainnet	3146	168000000000000	`fc5c40905a817a6d6d2407648cdbb5ad39a2875c31c309f5ee62d7e33da737ab`
testnet3	2	10000000000000	`80b7d8ec820546e6a40c7720435bf03f895812316bf3756d66dedb7cb0a04110`
simnet	3	30000000000000	`d6f72c64c85ed781166656e579807aede52e88aa97810fc3a359cfe29385020b`
regnet	3	30000000000000	`77b5ee4b9abcb50abb51322f4227aa9617524ac7f90dbbcb5f320f59a6fe1be2`

The serialized main-network list is 113256 bytes (3146×36). A second identifier follows from the fact that a Decred transaction hash covers only the transaction prefix, which excludes the signature script and `ValueIn`

(Chapter 4): the hash of block 1's coinbase is determined by the ledger and the transaction version. With version 1, as on the main network, it is, in display form,

`5e29cdb355b3fc7e76c98a9983cd44324b3efdd7815c866e33f6c72292cb8be6`

which matches the premine transaction identified in the Decred documentation [37].

Table 11.12 Main-network ledger: distribution of amounts

Amount (atoms)	Entries	Subtotal (atoms)
500000000000	167	83500000000000
28263795424	2972	84000000000128
201801801802	1	201801801802
80900900901	1	80900900901
72072072072	1	72072072072
63063063063	1	63063063063
54054054054	1	54054054054
18018018018	1	18018018018
10090089962	1	10090089962
	3146	168000000000000

The first main-network entry pays 500000000000 atoms to `76a91464e20eb6075561d30c23a517c5b73badbc120f0588ac` and the last pays 28263795424 atoms to `76a9145e4cde3d3d959bbc1efe5e5f0f63b9fbe00f486488ac`. The ledgers of the test and private networks are short enough to list in full (Table 11.13).

Table 11.13 Block-one ledgers of the test and private networks (all script version 0)

Network	Public key script	Amount (atoms)
testnet3	`76a914bb56576997d9f7abeebac585821b4d9a79d7ea0a88ac`	8000000000000
testnet3	`76a9147a5c4cca76f2e0b36db4763daacbd6cbb6ee6e7b88ac`	2000000000000
simnet	`76a91494ff37a0ee4d48abc45f70474f9b86f9da69a70988ac`	10000000000000
simnet	`76a914a6753ebbc08e2553e7dd6d64bdead4bcbff4fcf188ac`	10000000000000
simnet	`76a9147aa3211c2ead810bbf5911c275c69cc196202bd888ac`	10000000000000
regnet	`76a9147e4765ae88ba9ad5c9e4715c484e90b34d358d5188ac`	10000000000000
regnet	`76a91402fb1ac0137666d79165e13cecd403883615270788ac`	10000000000000
regnet	`76a91469de627d3231b14228653dd09cba75eeb872754288ac`	10000000000000

NOTE
In the dcrd source the ledger exists in two forms. `chaincfg/subsidydefinitions.go` holds the human-readable list of (script, amount) entries per network; it is compiled only with the build tag `subsidydefs` and serves as the input to the generator `chaincfg/generatesubsidytables.go`. The generator writes `chaincfg/subsidytables.go`, which stores, per network, all scripts concatenated into one hexadecimal string and a list of (end offset, amount) pairs; at start-up these are sliced back into the entries of `BlockOneLedger`. Only the generated file is compiled into the node, and it is consensus-critical through Rule 11.14 and the block-one subsidy. The two forms were checked to agree entry for entry on all four networks. Earlier versions kept the list in a file named `chaincfg/premine.go`, which no longer exists.

11.7 Total Supply

The supply of decred is the block-one ledger plus everything paid by the subsidy rules. For a given history of agenda activations it is largest when every block from `StakeValidationHeight` on contains five votes, no block is disapproved, and every coinbase, vote and treasurybase claims its full entitlement. Under those

assumptions, and with each split variant and the `treasury` agenda applied from its main-network activation height and never changed again (552448 for `treasury`, 657280 for DCP-0010, 794368 for DCP-0012: in each case the first block to which the new rules apply, the `Active` block of the deployment tables in [40], [38] and [39]), summing the per-block amounts of Table 11.8 over every height with dcrd's subsidy functions gives the figures of Table 11.14. They are an upper bound on the main-network supply under these assumptions, not a value fixed by any consensus rule.

Table 11.14 Maximum main-network supply under the stated assumptions (atoms)

Component	Atoms
Block-one ledger	168000000000000
Proof-of-work subsidy	779527937380932
Vote subsidy	958890472085010
Treasury subsidy	193583145837068
Total	**2100001555303010**
Sum of $S(h)$ over $h \geq 2$ (nominal, for comparison)	1935831507356784
Less: $S(h) - W - T$ of blocks 2 to 4095, which pay no vote share (4094 × 935874800; the table attributes $V(h)$ to the block at height h that contains the votes, and the last row corrects for votes being paid at the rate of the block they vote on)	3831471431200
Less: truncation $S(h) - W - 5V(h) - T$ of blocks from 4096 on	37531264
Plus: net effect of votes in block h earning $V(h-1)$ rather than $V(h)$	1556908690

The last four rows reconcile the total: $168000000000000 + 1935831507356784 - 3831471431200 - 37531264 + 1556908690 = 2100001555303010$. Further results under the same assumptions:

- The cumulative supply first exceeds 21×10^{14} atoms in block 8605295 (interval 1400). The maximum exceeds 21 million DCR by 1555303010 atoms, about 15.55 DCR, because of the one-block offset of the vote subsidy (Rule 11.3): without the last row of Table 11.14 the total would be 2099999998394320 atoms. The offset contributes, for each era of constant split variant, five times the per-vote subsidy at the start of the era minus five times that at its end ($609923945 + 177802470 + 769182275$ atoms for the three eras). dcrd's own test suite, which sums $V(h)$ instead of $V(h-1)$, checks the figure 2099999999800912 for the original variant applied forever.
- Under DCP-0012 the proof-of-work subsidy is last non-zero at height 10401791, the treasury subsidy at height 10954751, and the vote subsidy in block 10979328, whose five votes earn one atom each. No block after 10979328 creates any subsidy, although the full subsidy S stays positive until height 11010047.
- The realized supply is lower than the maximum: every missing vote forfeits one vote subsidy and part of the proof-of-work share (and, before the `treasury` agenda, part of the treasury share), every disapproved block forfeits its coinbase, and coinbases may claim less than allowed. `MaxAmount` (Chapter 2) bounds individual amounts and is not a supply cap; no consensus rule limits the cumulative supply.

Table 11.15 Main-network schedule by reduction interval (five votes per block). Per-block shares are for a block whose votes vote within the same interval, under the variant in force at the start of the interval (original through interval 106, DCP-0010 for intervals 107 to 129, DCP-0012 from interval 130); "Issued" is the total created in the interval and "Cumulative" includes the ledger. Interval 0 starts its vote share at height 4096.

k	First height	S	Work	Per vote	Treasury	Issued	Cumulative
0	0	3119582664	1871749598	187174959	311958266	15329005280848	183329005280848
1	6144	3088695706	1853217423	185321742	308869570	18976955665317	202305960946165
2	12288	3058114560	1834868736	183486873	305811456	18789065012553	221095025958718
3	18432	3027836198	1816701718	181670171	302783619	18603034647158	239698060605876
4	24576	2997857621	1798714572	179871457	299785762	18418846204706	258116906810582
5	30720	2968175862	1780905517	178090551	296817586	18236481376082	276353388186664

k	First height	S	Work	Per vote	Treasury	Issued	Cumulative
10	61440	2824117486	1694470491	169447049	282411748	17351386294046	364862896567950
25	153600	2432552105	1459531263	145953126	243255210	14945607418487	605440778188861
50	307200	1896827356	1138096413	113809641	189682735	11654112947312	934590216157870
89	546816	1286750131	772050078	77205007	128675013	7905796634394	1309421832808688
100	614400	1153344657	692006794	69200679	115334465	7086153014211	1391386193254264
106	651264	1086502835	651901701	65190170	108650283	6675476671217	1432453826242617
107	657408	1075745381	107574538	172119260	107574538	6609388196109	1439063214438726
129	792576	864249748	86424974	138279959	86424974	5309957348048	1569006292865055
130	798720	855692819	8556928	152313321	85569281	5257384264886	1574263677129941
150	921600	701278308	7012783	124827538	70127830	4308660135012	1669136082867336
200	1228800	426404419	4264044	75899986	42640441	2619832520760	1838018828956872
300	1843200	157646462	1576464	28061070	15764646	968581253295	2003143923576476
500	3072000	21548080	215480	3835558	2154808	132391583007	2086762823622948
750	4608000	1790825	17908	318766	179082	11002814020	2098901612484732
1000	6144000	148792	1487	26484	14879	914142509	2099910396925214
1250	7680000	12320	123	2192	1232	75663475	2099994163967890
1500	9216000	979	9	174	97	5996554	2100001050474236
1700	10444800	92	0	16	9	546816	2100001531476498
1782	10948608	10	0	1	1	36864	2100001555180125
1786	10973184	6	0	1	0	30720	2100001555303005
1787	10979328	5	0	0	0	5	2100001555303010
1791	11003904	1	0	0	0	0	2100001555303010
1792	11010048	0	0	0	0	0	2100001555303010

Intervals 89, 106 and 129 contain an activation (of `treasury`, DCP-0010 and DCP-0012 respectively); their "Issued" figures include both rules. The figures in this section other than S are exact consequences of the rules only under the stated assumptions; S and the share functions themselves are exact for every chain.

11.8 Summary of Checks

Table 11.16 maps each rule of this chapter to the error kind dcrd reports when it is violated, for use when testing an implementation against dcrd.

Table 11.16 Rules of this chapter and the corresponding dcrd error kinds

Rule	dcrd error kind(s)
Rule 11.4	`ErrFirstTxNotCoinbase`, `ErrMultipleCoinbases`
Rule 11.5	`ErrBadCoinbaseOutpoint`, `ErrBadCoinbaseFraudProof`, `ErrBadCoinbaseScriptLen`
Rule 11.7	`ErrFirstTxNotCoinbase` (missing output, wrong version, malformed or wrongly sized push), `ErrCoinbaseHeight` (wrong height)
Rule 11.6	`ErrNoTxOutputs`, `ErrNoTreasury`
Rule 11.8	`ErrBadCoinbaseAmountIn`
Rule 11.9	`ErrBadCoinbaseValue`, `ErrBadFees`
Rule 11.10	`ErrBadStakebaseAmountIn`
Rule 11.11	`ErrSSGenSubsidy` (per vote), `ErrBadStakebaseValue` (total)
Rule 11.12	`ErrTreasurybaseOutValue` (output), `ErrBadTreasurybaseAmountIn` (input)
Rule 11.13	`ErrBlockOneTx` (lock time, expiry), `ErrBlockOneInputs` (sequence)

Rule	dcrd error kind(s)
Rule 11.14	`ErrBlockOneOutputs`

CHAPTER 12

The Decentralized Treasury

A fixed share of every block subsidy is reserved for the development of the project. Since the activation of DCP-0006 [40] this share is not paid to an address but credited to an on-chain **treasury account**: a balance that exists only as consensus state, that no private key controls, and that can be debited only by a **treasury spend** transaction which is both signed by one of a small set of well-known keys and approved by the stakeholders through the ordinary proof-of-stake votes. This chapter specifies the account, the three transaction types that change it, the voting procedure for spends, and the three successive expenditure policies that limit how fast the account may be drawn down.

12.1 Background and Overview

12.1.1 The legacy arrangement

From block 2 until the activation of the `treasury` agenda, the coinbase transaction of every block was required to pay the treasury share of the subsidy in its output 0 to a fixed, network-specific script, `OrganizationPkScript` (on the main network a 2-of-3 pay-to-script-hash script controlled by a conventional legal entity), and from stake validation height on the amount was reduced in proportion to the number of votes in the block. The exact amounts and the coinbase rules of that era are specified in Chapter 11; the relevant facts here are that these payments created ordinary unspent outputs, that spending them required only the keys behind the organization script, and that they lived in the regular transaction tree, so that a block whose regular tree was disapproved by the next block's voters also lost its treasury payment.

> **RATIONALE**
> DCP-0006 gives the motivation: the legacy treasury put sovereignty over spending in the hands of the key holders rather than the stakeholders, exposed the funds to theft, coercion or loss of key holders, and was insufficiently transparent. The decentralized treasury converts the treasury from an output model to an account model whose debits require a stakeholder vote.

12.1.2 The account model

With the `treasury` agenda active, value enters and leaves the treasury through three new kinds of stake transaction, all of transaction version 3 (`TxVersionTreasury`) and all carried in the stake tree:

Table 12.1 Treasury transaction types

Type	Created by	Per block	Effect on the treasury account
Treasurybase	The block's miner	exactly one, first in the stake tree	Credits the treasury share of the block subsidy. Replaces the legacy coinbase output.
Treasury add (TAdd)	Anyone	at most 20	Burns ordinary coins (its inputs) and credits the amount of its `OP_TADD` output.

Type	Created by	Per block	Effect on the treasury account
Treasury spend (TSpend)	Holders of a Politeia key (Pi key)	only in treasury vote interval blocks	Debits the treasury by its input value and creates `OP_TGEN`-tagged outputs paying the recipients.

The balance of the account is not stored in any output. Every full node computes it from the chain (Section 12.4). Credits and debits are subject to coinbase maturity: the value changes made by a block take effect in the balance `CoinbaseMaturity` blocks later.

A treasury spend names its own voting window through its `Expiry` field. Ticket holders vote yes or no on it by appending an optional output to their ordinary vote transactions (Section 12.8). The spend may be included only in a block whose height is a multiple of the **treasury vote interval** (TVI), only within its window, only once on a given chain, only with a quorum of votes and a sufficient yes majority, and only if the total spent stays within the expenditure policy in force (Section 12.9).

12.1.3 Activation

The rules of this chapter apply to a block exactly when the `treasury` agenda (DCP-0006) **is active for the block** in the sense of Chapter 13. In dcrd the question is always asked of the block's parent: the agenda is active for block B when it is active "as of the block after" B's parent. The agenda is never active for the genesis block or for block 1, regardless of network.

> **HISTORY**
> On the main network the `treasury` agenda became active with block 552448; `revertttreasurypolicy` (DCP-0007) with block 657280; and `maxtreasuryspend` (DCP-0013) with block 1052416. These heights are informative. An implementation derives activation from the votes in the chain, as described in Chapter 13, and the other networks activate at different heights.

The balance of the treasury account starts at zero. Activation did not move any funds held under the legacy organization script: dcrd contains no code that credits a legacy balance, and the account balance is defined solely by the treasury transactions in blocks for which the agenda is active. Moving legacy funds into the account is possible only through ordinary treasury add transactions, which is one of the uses DCP-0006 names for them.

Activation of the `treasury` agenda changes several rules owned by other chapters. They are listed in Table 12.2 for orientation and those that concern treasury transactions are specified in Section 12.10.

Table 12.2 Rules changed by the `treasury` agenda

Change	Specified in
The coinbase transaction must have version 3; it no longer pays the treasury, and its height-encoding null-data output moves from output 1 to output 0.	Chapter 11
The treasury subsidy is no longer reduced by the number of votes in the block.	Chapter 11
The opcodes 0xc1, 0xc2 and 0xc3 are interpreted as `OP_TADD`, `OP_TSPEND` and `OP_TGEN`, and are forbidden in the outputs of regular transactions.	Chapter 6, Rule 12.30
`OP_TSPEND` counts as a signature operation.	Chapter 6, Section 12.10.1
Vote transactions may carry a final treasury-vote output.	Chapter 8, Section 12.8.2
Outputs tagged `OP_SSGEN` or `OP_SSRTX` require `CoinbaseMaturity` confirmations instead of `SStxChangeMaturity`.	Rule 8.22
Treasurybase, treasury add and treasury spend transactions are accepted in the stake tree.	this chapter

12.2 Parameters and Constants

Table 12.3 lists the network parameters and constants used in this chapter with their main-network values. Abbreviations used in formulas are given in the second column. The values for the other networks are tabulated in Chapter 14.

Table 12.3 Treasury parameters and constants (main network)

Name	Symbol	Value	Meaning
`TreasuryVoteInterval`	TVI	288	Treasury spends may only be included in blocks whose height is a nonzero multiple of TVI.
`TreasuryVoteIntervalMultiplier`	TVIM	12	Length of a voting window in TVIs.
`TreasuryVoteQuorumMultiplier`	TVQM	1	Quorum numerator.
`TreasuryVoteQuorumDivisor`	TVQD	5	Quorum denominator (quorum 20%).
`TreasuryVoteRequiredMultiplier`	TVRM	3	Required yes fraction numerator.
`TreasuryVoteRequiredDivisor`	TVRD	5	Required yes fraction denominator (60%).
`TreasuryExpenditureWindow`	TEW	2	Length of an expenditure policy window in voting windows.
`TreasuryExpenditurePolicy`	TEP	6	Number of prior policy windows averaged (DCP-0006 policy only).
`TreasuryExpenditureBootstrap`	TEB	1600000000000	Average assumed when no prior window has spends (DCP-0006 policy only), in atoms (16000 DCR).
`PiKeys`		2 keys	Compressed secp256k1 public keys allowed to sign treasury spends (Table 12.4).
`CoinbaseMaturity`	CM	256	Maturity of treasury credits and debits and of `OP_TGEN` outputs.
`TicketsPerBlock`	TPB	5	Votes per block; bounds the treasury vote count.
`StakeValidationHeight`	SVH	4096	Treasury spend voting windows may not start before this height.
`BaseSubsidy`		3119582664	Initial block subsidy; defines the DCP-0013 floor.
`TxVersionTreasury`		3	Transaction version of all treasury transactions.
`MaxTAddsPerBlock`		20	Maximum number of treasury adds per block.
`TSpendScriptLen`		100	Exact length of a treasury spend signature script.
`OP_TADD`, `OP_TSPEND`, `OP_TGEN`		0xc1, 0xc2, 0xc3	Treasury opcodes (Chapter 6).

Table 12.4 Main-network Pi keys (`PiKeys`), as 33-byte compressed public keys

Index	Public key
0	`03f6e7041f1cf51ee10e0a01cd2b0385ce3cd9debaabb2296f7e9dee9329da946c`
1	`0319a37405cb4d1691971847d7719cfce70857c0f6e97d7c9174a3998cf0ab86dd`

Several derived quantities recur. With the main-network values:

Voting window length $\mathrm{TVI} \cdot \mathrm{TVIM} = 3456$ blocks.
Expenditure policy window W $\mathrm{TVI} \cdot \mathrm{TVIM} \cdot \mathrm{TEW} = 6912$ blocks.
Maximum votes in a window $\mathrm{TPB} \cdot \mathrm{TVI} \cdot \mathrm{TVIM} = 17280$.
Quorum $\left\lfloor 17280 \cdot \frac{\mathrm{TVQM}}{\mathrm{TVQD}} \right\rfloor = 3456$ votes.
Lower bound on the expiry of an includable spend $2 + \mathrm{SVH} + \mathrm{TVI} \cdot \mathrm{TVIM} = 7554$ (Rule 12.21). The bound is not itself a valid expiry, because Expiry $- 2$ must be a multiple of TVI; the smallest valid expiry is 7778, with window $[4320, 7776]$.
DCP-0013 spend floor $\left\lfloor \frac{\mathrm{BaseSubsidy}}{10} \right\rfloor \cdot \mathrm{TVI} \cdot \mathrm{TVIM} = 1078127767296$ atoms.

TEST VECTOR Derived treasury quantities on every network

network	floor	W	maxVotes	quorum	bound	minExp	TVI	TVIM	CM	SVH
mainnet	1078127767296	6912	17280	3456	7554	7778	288	12	256	4096
testnet3	60000000000	960	1200	240	1010	1022	60	4	16	768
simnet	720000000000	576	720	144	290	290	48	3	16	144
regnet	60000000000	48	60	12	158	158	4	3	16	144

`bound` is $2 + \mathrm{SVH} + \mathrm{TVI} \cdot \mathrm{TVIM}$; `minExp` is the smallest expiry at or above it with $\mathrm{Expiry} - 2$ a multiple of TVI.

12.3 Classification of Treasury Transactions

Every transaction is classified as exactly one of: regular, ticket, vote, revocation, treasury add, treasury spend or treasurybase. The ticket, vote and revocation tests (Chapter 8) are applied first. A transaction that is none of these is tested for the three treasury types only if its version is at least 3, in the order treasury add, treasury spend, treasurybase; the first structural predicate that holds determines the type, and a transaction that satisfies none is regular. The three structural predicates are defined in Algorithm 12.3, Algorithm 12.4 and Algorithm 12.2. Each requires version exactly 3, and the three are mutually exclusive, so the order matters only for performance.

Classification is a pure function of the transaction and does not depend on whether the `treasury` agenda is active. The following two rules make the agenda's state matter.

RULE 12.1 Treasury transactions only in the stake tree
A block's regular transaction tree **MUST NOT** contain any transaction that is classified as anything other than regular, including treasury add, treasury spend and treasurybase transactions. This rule applies whether or not the `treasury` agenda is active for the block.

dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 12.2 No treasury transactions before activation
If the `treasury` agenda is not active for a block, its stake tree **MUST NOT** contain any transaction classified as treasury add, treasury spend or treasurybase. (dcrd counts the stake transactions of each type it accepts, counting the treasury types only when the agenda is active, and rejects the block when the counts do not add up to the number of stake transactions.)

dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 12.3 Permitted stake transactions before stake validation height
In a block whose height is below `StakeValidationHeight`, every stake transaction **MUST** be a ticket, a treasury add or a treasurybase (the latter two only when the agenda is active). In particular a treasury spend is never valid below stake validation height.

dcrd: internal/blockchain/validate.go · checkBlockContext

The coinbase test of the regular tree (Chapter 11) also depends on the agenda and on the treasury spend format, because a treasury spend, like a coinbase, has a single input that references the null outpoint.

RULE 12.4 Coinbase detection with the treasury active
When the `treasury` agenda is active, a transaction is a coinbase only if, in addition to the conditions of Chapter 11 (exactly one input whose previous outpoint has index $2^{32} - 1$ and the all-zero hash), its version is 3 and it is **not treasury-spend-like**. A transaction is treasury-spend-like when it has at least two outputs, its input 0 signature script and the scripts of outputs 0 and 1 are all non-empty, the last byte of the input 0 signature script is `OP_TSPEND` (0xc2), the first byte of the output 0 script is `OP_RETURN`

(0x6a), and the first byte of the output 1 script is `OP_TGEN` (0xc3). Note that a treasurybase satisfies this coinbase test; dcrd excludes it explicitly wherever the distinction matters, as noted below.

dcrd: blockchain/standalone/tx.go · IsCoinBaseTx

12.4 The Treasury Balance

12.4.1 Treasury values of a block

Each block for which the `treasury` agenda is active carries a list of **treasury values**: signed amounts, in atoms, that change the treasury balance. Credits are positive and debits negative. The list is derived from the block's stake tree in transaction order, as in Algorithm 12.1. The **net change** of a block is the sum of its values. A block for which the agenda is not active has no treasury values.

ALGORITHM 12.1 Treasury values of a block

```
def treasury_values(block) -> list[tuple[str, int64]]:
    values = []
    for tx in block.stake_transactions:            # in block order
        t = classify(tx)
        if t == "tadd":
            values.append(("tadd", tx.outputs[0].value))   # change ignored
        elif t == "treasurybase":
            values.append(("tbase", tx.outputs[0].value))
        elif t == "tspend":
            total_out = 0
            for out in tx.outputs[1:]:                     # output 0 is the OP_RETURN
                values.append(("tspend", -out.value))
                total_out += out.value
            # The fee, recorded as a non-positive debit.
            values.append(("fee", total_out - tx.inputs[0].value_in))
    return values

def is_debit(kind) -> bool:
    return kind == "tspend" or kind == "fee"

def net_change(block) -> int64:
    return sum(amount for (_, amount) in treasury_values(block))
```

dcrd: internal/blockchain/treasury.go · dbPutTreasuryBalance

For a treasury spend the values sum to exactly $-$`ValueIn` of its single input: the payouts plus the recorded fee, which is `ValueIn` minus the payouts. The transaction fee (`ValueIn` minus the sum of all output values) is paid out of the treasury and, like every other stake-tree fee, is available to the block's coinbase (Chapter 11). The two differ only by the value of output 0, which is a null-data output and is therefore destroyed.

12.4.2 Balance and maturity

RULE 12.5 Treasury balance

Let B be a block at height H for which the `treasury` agenda is active, and let A be the height of the earliest block on B's chain for which the agenda is active. The **treasury balance as of B** is

$$\mathrm{bal}(B) = \sum_{h=A}^{H-\mathrm{CM}} \mathrm{net}(B_h), \tag{12.1}$$

where B_h is B's ancestor at height h and the sum is zero when $H - \mathrm{CM} < A$. Equivalently, $\mathrm{bal}(B) = 0$ if the agenda is not active for B's parent or if $H - \mathrm{CM} < A$, and otherwise $\mathrm{bal}(B) = \mathrm{bal}(\mathrm{parent}(B)) + \mathrm{net}(B_{H-\mathrm{CM}})$.

dcrd: internal/blockchain/treasury.go · calculateTreasuryBalance
internal/blockchain/chain.go · connectBlock

The balance as of B therefore includes the values of B's ancestor `CoinbaseMaturity` blocks back but none of the values of the CM most recent blocks, B itself included. Credits (treasurybases and treasury adds) and debits (treasury spends and their fees) mature alike. With $\mathrm{CM} = 256$, the treasurybase of the first active block on the main network, 552448, first counts in the balance of block 552704.

The balance does not depend on the approval of any block by the following block's voters. All treasury transactions are in the stake tree, which a disapproval vote never removes (Chapter 10). This is a deliberate change from the legacy arrangement, in which a disapproved block lost its treasury payment.

IMPLEMENTATION PITFALL

The balance used to check a treasury spend (Rule 12.25) does not reflect debits from treasury spends mined in the preceding $\mathrm{CM} - 1$ blocks, because they have not yet matured. On the main network, testnet and simnet $\mathrm{TVI} > \mathrm{CM}$, so the spends of one TVI block have matured by the next. On regnet ($\mathrm{TVI} = 4$, $\mathrm{CM} = 16$) they have not. An implementation must use exactly the balance of Rule 12.5 and must not subtract immature debits. On regnet the balance of a later block can therefore become negative once the spends of several TVI blocks less than CM apart mature, each of which passed Algorithm 12.10; no consensus rule checks the balance of a block that contains no treasury spend. dcrd cannot record a negative balance in its treasury state and fails to connect such a block with an internal database error, not with a rule violation.

12.5 Treasurybase

The treasurybase carries the treasury share of the block subsidy into the account. It has a single input that, like a coinbase input, references no previous output and whose `ValueIn` states the amount created; an `OP_TADD` output whose value is that amount; and a null-data output that commits to the block height, which makes every treasurybase hash distinct.

Table 12.5 Treasurybase transaction

Field	Type	Bytes	Description
`Version`	`uint16`	2	3.
`TxIn[0].PreviousOutPoint`	`outpoint`	37	Null outpoint: hash all zero, index 0xffffffff, tree 0 (regular).
`TxIn[0].Sequence`	`uint32`	4	Unconstrained by consensus (dcrd writes 0xffffffff).
`TxIn[0].ValueIn`	`int64`	8	The treasury subsidy for the block.
`TxIn[0].BlockHeight`	`uint32`	4	0x00000000 (null fraud proof).
`TxIn[0].BlockIndex`	`uint32`	4	0xffffffff (null fraud proof).
`TxIn[0].SignatureScript`	`varbytes`	1	Empty.
`TxOut[0].Value`	`int64`	8	The treasury subsidy for the block.
`TxOut[0].Version`	`uint16`	2	0.
`TxOut[0].PkScript`	`varbytes`	2	`c1` (`OP_TADD`), exactly one byte.
`TxOut[1].Value`	`int64`	8	0 (implied by the amount rules).
`TxOut[1].Version`	`uint16`	2	0.
`TxOut[1].PkScript`	`varbytes`	15	14 bytes: `6a 0c` followed by LE_{32}(height) and 8 arbitrary bytes.
`LockTime, Expiry`	`uint32`	8	Subject only to the general finality and expiry rules (Chapter 7).

ALGORITHM 12.2 Treasurybase structure

```
NULL_OUTPOINT = (bytes(32), 0xffffffff, 0)        # (hash, index, tree)

def is_treasurybase(tx) -> bool:                    # full predicate (stake package)
    if tx.version != 3:                            return False
    if len(tx.inputs) != 1 or len(tx.outputs) != 2: return False
    if len(tx.inputs[0].signature_script) != 0:    return False
    if tx.inputs[0].previous_outpoint != NULL_OUTPOINT: return False
    if any(out.script_version != 0 for out in tx.outputs): return False
    if tx.outputs[0].pk_script != bytes([0xc1]):   return False   # OP_TADD only
    return is_strict_null_data(tx.outputs[1], 12)

def is_strict_null_data(out, n) -> bool:            # n <= 75
    s = out.pk_script
    return out.script_version == 0 and len(s) == 2 + n and s[0] == 0x6a and s[1] == n

def is_treasurybase_minimal(tx) -> bool:            # minimal predicate (standalone)
    return (tx.version == 3 and len(tx.inputs) == 1 and len(tx.outputs) == 2
            and len(tx.inputs[0].signature_script) == 0
            and tx.outputs[0].pk_script == bytes([0xc1])
            and len(tx.outputs[1].pk_script) == 14
            and tx.outputs[1].pk_script[0] == 0x6a
            and tx.outputs[1].pk_script[1] == 0x0c
            and tx.inputs[0].previous_outpoint == NULL_OUTPOINT)
```

dcrd: blockchain/stake/treasury.go · CheckTreasuryBase
blockchain/standalone/tx.go · IsTreasuryBase

`is_strict_null_data` is stated here for the lengths used in this chapter (12 and 32). For these lengths the general strict null-data test of Chapter 6 reduces to the form shown: `OP_RETURN` followed by the single canonical push opcode equal to the length, and nothing else. The minimal predicate differs from the full one only in not checking output script versions. It is used by the positional rules below. A transaction that passes the minimal predicate but not the full one is never valid, since it is then classified as regular, and a regular transaction is rejected in the stake tree (Rule 10.30) and fails the regular-tree checks elsewhere.

RULE 12.6 Treasurybase required and unique
If the `treasury` agenda is active for a block, its stake tree **MUST** be non-empty and its first transaction **MUST** satisfy the minimal treasurybase predicate of Algorithm 12.2, and no other transaction in the stake tree **MAY** satisfy that predicate.

dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 12.7 Treasurybase form
The first stake transaction of a block for which the agenda is active **MUST** be classified as a treasurybase, so it **MUST** satisfy the full predicate of Algorithm 12.2. As a treasurybase it **MUST** additionally have a null fraud proof (`BlockHeight` = 0 and `BlockIndex` = 0xffffffff) and an empty signature script.

dcrd: blockchain/stake/treasury.go · CheckTreasuryBase
internal/blockchain/validate.go · checkTransactionContext

RULE 12.8 Treasurybase height commitment
For a block at height $H \geq 2$ for which the agenda is active, output 1 of the treasurybase **MUST** have script version 0 and a script of exactly 14 bytes beginning `6a 0c`, and bytes 2 to 5 of that script, read as a little-endian `uint32`, **MUST** equal H (as `uint32`). The remaining 8 bytes are unconstrained.

dcrd: internal/blockchain/validate.go · checkTreasurybaseUniqueHeight

RULE 12.9 Treasurybase amount

Let S be the treasury subsidy for the block, computed as in Chapter 11 with the treasury agenda active (that is, not reduced by the number of votes). For a block at height $H \geq 2$ for which the agenda is active:

- the treasurybase **MUST** have exactly two outputs, output 0 **MUST** have script version 0 and the one-byte script `OP_TADD`, and its value **MUST** equal S;
- the `ValueIn` of its input **MUST** equal S;
- the sum of its output values **MUST NOT** exceed its `ValueIn`. With the two preceding conditions, this forces output 1 to have value 0.

dcrd: internal/blockchain/subsidy.go · checkTreasuryBase
internal/blockchain/validate.go · checkTransactionsAndConnect
internal/blockchain/validate.go · CheckTransactionInputs

RULE 12.10 Treasurybase creates no outputs

Connecting a treasurybase spends nothing and adds none of its outputs to the set of unspent transaction outputs. Its only effect on state is its treasury value (Algorithm 12.1).

dcrd: internal/blockchain/utxoviewpoint.go · connectStakeTransaction

A treasurybase has no signature operations (Section 12.10.1). Its input is not checked against any previous output and no script is executed for it.

TEST VECTOR Treasurybase, main network, height 1052416, 5 votes

Treasury subsidy at height 1052416: 56903836 atoms (with the treasury agenda active; the legacy rule would give 34142301 with 3 votes). The 8 arbitrary bytes are `01 02 ... 08`.

Serialized: `0300000001000000000000000000000000000000000000000000000000000000000000000000000000ffffffff00ffffffff029c486403000000000000001c10000000000000000000000e6a0c000f10000102030405060708000000000000000019c486403000000000000000ffffffff00`

Transaction hash (display form): `9e57dfea4b6be2bcbab6c05e1b6a481b486fb7a81bcd2aeb168865128f202b5b`

12.6 Treasury Add

A treasury add moves existing coins into the treasury account. Anyone may create one. It spends ordinary outputs, exactly like a regular transaction, and pays the amount to be credited into an `OP_TADD` output that is never spendable. Change, if any, must go to a stake change output.

Table 12.6 Treasury add transaction

Field	Type	Bytes	Description
`Version`	`uint16`	2	3.
`TxIn[0..n-1]`	`inputs`	var	One or more ordinary inputs spending existing outputs, with ordinary signature scripts.
`TxOut[0]`	`output`	var	Value: the amount credited to the treasury. Script version 0, script exactly `c1` (`OP_TADD`).
`TxOut[1]`	`output`	var	Optional. Stake change: script version 0, `OP_SSTXCHANGE` (0xbd) followed by a P2PKH or P2SH script; value nonzero.

ALGORITHM 12.3 Treasury add structure

```
def is_p2pkh(s) -> bool:   # OP_DUP OP_HASH160 <20> OP_EQUALVERIFY OP_CHECKSIG
    return (len(s) == 25 and s[0] == 0x76 and s[1] == 0xa9 and s[2] == 0x14
            and s[23] == 0x88 and s[24] == 0xac)
```

```
def is_p2sh(s) -> bool:    # OP_HASH160 <20> OP_EQUAL
    return len(s) == 23 and s[0] == 0xa9 and s[1] == 0x14 and s[22] == 0x87

def is_tagged(out, opcode) -> bool:
    s = out.pk_script
    return (out.script_version == 0 and len(s) >= 1 and s[0] == opcode
            and (is_p2pkh(s[1:]) or is_p2sh(s[1:])))

def is_tadd(tx) -> bool:
    if tx.version != 3:                          return False
    if len(tx.inputs) < 1:                       return False
    if len(tx.outputs) not in (1, 2):            return False
    for out in tx.outputs:
        if out.script_version != 0 or len(out.pk_script) == 0:
            return False
    if tx.outputs[0].pk_script != bytes([0xc1]): return False    # OP_TADD only
    if len(tx.outputs) == 2 and not is_tagged(tx.outputs[1], 0xbd):
        return False                                             # OP_SSTXCHANGE
    return True
```

dcrd: blockchain/stake/treasury.go · CheckTAdd | blockchain/stake/scripttype.go · isTaggedScript

RULE 12.11 Treasury add change

If a treasury add has two outputs, the value of output 1 **MUST NOT** be zero.

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 12.12 Treasury add inputs

No input of a treasury add **MAY** reference the null outpoint. Its inputs are subject to every input rule that applies to stake transactions (Chapter 7): they **MUST** reference existing unspent outputs, with correct fraud-proof fields, satisfying all maturity requirements; they **MUST NOT** spend outputs created in the same block (in either tree), except the historical outputs of Rule 8.23; they **MUST NOT** spend ticket (`OP_SSTX`) outputs; and the sum of the output values **MUST NOT** exceed the sum of the input values.

dcrd: internal/blockchain/validate.go · checkTransactionContext
internal/blockchain/validate.go · CheckTransactionInputs

RULE 12.13 Treasury add limit

A block **MUST NOT** contain more than `MaxTAddsPerBlock` = 20 treasury adds.

dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 12.14 Treasury add outputs are unspendable

Output 0 of a treasury add is never added to the set of unspent transaction outputs. In addition, when the agenda is active for a block, no input of any transaction in it **MAY** spend an output whose script version is 0 and whose script is exactly the single byte 0xc1. (This matters only for such outputs created by regular transactions before activation, when 0xc1 was an unknown opcode.)

dcrd: internal/blockchain/utxoviewpoint.go · addTxOuts
internal/blockchain/validate.go · CheckTransactionInputs

The fee of a treasury add (inputs minus outputs) is a stake-tree fee and is available to the block's coinbase. The change output is an ordinary `OP_SSTXCHANGE` output and matures under the general rule for stake change, after `SStxChangeMaturity` blocks, or after `CoinbaseMaturity` blocks if the treasury add has a nonzero `Expiry` (Chapter 7). The amount of output 0 may be zero; no rule forbids it. Treasury adds are permitted below stake validation height.

12.7 Treasury Spend

A treasury spend debits the account. It has a single input that references no previous output. Its `ValueIn` is the total amount taken from the treasury, and its signature script carries a Schnorr signature by one of the Pi keys. Output 0 commits to that amount and makes the transaction unique. Each further output pays a recipient.

12.7.1 Format

Table 12.7 Treasury spend transaction

Field	Type	Bytes	Description
`Version`	`uint16`	2	3.
`TxIn[0].PreviousOutPoint`	`outpoint`	37	Null outpoint: hash all zero, index 0xffffffff, tree 0.
`TxIn[0].Sequence`	`uint32`	4	Bit 31 (0x80000000, relative lock time disabled) must be set (Rule 12.19). dcrd writes 0xffffffff.
`TxIn[0].ValueIn`	`int64`	8	Total debited from the treasury: sum of payouts plus fee.
`TxIn[0].BlockHeight`	`uint32`	4	0x00000000 (null fraud proof).
`TxIn[0].BlockIndex`	`uint32`	4	0xffffffff (null fraud proof).
`TxIn[0].SignatureScript`	`varbytes`	101	Exactly 100 bytes, Table 12.8.
`TxOut[0]`	`output`	45	Value: any (normally 0). Script version 0. Script: 34 bytes, Table 12.9.
`TxOut[1..n]`	`outputs`	var	At least one. Script version 0. Script: `OP_TGEN` (0xc3) followed by a P2PKH (25-byte) or P2SH (23-byte) script.
`LockTime`	`uint32`	4	Subject to the general finality rule.
`Expiry`	`uint32`	4	Determines the voting window (Section 12.8.1).

Table 12.8 Treasury spend signature script (100 bytes)

Offset	Bytes	Content
0	1	`0x40` (`OP_DATA_64`)
1	64	Schnorr signature, EC-Schnorr-DCRv0 encoding $r \parallel s$ (Chapter 3). No hash-type byte.
65	1	`0x21` (`OP_DATA_33`)
66	33	Pi public key, compressed: first byte `0x02` or `0x03`.
99	1	`0xc2` (`OP_TSPEND`)

Table 12.9 Treasury spend output 0 script (34 bytes)

Offset	Bytes	Content
0	1	`0x6a` (`OP_RETURN`)
1	1	`0x20` (`OP_DATA_32`)
2	8	LE_{64} of `TxIn[0].ValueIn` (two's complement).
10	24	Arbitrary bytes. dcrd uses random bytes so that otherwise identical spends have distinct hashes.

ALGORITHM 12.4 **Treasury spend structure**

```
def check_tspend(tx) -> tuple[bytes, bytes] | None:
    """Returns (signature, pubkey) if tx has the treasury spend form, else None."""
    if tx.version != 3:                                  return None
    if len(tx.inputs) != 1 or len(tx.outputs) < 2:       return None
    for out in tx.outputs:
        if out.script_version != 0 or len(out.pk_script) == 0:
            return None
    s = tx.inputs[0].signature_script
```

```
    if len(s) != 100 or s[0] != 0x40 or s[65] != 0x21 or s[99] != 0xc2:
        return None
    signature, pubkey = s[1:65], s[66:99]
    if pubkey[0] not in (0x02, 0x03):                return None
    if not is_strict_null_data(tx.outputs[0], 32):   return None
    for out in tx.outputs[1:]:
        p = out.pk_script
        if p[0] != 0xc3:                             return None   # OP_TGEN
        if not (is_p2pkh(p[1:]) or is_p2sh(p[1:])):  return None
    return signature, pubkey

def is_tspend(tx) -> bool:
    return check_tspend(tx) is not None
```

dcrd: blockchain/stake/treasury.go · CheckTSpend

Note that the structural predicate does not test the previous outpoint of the input. It is tested by the next rule, so a transaction of the treasury spend form whose input references a real output is classified as a treasury spend and then rejected.

12.7.2 Transaction rules

RULE 12.15 Treasury spend input

The input of a treasury spend **MUST** reference the null outpoint (all-zero hash, index 0xffffffff, tree 0), **MUST** have a null fraud proof (`BlockHeight` = 0, `BlockIndex` = 0xffffffff), and its signature script **MUST** be exactly 100 bytes long.

dcrd: internal/blockchain/validate.go · checkTransactionContext

RULE 12.16 Treasury spend key

If the agenda is active for the block, the 33-byte public key at offset 66 of a treasury spend's signature script **MUST** be byte-for-byte equal to one of the network's `PiKeys`.

dcrd: internal/blockchain/validate.go · CheckTransactionInputs | chaincfg/params.go · PiKeyExists

RULE 12.17 Treasury spend signature

If the agenda is active for the block, the 64 bytes at offset 1 of the signature script **MUST** parse as an EC-Schnorr-DCRv0 signature, the public key **MUST** parse as a secp256k1 point, and the signature **MUST** verify under that key for the message hash

$$m = \mathrm{SigHash}(\mathrm{SigHashAll}, \mathrm{tx}, 0, \mathrm{subscript} = \mathrm{empty}), \tag{12.2}$$

that is, the signature hash of type `SigHashAll` (0x01) for input index 0 computed as specified in Chapter 6 with an empty subscript. Signature parsing and verification are specified in Chapter 3.

dcrd: internal/blockchain/treasury.go · verifyTSpendSignature
txscript/sign/sign.go · TSpendSignatureScript

The signature hash does not cover the input's `ValueIn` (no Decred signature hash does). It covers every output, and output 0 commits to `ValueIn`, which Rule 12.18 requires to match; the key holder therefore fixes the exact amount, the recipients and the window (through `Expiry`). The signature script is not covered (signature scripts never are), and it is never executed by the script engine: script validation skips every input whose previous outpoint index is 0xffffffff (Chapter 6), so the signature is checked only by Rule 12.17.

RULE 12.18 Treasury spend amounts

For a treasury spend in a block for which the agenda is active:

- its `ValueIn` **MUST** satisfy $0 \le \mathrm{ValueIn} \le \mathrm{MaxAmount}$;

- the eight bytes at offsets 2 to 9 of the output 0 script, read as a little-endian `uint64` and reinterpreted as `int64`, **MUST** equal `ValueIn`;
- the sum of all its output values **MUST NOT** exceed `ValueIn`. The difference is the fee.

dcrd: internal/blockchain/validate.go · checkTreasurySpendInputs
internal/blockchain/validate.go · CheckTransactionInputs

RULE 12.19 Treasury spend sequence number

If the `lnfeatures` agenda (relative lock times, Chapter 7) is active for the block, bit 31 (0x80000000, `SequenceLockTimeDisabled`) of the `Sequence` field of a treasury spend's input **MUST** be set.

dcrd: internal/blockchain/sequencelock.go · calcSequenceLock
internal/blockchain/validate.go · checkConnectBlock

IMPLEMENTATION PITFALL

Rule 12.19 is not stated in DCP-0006. It follows from the general relative lock-time rule. That rule is evaluated for every stake transaction of version 2 or higher that is neither a coinbase nor a vote, and for each input without the disable bit it looks up the referenced output. A treasury spend is not a coinbase (Rule 12.4) and its input references the null outpoint, which never exists, so the block is rejected (dcrd reports `ErrMissingTxOut`). The behaviour was confirmed by running dcrd with a treasury spend whose sequence number is 0. The treasurybase is a coinbase under Rule 12.4 and is exempt. On the main network `lnfeatures` became active long before `treasury`, so the rule applies to every treasury spend there. dcrd's memory pool does not evaluate sequence locks for treasury spends (policy), so it may accept treasury spends that no block may contain.

12.7.3 Inclusion in a block

The following rules restrict where a treasury spend may appear. They reference the voting window and vote tally defined in Section 12.8 and the expenditure check of Section 12.9.

RULE 12.20 Treasury spends only on a TVI

If the agenda is active for a block at height $H > 1$ and $H \bmod \mathrm{TVI} \neq 0$, the block **MUST NOT** contain any treasury spend.

dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 12.21 Full voting window after stake validation height

If the agenda is active for a block at height $H > 1$ with $H \bmod \mathrm{TVI} = 0$, every treasury spend in the block **MUST** have Expiry $\geq 2 + \mathrm{SVH} + \mathrm{TVI} \cdot \mathrm{TVIM}$ (compared as unsigned 64-bit integers), so that its voting window starts at or after stake validation height.

dcrd: internal/blockchain/validate.go · checkBlockContext

RULE 12.22 Treasury spend inside its window

Every treasury spend in a block at height H **MUST** have a valid expiry (Rule 12.27) and **MUST** satisfy $\mathrm{start} \leq H \leq \mathrm{end}$, where $(\mathrm{start}, \mathrm{end})$ is its voting window and the comparison is on the `uint32` value of H.

dcrd: internal/blockchain/validate.go · tspendChecks
blockchain/standalone/treasury.go · InsideTSpendWindow

RULE 12.23 Treasury spend not previously mined

A block **MUST NOT** contain a treasury spend whose transaction hash equals that of a treasury spend contained in any of the block's ancestors. Within a single block the general rule against duplicate transactions applies (Chapter 10).

dcrd: internal/blockchain/validate.go · tspendChecks
internal/blockchain/treasury.go · checkTSpendExists

RULE 12.24 Treasury spend approved
Every treasury spend in a block **MUST** pass the vote check of Algorithm 12.8 for that block.

dcrd: internal/blockchain/validate.go · tspendChecks
internal/blockchain/treasury.go · checkTSpendHasVotes

RULE 12.25 Treasury expenditure
For a block at height H with $H \bmod \mathrm{TVI} = 0$ for which the agenda is active, let T be the sum of the `ValueIn` of all treasury spends in the block (as `int64`; the block is invalid if the sum overflows). If $T > 0$, the block **MUST** pass Algorithm 12.10.

dcrd: internal/blockchain/validate.go · tspendChecks
internal/blockchain/treasury.go · checkTSpendsExpenditure

A treasury spend has no inputs to spend, so connecting it adds its outputs to the set of unspent outputs and marks nothing spent. Output 0 is a null-data script and is never added, and neither is a payout of value zero (Chapter 7).

RULE 12.26 Maturity of treasury payouts
If the agenda is active for a block, an input in it that spends an output whose script is `OP_TGEN`-tagged (script version 0, first byte 0xc3, followed by a P2PKH or P2SH script) **MUST** satisfy $h_{\text{spend}} - h_{\text{origin}} \geq \mathrm{CM}$, where h_{spend} is the height of the spending block and h_{origin} the height of the block containing the treasury spend.

dcrd: internal/blockchain/validate.go · CheckTransactionInputs

Because a treasury spend always has a nonzero `Expiry`, its outputs are also subject to the general rule that outputs of transactions with an expiry require `CoinbaseMaturity` confirmations (Chapter 7), which imposes the same bound.

TEST VECTOR Signed treasury spend (simnet)
Simnet Pi key 0, private key `62deae1ab2b1ebd96a28c80e870aee325bed359e83d8db2464ef999e616a9eef`, public key `02a36b785d584555696b69d1b2bbeff4010332b301e3edd316d79438554cacb3e7`. One payout of 10000000000 atoms to the P2PKH hash `11` ×20, fee 10000, so `ValueIn` = 10000010000. The arbitrary bytes of output 0 are `00 01 ... 17`. Expiry 1154 (computed for a next block height of 1000 on simnet: window [1008, 1152]). Sequence 0xffffffff, lock time 0.

Unsigned serialization (empty signature script): `0300000001000000000000000000000000000000000000000000000000000000000000ffffffff00ffffffff020000000000000000000226a20100b0c540200000000010203040506070809 0a0b0c0d0e0f10111213141516170 0e40b540200000000001ac376a914111111111111111111111111111111111111188ac0000000082040000011 00b0c540200000000000000ffffffff00`

Signature hash (`SigHashAll`, input 0, empty subscript): `49e42bae2fc61c807b3f3e8dd9cc67b3fdc04bc9da01084731c97d256e0dae8d`

Signature ($r \parallel s$, deterministic): `bdf258f7e1441340a762bac301884a2237b1b05c3e26a58487258e6c360f1bdc7ee535af48c50bbb1f59a61c09fc0024c578b41ec2e1e61aca7c08f704671780`

Signed serialization: `0300000001000000000000000000000000000000000000000000000000000000000000ffffffff00ffffffff020000000000000000000226a20100b0c540200000000010203040506070809 0a0b0c0d0e0f10111213141516170 0e40b540200000000001ac376a914111111111111111111111111111111111111188ac0000000082040000011 00b0c540200000000000000ffffffff6440bdf258f7e1441340a762bac301884a2237b1b05c3e26a58487258e6c360f1bdc7ee535af48c50bbb1f59a61c09fc0024c578b41ec2e1e61aca7c08f7046717802102a36b785d584555696b69d1b2bbeff4010332b301e3edd316d79438554cacb3e7c2`

Transaction hash (display form): `8dfae941b11684372a19d55c100ef79c130ab7676b8cd546f89664e7b0b2a6ea`

Decreasing the payout value by one atom without re-signing makes verification fail.

12.8 Treasury Spend Voting

12.8.1 The voting window

A treasury spend carries no explicit window. The window is derived from its `Expiry` field, which therefore has a restricted form: it must be exactly two more than a nonzero multiple of TVI and large enough for a whole window to fit above height zero.

RULE 12.27 Treasury spend expiry and window

The expiry e (a `uint32`) of a treasury spend is valid if and only if $e \geq \mathrm{TVI} \cdot \mathrm{TVIM} + 2$ and $(e - 2) \bmod \mathrm{TVI} = 0$. The voting window of a treasury spend with a valid expiry is the pair

$$\mathrm{start} = e - \mathrm{TVI} \cdot \mathrm{TVIM} - 2, \quad \mathrm{end} = e - 2. \tag{12.3}$$

A treasury spend with an invalid expiry is outside every window and can never be included in a block.

dcrd: blockchain/standalone/treasury.go · CalcTSpendWindow

Both bounds are multiples of TVI, and $\mathrm{end} - \mathrm{start} = \mathrm{TVI} \cdot \mathrm{TVIM}$. The window is used in two different ways:

- **Votes** are counted from blocks at heights start through $H - 1$, where $H \leq \mathrm{end}$ is the height of the block that includes the spend. Votes in a block at height end or above can never count, so for voting the window is the half-open interval $[\mathrm{start}, \mathrm{end})$.
- **Inclusion** is allowed in TVI blocks at heights $\mathrm{start} \leq H \leq \mathrm{end}$, an interval closed at both ends. Since no votes have been counted at $H = \mathrm{start}$ and the quorum is positive on every network, a spend can never be included at $H = \mathrm{start}$. Even if every ticket votes yes, the earliest inclusion is at the first TVI H with $\mathrm{TPB} \cdot (H - \mathrm{start}) \geq \left\lfloor \mathrm{TPB} \cdot \mathrm{TVI} \cdot \mathrm{TVIM} \cdot \frac{\mathrm{TVRM}}{\mathrm{TVRD}} \right\rfloor$ ($\mathrm{start} + 2304$ on the main network, see Algorithm 12.8).

The `+ 2` exists because of transaction expiry. A transaction is invalid in a block at height h when $\mathrm{Expiry} \neq 0$ and $h \geq \mathrm{Expiry}$ (Chapter 7), and the memory pool removes transactions one block early. With $\mathrm{end} = e - 2$, a spend remains valid, and in the memory pool, at its last possible inclusion height. The general expiry rule never rejects a treasury spend that satisfies Rule 12.22, since then $H \leq e - 2 < e$.

Wallets and the Pi key holders compute the expiry for a new spend from the height of the next block n as in Algorithm 12.5. Nothing requires this choice; it yields the earliest window that starts strictly after n.

ALGORITHM 12.5 Window computation and conventional expiry

```
def tspend_window(expiry: uint32) -> tuple[uint32, uint32] | None:
    if expiry < TVI * TVIM + 2:
        return None                                  # invalid expiry
    if (expiry - 2) % TVI != 0:                      # expiry - 2 > 0 here
        return None
    return expiry - TVI * TVIM - 2, expiry - 2       # (start, end)

def inside_tspend_window(height: int64, expiry: uint32) -> bool:
    w = tspend_window(expiry)
    if w is None:
        return False
    start, end = w
    return start <= uint32(height) <= end

# Non-normative: the expiry chosen by wallets for a spend created when the
# next block will be at height n (n >= 0).
def tspend_expiry(n: int) -> uint32:
    next_tvi = n + (TVI - n % TVI)                   # strictly greater than n
    return uint32(next_tvi + TVI * TVIM + 2)
```

dcrd: blockchain/standalone/treasury.go · CalcTSpendWindow
blockchain/standalone/treasury.go · CalcTSpendExpiry
blockchain/standalone/treasury.go · InsideTSpendWindow

TEST VECTOR Voting windows (main network: TVI 288, TVIM 12)

```
next height n   expiry    start     end       note
1052415         1056098   1052640   1056096
1052416         1056098   1052640   1056096   (activation block of DCP-0013)
1052640         1056386   1052928   1056384   n is itself a TVI

expiry 3457     invalid   (below TVI*TVIM + 2 = 3458)
expiry 3458     start 0, end 3456
expiry 3459     invalid   (3457 not a multiple of 288)
expiry 1056097  invalid;  expiry 1056096  invalid

inside(h, expiry 1056098): h=1052640 true, 1052641 true, 1056096 true, 1056097 false
```

12.8.2 The treasury vote output

A ticket holder votes on treasury spends through the vote transaction it already casts when its ticket is selected (Chapter 8). With the `treasury` agenda, a vote transaction may end with one extra null-data output that lists (treasury spend hash, vote) pairs. Chapter 8 specifies the rest of the vote format and refers here for this output: its vote predicate calls `parse_treasury_votes` of Algorithm 12.6. In a vote with only two outputs the last output is the vote bits output itself, so that output is then the one parsed (Chapter 8).

Table 12.10 Treasury vote output (last output of a vote)

Offset	Bytes	Content
0	1	`0x6a` (`OP_RETURN`)
1	1 or 2	Push opcode: `OP_DATA_N` with $N = 2 + 33k \leq 75$ (so $k \leq 2$), or `0x4c` (`OP_PUSHDATA1`) followed by a length byte $L = 2 + 33k \leq 255$ (so $k \leq 7$).
p	2	Discriminator `54 56` (ASCII `TV`), where $p = 2$ for `OP_DATA_N` and 3 for `OP_PUSHDATA1`.
$p + 2 + 33i$	32	Hash of the i-th treasury spend voted on, in internal byte order (not reversed).
$p + 34 + 33i$	1	Vote: `0x01` yes or `0x02` no.

ALGORITHM 12.6 Treasury vote output parsing

```
def is_null_data(out) -> bool:
    # Script version 0 and: OP_RETURN alone, or OP_RETURN followed by
    # exactly one push (OP_0, OP_1..OP_16, OP_DATA_1..OP_DATA_75,
    # OP_PUSHDATA1/2/4) that consumes the
    # rest of the script and pushes at most 256 bytes. Non-canonical pushes
    # are allowed. Parsing is as in the script chapter.
    ...

def parse_treasury_votes(s: bytes) -> list[tuple[bytes, int]]:
    # s is a script already known to be null data
    if len(s) < 4 or s[1] > 0x4c:                  # beyond OP_PUSHDATA1
        reject("ErrSSGenInvalidDiscriminatorLength")
    p = 3 if s[1] == 0x4c else 2
    if p + 2 > len(s):
        reject("ErrSSGenInvalidNullScript")
    if s[p:p+2] != b"TV":                          # 0x54 0x56
        reject("ErrSSGenUnknownDiscriminator")
    body = s[p+2:]
    if len(body) < 33 or len(body) % 33 != 0:
        reject("ErrSSGenInvalidTVLength")
```

```
    votes, seen = [], set()
    for i in range(0, len(body), 33):
        h, v = body[i:i+32], body[i+32]
        if v not in (0x01, 0x02):                    # 0x01 yes, 0x02 no
            reject("ErrSSGenInvalidTreasuryVote")
        if h in seen:                                # no duplicate tspend
            reject("ErrSSGenDuplicateTreasuryVote")
        seen.add(h)
        votes.append((h, v))
    return votes

def treasury_votes(vote_tx) -> list[tuple[bytes, int]]:
    last = vote_tx.outputs[-1]
    if not is_null_data(last):
        return []                                    # no treasury vote output
    votes = parse_treasury_votes(last.pk_script)
    if vote_tx.version != 3:                         # TxVersionTreasury
        reject("ErrSSGenInvalidTxVersion")
    return votes
```

dcrd: blockchain/stake/staketx.go · CheckSSGenVotes
blockchain/stake/staketx.go · GetSSGenTreasuryVotes
blockchain/stake/staketx.go · IsNullDataScript

RULE 12.28 Treasury vote output

If the last output of a vote transaction is a null-data script (as defined in Algorithm 12.6), the vote **MUST** pass every check of Algorithm 12.6: the output **MUST** be a well-formed treasury vote output listing at least one pair, every vote byte **MUST** be 0x01 or 0x02, no treasury spend hash **MAY** appear twice, and the vote transaction's version **MUST** be 3. A transaction that fails these checks is not a vote at all; it is then classified as regular, and a block containing it in its stake tree is invalid because the stake tree may not contain regular transactions (Chapter 10).

dcrd: blockchain/stake/staketx.go · CheckSSGenVotes
blockchain/stake/staketx.go · GetSSGenTreasuryVotes

The number of pairs is limited only by the push encoding: at most 2 with `OP_DATA_N` and at most 7 with `OP_PUSHDATA1`, since its length byte cannot exceed 255 and pushes larger than `OP_PUSHDATA1` are rejected. No rule requires a hash to identify an existing treasury spend: a vote may name any 32 bytes. Pairs whose hash matches no includable spend are simply never counted. A vote byte of 0x00 is invalid; to abstain, a voter omits the spend.

IMPLEMENTATION PITFALL

The treasury vote output counts toward the vote's limit of `MaxOutputsPerSSGen` = 66 outputs (Chapter 8), which dcrd's source notes "should technically be increased by 1" but cannot be without a consensus vote. When the agenda is active the output is excluded from the count of payment outputs that must match the ticket's commitments. When the agenda is not active it is not excluded, so a vote carrying a treasury vote output then fails the payee checks of Chapter 8. Treasury vote outputs are therefore effectively invalid before activation.

TEST VECTOR Treasury vote outputs

One yes vote for the treasury spend of the previous vector (`OP_RETURN OP_DATA_35 'T' 'V' <hash> 01`): `6a235456eaa6b2b0e76496f846d58c6b67b70a139cf70e105cd5192a378416b141e9fa8d01`

Note the internal byte order of the hash; its display form is `8dfae941b11684372a19d55c100ef79c130ab7676b8cd546f89664e7b0b2a6ea`. Three pairs (104-byte script, `OP_PUSHDATA1` with length 101) and seven pairs (236-byte script, length 233) both parse. The same hash listed twice fails with `ErrSSGenDuplicateTreasuryVote`; a vote byte of 0x03 fails with `ErrSSGenInvalidTreasuryVote`.

12.8.3 Tallying and approval

The tally for a treasury spend X considered for inclusion in block B at height H counts every yes and no pair naming X's hash in every vote transaction in the stake trees of B's ancestors at heights start through $H-1$. Votes in B itself are not counted. Each ticket votes at most once, so no further deduplication is needed. Votes are counted by inclusion height, regardless of maturity and regardless of whether the regular tree of the block containing them was approved.

ALGORITHM 12.7 Treasury spend vote tally

```
def tally(B, X) -> tuple[uint32, uint32, uint32, uint32]:
    w = tspend_window(X.expiry)
    if w is None:
        reject("ErrInvalidTSpendWindow")    # or ErrInvalidTreasurySpendExpiry
    start, end = w
    if not inside_tspend_window(B.height, X.expiry):
        reject("ErrInvalidTSpendWindow")
    yes: uint32 = 0
    no: uint32 = 0
    n = B.parent
    while n is not None and n.height >= start:
        for stx in n.stake_transactions:
            if not is_vote(stx):            # staking chapter's vote predicate
                continue
            for (h, v) in treasury_votes(stx):
                if h == txhash(X):
                    if v == 0x01: yes += 1  # overflow would reject the block
                    if v == 0x02: no += 1   # (unreachable in practice)
        n = n.parent
    return start, end, yes, no
```

dcrd: internal/blockchain/treasury.go · tSpendCountVotes

The approval test requires a quorum of all possible votes in the window and a yes count of at least TVRM/TVRD of the votes already cast **plus all votes that could still be cast** before the end of the window. Counting the remaining votes as potential no votes lets a spend with an insurmountable yes majority be approved at an earlier TVI ("short-circuiting"). There is no corresponding early rejection in consensus. A spend that fails simply cannot be included, and once its window has passed it can never be included.

ALGORITHM 12.8 Treasury spend approval

```
def check_tspend_votes(B, X):
    start, end, yes, no = tally(B, X)
    max_votes = uint32(uint32(TPB) * (end - start))          # uint32 product
    quorum = uint64(max_votes) * TVQM // TVQD                # uint64
    cast = uint64(uint32(yes + no))
    if cast < quorum:
        reject("ErrNotEnoughTSpendVotes")                    # quorum not met
    remaining_blocks = uint32(end - uint32(B.height))        # >= 0, see tally
    max_remaining = uint64(uint32(remaining_blocks * uint32(TPB)))
    required = (cast + max_remaining) * TVRM // TVRD          # uint64
    if uint64(yes) < required:
        reject("ErrNotEnoughTSpendVotes")                    # not enough yes votes
```

dcrd: internal/blockchain/treasury.go · checkTSpendHasVotes

In Algorithm 12.8 `remaining_blocks` is end $-H$, which is non-negative because the tally has already checked $H \le$ end. All products fit easily in their types on every network. The widths are stated because dcrd computes them that way.

NOTE
With the main-network values the threshold has a simple form when every ticket votes on the spend: cast + remaining = 17280, so the required number of yes votes is $\lfloor 17280 \cdot \frac{3}{5} \rfloor = 10368$, a constant. The earliest approval with unanimous yes votes from a window starting at height s is at the first TVI H with $5(H - s) \geq 10368$, that is $H = s + 2304$ (8 TVIs).

TEST VECTOR Approval at each TVI of a main-network window
Expiry 7778, the smallest valid on the main network, window [4320, 7776]. Votes per block in heights 4320 to 7775. Each entry gives the TVI block height and the counted yes/no votes, followed by "pass" or the reason for failure.

```
5 yes per block
  4320  0/0      quorum            5184  4320/0    yes < 10368
  4608  1440/0   quorum            5760  7200/0    yes < 10368
  4896  2880/0   quorum            6336  10080/0   yes < 10368
                                   6624  11520/0   pass (and every later TVI)
3 yes, 2 no per block
  6624  6912/4608   yes < 10368 (remaining 5760)
  7488  9504/6336   yes < 10368 (remaining 1440)
  7776  10368/6912  pass (exactly 60%)
1 yes, 4 abstaining per block
  7488  3168/0   quorum (3456)    7776  3456/0   pass (quorum exactly; required 2073)
as above but no vote at height 4320
  7776  3455/0   quorum
```

12.8.4 Summary of inclusion conditions

A treasury spend X may be included in block B at height H exactly when all of the following hold, in addition to the transaction rules of Section 12.7.2 and the general rules for stake transactions:

Table 12.11 Conditions for including a treasury spend

Condition	Rule
The `treasury` agenda is active for B and $H \geq$ SVH.	Rule 12.2, Rule 12.3
$H \bmod \text{TVI} = 0$.	Rule 12.20
X.Expiry $\geq 2 + \text{SVH} + \text{TVI} \cdot \text{TVIM}$.	Rule 12.21
X.Expiry is valid and start $\leq H \leq$ end.	Rule 12.27, Rule 12.22
X is not in any ancestor of B.	Rule 12.23
Quorum and yes threshold are met, counting votes in [start, $H - 1$].	Rule 12.24
The total of all spends in B passes the balance and policy checks.	Rule 12.25

There is no consensus limit on the number of treasury spends in a block other than the block size and signature operation limits. (The memory pool holds at most seven, see Section 12.10.5.) A spend that is mined on one chain may be mined again on a competing chain that does not contain it, since Rule 12.23 considers only ancestors.

12.9 Expenditure Policy

Even an approved treasury spend may be included only if the treasury can afford it and the total spent in the recent past stays below a limit. The limit has had three definitions, each selected by an agenda. All three work on **expenditure policy windows** of $W = \text{TVI} \cdot \text{TVIM} \cdot \text{TEW}$ consecutive blocks counted backwards from the parent of the TVI block being validated, and all three sum treasury values by **inclusion** height (the block containing the transaction), not by maturity.

12.9.1 Common procedure

ALGORITHM 12.9 Window sums

```
def window_sums(n, count: int) -> tuple[int64, int64, Block | None]:
    """Sums the treasury values of `count` blocks: n and its ancestors, newest
    first. Returns (spent, added, next older block or None)."""
    spent = added = 0
    for _ in range(count):
        if n is None:
            break
        if not treasury_active(n):       # no treasury values: stop for good
            n = None
            break
        for (kind, amount) in treasury_values(n):
            if is_debit(kind):
                spent += -amount         # treasury spend payouts and fees
            else:
                added += amount          # treasurybases and treasury adds
        n = n.parent
    return spent, added, n
```

dcrd: internal/blockchain/treasury.go · sumPastTreasuryChanges

For each treasury spend, `spent` includes exactly its `ValueIn`. Since the agenda never becomes inactive again, stopping at the first block without treasury values is the same as letting all earlier blocks contribute nothing.

ALGORITHM 12.10 Expenditure check

```
def check_expenditure(B, total: int64):
    # B: a block at a TVI height for which the treasury agenda is active.
    # total: sum of ValueIn of all treasury spends in B (> 0).
    bal = treasury_balance(B)                  # balance as of B (rule above)
    if bal - total < 0:
        reject("ErrInvalidExpenditure")        # treasury may not go negative
    P = B.parent
    if agenda_active("maxtreasuryspend", B):          # DCP-0013
        allowed = max_spend_dcp0013(P)
    elif agenda_active("reverttreasurypolicy", B):    # DCP-0007
        allowed = max_spend_dcp0007(P)
    else:                                             # DCP-0006
        allowed = max_spend_dcp0006(P)
    if total > allowed:
        reject("ErrInvalidExpenditure")
```

dcrd: internal/blockchain/treasury.go · checkTSpendsExpenditure
internal/blockchain/treasury.go · maxTreasuryExpenditure

The policy is selected by the agendas active for B (asked of P in dcrd's terms), with `maxtreasuryspend` taking precedence over `reverttreasurypolicy` regardless of whether the latter is active. In every policy the recent window covers heights $H - W$ through $H - 1$ for a TVI block at height H.

12.9.2 The original policy (DCP-0006)

The original policy limits the spends in the recent window to 150% of the average spending of the TEP preceding windows, ignoring windows without any spending. When none of them has spending, the bootstrap value TEB is used as the average.

ALGORITHM 12.11 Maximum expenditure, DCP-0006

```
def max_spend_dcp0006(P) -> int64:
    W = TVI * TVIM * TEW
    spent_recent, _, n = window_sums(P, W)          # heights H-W .. H-1
    prior_total = 0
    nonempty = 0
    # iteration i = 0..TEP-1 covers heights H-(i+2)W .. H-(i+1)W-1
    for _ in range(TEP):
        if n is None:
            break
        spent, _, n = window_sums(n, W)
        if spent > 0:
            prior_total += spent
            nonempty += 1
    if nonempty > 0:
        avg = prior_total // nonempty
    else:
        avg = TEB
    allowance = avg + avg // 2
    return allowance - spent_recent if allowance > spent_recent else 0
```

dcrd: internal/blockchain/treasury.go · maxTreasuryExpenditureDCP0006

HISTORY
The DCP-0006 policy bases the limit on past spending rather than on income, unlike the policy in the original Politeia proposal for the decentralized treasury, and according to DCP-0007 it prevented payments from the treasury from being made. The `reverttreasurypolicy` agenda replaced it. On the main network it governs TVI blocks from 552448 up to but not including 657280.

12.9.3 The reverted policy (DCP-0007)

The `reverttreasurypolicy` agenda (DCP-0007 [41]) limits spends in the recent window to 150% of the treasury's income (treasurybases and treasury adds) in the same window.

ALGORITHM 12.12 Maximum expenditure, DCP-0007

```
def max_spend_dcp0007(P) -> int64:
    W = TVI * TVIM * TEW
    spent, added, _ = window_sums(P, W)             # heights H-W .. H-1
    allowance = added + added // 2
    return allowance - spent if allowance > spent else 0
```

dcrd: internal/blockchain/treasury.go · maxTreasuryExpenditureDCP0007

12.9.4 The balance-based policy (DCP-0013)

The `maxtreasuryspend` agenda (DCP-0013 [42]) limits the spends in a policy window to 4% of the balance the treasury would have had without those spends, but never less than a fixed floor and never more than the balance itself. The floor is the treasury subsidy of one voting window at the initial, unreduced subsidy rate. It adds no network parameter: the floor is computed from `BaseSubsidy`, TVI and TVIM.

$$F = \left\lfloor \frac{\text{BaseSubsidy}}{10} \right\rfloor \cdot \text{TVI} \cdot \text{TVIM} \tag{12.4}$$

ALGORITHM 12.13 **Maximum expenditure, DCP-0013**

```
FLOOR = (BaseSubsidy // 10) * (TVI * TVIM)      # 1078127767296 on mainnet

def max_spend_dcp0013(P) -> int64:
    W = TVI * TVIM * TEW
    spent, _, _ = window_sums(P, W)                # heights H-W .. H-1
    bal = treasury_balance(child_of(P))            # balance as of the TVI block
    max_spendable = (bal + spent) * 4 // 100       # int64; operands non-negative
    if max_spendable < FLOOR:
        max_spendable = FLOOR
    allowed = max_spendable - spent if max_spendable > spent else 0
    if allowed > bal:
        allowed = bal
    return allowed
```

dcrd: internal/blockchain/treasury.go · maxTreasuryExpenditureDCP0013
internal/blockchain/chain.go · New

`child_of(P)` is the TVI block being validated, so `bal` is the same balance used by the non-negativity check of Algorithm 12.10. Adding `spent` back approximates the balance before the window's spends. It is exact only if those spends have already matured, which on the main network holds for spends in all earlier TVI blocks. The division by 10 in the floor is part of the definition. It is not tied to the treasury proportion of the subsidy.

HISTORY
Under DCP-0007 the limit shrinks with the block subsidy and would eventually leave most of the accumulated balance unspendable. DCP-0013 targets a maximum loss of about 20% of the treasury in the event of a key compromise (five policy windows of 4% each during the roughly four months a consensus change takes) while the floor keeps the limit stable over the emission schedule. On the main network the DCP-0013 policy governs TVI blocks from 1052416 on.

12.9.5 Test vectors

The three policy functions were evaluated by dcrd's own code on synthetic main-network treasury states: every block from height 2 is given a treasurybase of 10^8 atoms (the agenda is never active for block 1, Section 12.1.3). The round amount stands in for the subsidy schedule; the vectors exercise the policy arithmetic only. The block being validated is the TVI block at height 48384 (the parent is at 48383). The windows are therefore: recent $[41472, 48383]$, and prior windows $[34560, 41471]$, $[27648, 34559]$, $[20736, 27647]$, $[13824, 20735]$, $[6912, 13823]$ and $[0, 6911]$. Each treasury spend below has a fee of 10000 atoms included in the stated `ValueIn`.

TEST VECTOR Expenditure limits for the TVI block at height 48384

```
S0: no treasury spends or adds.
    balance 4812700000000 (48127 tbases), recent spent 0, added 691200000000
    DCP-0006  2400000000000   (bootstrap: 16000 DCR * 1.5)
    DCP-0007  1036800000000   (6912 DCR * 1.5)
    DCP-0013  1078127767296   (4% of balance below floor; floor < balance)

S1: spends with ValueIn 12000 DCR at 20160, 9000 DCR at 34560, 5000 DCR at 43200;
    treasury add 500 DCR at 45000.
    balance 2262700000000, recent spent 500000000000, added 741200000000
    DCP-0006  1075000000000   (avg of 2 non-empty windows 10500 DCR; 15750 - 5000)
    DCP-0007  611800000000    (7412 * 1.5 = 11118 DCR; minus 5000)
    DCP-0013  578127767296    (floor - 5000 DCR)

S2: as S1 plus a treasury add of 1000000 DCR at height 1000.
    balance 102262700000000, recent spent 500000000000
    DCP-0006  1075000000000,  DCP-0007  611800000000 (unchanged)
    DCP-0013  3610508000000   ((1022627 + 5000) * 4% = 41105.08 DCR; minus 5000)
```

A block at height 48384 whose treasury spends total exactly the stated limit passes the policy check, and one atom more fails it.

12.10 Interactions with Other Rules

12.10.1 Signature operations

The signature operation limits of Chapter 10 apply to treasury transactions with the following treasury-specific counting, specified in full in Chapter 6:

RULE 12.29 Signature operations of treasury transactions
When the agenda is active: a treasurybase counts zero signature operations; each `OP_TSPEND` in a script counts one signature operation (so the input of a treasury spend counts one); outputs are counted as for any transaction (an `OP_TGEN`-tagged P2PKH output counts one for its `OP_CHECKSIG`; an `OP_TGEN`-tagged P2SH output counts zero); and treasury spends and treasurybases contribute no pay-to-script-hash signature operations. A treasury add is counted like any other transaction.

dcrd: internal/blockchain/validate.go · countSigOps
internal/blockchain/validate.go · countP2SHSigOps
txscript/script.go · countSigOpsV0

Treasury transactions count toward the block size like any other transaction. There is no separate size limit.

12.10.2 Regular transactions and treasury opcodes

RULE 12.30 No treasury opcodes in regular outputs
When the agenda is active for a block, no output script of a transaction classified as regular **MAY** contain any of the opcodes 0xc1, 0xc2 or 0xc3, in addition to the stake opcodes 0xba to 0xbd that are always forbidden there (Chapter 7). Scripts are parsed as in Chapter 6. A parse failure rejects the block.

dcrd: internal/blockchain/validate.go · checkTransactionContext
txscript/script.go · ContainsStakeOpCodes

When the agenda is active, scripts are executed with the treasury script flag, under which 0xc1 to 0xc3 are the treasury opcodes rather than unknown opcodes reserved for upgrades. Their execution semantics, and the handling of `OP_TGEN`-tagged scripts when their outputs are spent, belong to Chapter 6.

12.10.3 Maturity of vote and revocation outputs

When the `treasury` agenda is active for the spending block, an output whose script is tagged with `OP_SSGEN` or `OP_SSRTX` matures after `CoinbaseMaturity` blocks instead of `SStxChangeMaturity`. The rule is Rule 8.22.

NOTE
This change is not mentioned in DCP-0006 but is part of the code that implements it. On the main network it raised the maturity of vote and revocation outputs from 1 to 256 blocks.

12.10.4 Other interactions

- **Disapproval.** A treasurybase, treasury add or treasury spend is never undone by disapproval of its block's regular tree. Treasury adds may spend outputs of the parent's regular tree like any stake transaction; if the parent is disapproved, those outputs do not exist and the treasury add is invalid (Chapter 10).
- **Fees.** The fees of treasury adds and treasury spends are stake-tree fees available to the coinbase. The treasurybase has no fee.
- **Header commitments.** The version 2 compact filters committed to by the block header include the payout scripts of treasury spends, the previous output scripts spent by treasury adds and the change scripts of treasury adds (Chapter 16).

12.10.5 Memory pool and mining policy (non-normative)

The following is dcrd policy, not consensus:

- A treasurybase is never accepted into the memory pool.
- A treasury spend is accepted only if the treasury agenda is active, the next block is at or above stake validation height, its expiry is valid, its voting window does not start $2 \cdot \mathrm{TVI} \cdot \mathrm{TVIM}$ or more blocks after the next block, its Pi key is known and its signature valid, and it has not been mined in an ancestor of the current tip. The memory pool holds at most seven treasury spends at a time (`MempoolMaxConcurrentTSpends`), matching the maximum number of pairs in one treasury vote output.
- Treasury adds and treasury spends must pay at least the minimum relay fee.
- Voting wallets append treasury votes for the spends in their memory pool according to their configured policy.
- The block template generator includes, in TVI blocks, those treasury spends that pass the vote check, subject to the maximum expenditure for the block. It creates the treasurybase with random bytes after the height.

12.11 Error Kinds

Table 12.12 maps the error kinds that dcrd reports for violations of the rules in this chapter to the rules. It is informative: only validity matters for consensus, not which error is reported.

Table 12.12 dcrd error kinds and the rules they correspond to

Error kind	Rule
`ErrStakeTxInRegularTree`, `ErrNonstandardStakeTx`, `ErrInvalidEarlyStakeTx`	Rule 12.1, Rule 12.2, Rule 12.3
`ErrFirstTxNotTreasurybase`, `ErrMultipleTreasurybases`	Rule 12.6, Rule 12.8
`ErrTreasuryBaseInvalid*` (stake), `ErrBadTreasurybaseOutpoint`, `ErrBadTreasurybaseFraudProof`, `ErrBadTreasurybaseScriptLen`	Rule 12.7
`ErrTreasurybaseTxNotOpReturn`, `ErrTreasurybaseHeight`	Rule 12.8
`ErrInvalidTreasurybaseTxOutputs`, `ErrInvalidTreasurybaseVersion`, `ErrInvalidTreasurybaseScript`, `ErrTreasurybaseOutValue`, `ErrBadTreasurybaseAmountIn`, `ErrSpendTooHigh`	Rule 12.9
`ErrTAddInvalid*` (stake)	Algorithm 12.3 (classification)
`ErrInvalidTAddChange`	Rule 12.11
`ErrTooManyTAdds`	Rule 12.13
`ErrBadTxInput` (spending an `OP_TADD` output)	Rule 12.14
`ErrTSpendInvalid*` (stake)	Algorithm 12.4 (classification)
`ErrBadTSpendOutpoint`, `ErrBadTSpendFraudProof`, `ErrBadTSpendScriptLen`	Rule 12.15
`ErrUnknownPiKey`	Rule 12.16
`ErrInvalidPiSignature`	Rule 12.17
`ErrBadTxInput`, `ErrInvalidTSpendValueIn`, `ErrSpendTooHigh`	Rule 12.18
`ErrMissingTxOut` (sequence lock)	Rule 12.19
`ErrNotTVI`	Rule 12.20
`ErrInvalidTVoteWindow`	Rule 12.21
`ErrInvalidTSpendWindow`, `ErrInvalidTreasurySpendExpiry`	Rule 12.27, Rule 12.22
`ErrTSpendExists`	Rule 12.23
`ErrNotEnoughTSpendVotes`, `ErrTooManyTreasurySpendVotes` (surfaced by block validation as `ErrNotEnoughTSpendVotes`)	Rule 12.24
`ErrInvalidExpenditure`, `ErrBadTxOutValue` (sum overflow)	Rule 12.25
`ErrImmatureSpend`, `ErrExpiryTxSpentEarly`	Rule 12.26, Rule 8.22

Error kind	Rule
`ErrSSGenInvalidDiscriminatorLength`, `ErrSSGenInvalidNullScript`, `ErrSSGenUnknownDiscriminator`, `ErrSSGenInvalidTVLength`, `ErrSSGenInvalidTreasuryVote`, `ErrSSGenDuplicateTreasuryVote`, `ErrSSGenInvalidTxVersion`	Rule 12.28
`ErrRegTxCreateStakeOut`, `ErrScriptMalformed`	Rule 12.30
`ErrTooManySigOps`	Rule 12.29

CHAPTER 13

Consensus Rule Changes

Decred changes its consensus rules by an on-chain vote of the stakeholders. New rules are shipped in node software in a dormant state, together with a description of the vote that can enable them. Proof-of-work miners signal that they run the new software through the block version, and voters signal it through the version they attach to their votes. Once enough of both have upgraded, the stakeholders vote on the change for a fixed interval of blocks; a qualified majority locks the outcome in, and the new rules take effect one interval later. Every step is a deterministic function of the block chain, so every node reaches the same verdict on every block.

Other chapters state rules of the form "if agenda *X* is active for the block". This chapter defines exactly what that means, specifies the version rules that the upgrade process imposes on block headers, lists every agenda ever defined on every network, and records the outcome of each vote on the main network.

13.1 Overview

13.1.1 Terminology

Table 13.1 defines the vocabulary used throughout the chapter.

Table 13.1 Terms used in this chapter

Term	Meaning
Agenda	A proposed consensus change, identified by a short ASCII *agenda ID* such as `blake3pow`. The rules of other chapters refer to agendas by ID.
Deployment	The parameters under which an agenda is voted on: the vote definition (mask and choices), a start time, an expire time and, optionally, a forced choice. Deployments are part of the network parameters (`Deployments`).
Deployment version	The key under which a deployment is listed in `Deployments`. Several deployments can share a version and are then voted on in parallel. Only votes whose vote version equals the deployment version count toward it.
Vote version	A 32-bit number carried in every vote (Section 13.4). Voting software sets it to the highest deployment version it knows.
Vote bits	The 16-bit field of a vote that carries the approval of the previous block in bit 0 and the voter's choices on agendas in the other bits.
Block version	The `Version` field of the block header (signed 32-bit). Miners raise it to signal that they run upgraded software.
Stake version	The `StakeVersion` field of the block header (unsigned 32-bit). It records the majority vote version of the stakeholders and, once enforced, is fully determined by the chain (Section 13.5).
Stake version interval (SVI)	A run of `StakeVersionInterval` blocks over which vote versions are tallied.
Rule change interval (RCI)	A run of `RuleChangeActivationInterval` blocks over which the votes on an agenda are tallied. The threshold state of a deployment can change only at the boundary between two such intervals.
Threshold state	One of `Defined`, `Started`, `LockedIn`, `Active` and `Failed`: the stage that a deployment has reached (Section 13.6).

Term	Meaning
Active for block B	The predicate consulted by agenda-dependent rules when validating block B (Rule 13.6).

13.1.2 The upgrade cycle

The following sequence is informative; the normative rules are in the rest of the chapter. A summary for users is in the Decred documentation [37], and the governance process that precedes a vote (an off-chain proposal and a Decred Change Proposal, or DCP) is described in DCP-0000 [7].

1. A DCP specifies the change and a deployment: a new deployment version v, an agenda ID, a mask of vote bits, the choices, and start and expire times. New node software containing the dormant rules and the deployment is released. The software mines blocks of version v (on the main network) and casts votes of version v.
2. **Proof-of-work upgrade.** The upgrade of the miners is complete for the purposes of a deployment of version v once `BlockRejectNumRequired` (950) of the last `BlockUpgradeNumToCheck` (1000) blocks have a block version of at least v. The same count, applied to version $v + 1$, makes blocks of version v and below invalid (Rule 13.1).
3. **Proof-of-stake upgrade.** The upgrade of the voters is complete once 75 % of the votes in a stake version interval carry vote version v. From the following interval on, the header stake version of every block must be v (Rule 13.4).
4. **Voting.** At the first rule change interval boundary at which both upgrades are complete and the start time has passed, the deployment moves to `Started`. During each interval in the `Started` state the votes of version v are tallied. If at least `RuleChangeActivationQuorum` (4032) votes are not abstentions and one choice receives at least 75 % of them, the vote is decided: a winning "yes" choice moves the deployment to `LockedIn`, a winning "no" choice to `Failed`. Otherwise voting continues in the next interval until the expire time passes.
5. **Activation.** A deployment that is `LockedIn` for one interval becomes `Active` at the next boundary, and its rules apply to every block from then on. The lock-in interval gives everyone the time to upgrade.

On the main network a rule change interval is 8064 blocks (four weeks), so a successful vote takes at least eight weeks from the start of voting to activation.

> **RATIONALE**
> The design descends from Bitcoin's BIP 9 version bits [43], which also defines the five threshold states. It differs in who decides: in Decred the proof-of-work miners only signal readiness, and the decision is taken by the stakeholders through their votes. Abstentions are excluded from both the majority calculation and the quorum, so that an uninformed or indifferent stakeholder does not bias the outcome in either direction.

13.1.3 Parameters

Table 13.2 lists the network parameters used by the algorithms of this chapter. The complete parameter sets are in Chapter 14.

Table 13.2 Voting-related network parameters. The last row is a constant of the validation code, not a field of the parameters (Section 13.3).

Parameter	mainnet	testnet3	simnet	regnet
`RuleChangeActivationInterval` (RCI)	8064	5040	320	320
`RuleChangeActivationQuorum`	4032	2520	160	160
`RuleChangeActivationMultiplier`	3	3	3	3
`RuleChangeActivationDivisor`	4	4	4	4
`StakeVersionInterval` (SVI)	2016	2016	112	112
`StakeValidationHeight` (SVH)	4096	768	144	144
`StakeEnabledHeight`	512	32	32	32

Parameter	mainnet	testnet3	simnet	regnet
StakeMajorityMultiplier	3	3	3	3
StakeMajorityDivisor	4	4	4	4
BlockEnforceNumRequired	750	51	51	51
BlockRejectNumRequired	950	75	75	75
BlockUpgradeNumToCheck	1000	100	100	100
TicketsPerBlock	5	5	5	5
Latest enforced block version	11	12	12	12

StakeEnabledHeight is listed because it bounds the earliest votes, but none of the algorithms below consults it directly: they key on StakeValidationHeight, the first height at which blocks carry votes (Chapter 8). On the main network the quorum is 10 % of the maximum number of votes in a rule change interval ($8064 \times 5 = 40320$).

13.2 Interval Arithmetic

Both kinds of interval are aligned to the stake validation height rather than to height 0. An interval of length I consists of the heights $\mathrm{SVH} + kI$ through $\mathrm{SVH} + (k+1)I - 1$ for an integer k; on the main network the rule change intervals are $[4096, 12159]$, $[12160, 20223]$, and so on. Every rule in this chapter that consults "the previous interval" uses the function of Algorithm 13.1, which returns the height of the last block of the interval preceding the one that contains height h.

ALGORITHM 13.1 Final block of the previous interval

```
def want_height(svh: int64, interval: int64, h: int64) -> int64:
    # Height of the final block of the interval that precedes the interval
    # containing height h.  Intervals start at heights svh + k*interval.
    offset = tmod(svh, interval)
    adjusted = h - offset - 1
    return adjusted - tmod(adjusted + 1, interval) + offset
```

dcrd: internal/blockchain/stakeversion.go · calcWantHeight

For $h \geq \mathrm{SVH}$ this simplifies to $h - 1 - ((h - \mathrm{SVH}) \bmod I)$. All consensus uses of the function satisfy $h \geq \mathrm{SVH}$. The uses that locate the previous interval of a block (the threshold state and the stake version interval preceding a block) have $h \geq \mathrm{SVH} + I$, so their result is at least $\mathrm{SVH} + I - 1$ and refers to a block whose whole interval lies at or above the stake validation height; the one use with a smaller argument, finding the first block of a stake version interval in Algorithm 13.3, has $h \geq \mathrm{SVH} + I - 1$ and a result of at least $\mathrm{SVH} - 1$. If h is the first height of an interval, the result is $h - 1$.

TEST VECTOR Previous-interval heights with main-network parameters

```
RCI = 8064, SVH = 4096                SVI = 2016, SVH = 4096
want_height(..., 4096)    =   4095    want_height(..., 6111) = 4095
want_height(..., 12159)   =   4095    want_height(..., 6112) = 6111
want_height(..., 12160)   =  12159    want_height(..., 6113) = 6111
want_height(..., 12161)   =  12159    want_height(..., 8127) = 6111
want_height(..., 20224)   =  20223    want_height(..., 8128) = 8127
want_height(..., 149247)  = 141183
want_height(..., 149248)  = 149247
want_height(..., 1052416) = 1052415
```

13.3 Block Versions

The block version is the miners' signal. It plays three roles: old block versions become invalid once nearly all miners have upgraded, a deployment cannot start until nearly all miners mine its version, and the stake version rules are enforced only once the miners have moved to version 3. All three use the same count, Algorithm 13.2, over the most recent blocks.

ALGORITHM 13.2 Proof-of-work version majority

```
def pow_majority(min_version: int32, start: Node | None, required: uint64) -> bool:
    # Counts, among BlockUpgradeNumToCheck blocks ending with `start` and going
    # back through its ancestors (fewer near genesis), those whose header
    # version is >= min_version.  Versions compare as signed 32-bit integers.
    found = 0
    node = start
    i = 0
    while i < BLOCK_UPGRADE_NUM_TO_CHECK and found < required and node is not None:
        if node.header.version >= min_version:
            found += 1
        node = node.parent
        i += 1
    return found >= required
```

dcrd: internal/blockchain/chain.go · isMajorityVersion

The early exit once `found` reaches `required` does not change the result. A block's own version is never part of the count used to judge it: every caller passes the parent of the block being validated, or an older block.

RULE 13.1 Obsolete block versions

Let B be a block other than the genesis block, P its parent, and V its header version (`int32`). Let $L = 11$ on the main network and $L = 12$ on every other network. If $V < L$, B is not one of the exempt blocks of Table 13.3, and `pow_majority(V + 1, P, BlockRejectNumRequired)` holds, then B **MUST** be rejected (`ErrBlockVersionTooOld`). Blocks with $V \geq L$ are never rejected by this rule.

dcrd: internal/blockchain/validate.go · checkBlockHeaderPositional, isOldBlockVersionByMajority, isDCP0005Violation

On the main network the rule rejects a block of version $V < 11$ once 950 of the 1000 blocks ending with its parent have version $V + 1$ or higher. Since the count is monotone in the version, this single comparison is equivalent to rejecting V when a 95 % majority exists for *any* version above it.

Table 13.3 Blocks exempt from Rule 13.1. They were mined with the old version after the version-7 (main network) or version-8 (testnet3) majority had formed, and were accepted because of a bug in software of the time.

Network	Height	Condition
mainnet	413762	version 6 and block hash `000000000000000002086ed61f62a546f3bfa8f3e567b003f097efa982008c47b`
mainnet	414036	version 6 and block hash `000000000000000194fa59310b9e988ecb23de0c716d6e8f1b2aa31d9592387`
mainnet	424011	version 6 and block hash `000000000000000317fc6c7a8a6578be7dfa9c96eb81d620050a3732b02d572`
mainnet	428809	version 6 and block hash `0000000000000000003147798ccffcecaa420fb1c7934d8f4e33809a871ee34aaa`
mainnet	430191	version 6 and block hash `000000000000000002127ad6d4cb30cc16f6344589b417e42650388bb0690a88e`
testnet3	$\leq$ 323282	every block of version 7 whose header height is at most 323282

Hashes are in display form. The main-network condition compares the header height field, the block hash and the version; the testnet3 condition compares only the header height field and the version.

HISTORY
The rule was originally a series of explicit checks, one per version ("reject version 6 once 95 % are version 7", and so on); dcrd replaced them with a loop over the versions in 2020 and with the single comparison above in 2022. The block version that the reference miner produces has been raised with each set of deployments: version 3 came with the stake version mechanism (dcrd source, December 2016), and versions 4 through 11 came with deployment versions 4 through 11 of the main network. On the test networks the miner produces version 12, matching their deployment versions, which are one higher than the main network's. The heights at which each main-network version reached its majorities are not recorded in the sources used for this document.

The block version also gates deployments (Section 13.6): a deployment of version v cannot move from `Defined` to `Started` until `pow_majority(v, W, BlockRejectNumRequired)` holds at the end W of a rule change interval. The block version number is compared directly with the deployment version number, which is why the reference software mines blocks whose version equals its newest deployment version.

IMPLEMENTATION PITFALL
The rule is checked against the parent's ancestry, not against a fixed height, so a side chain is judged on its own blocks. dcrd applies it to every header when the header is first accepted, including ancestors of its configured assumed-valid block: the assume-valid optimization skips only the contextual checks listed in Chapter 10.

13.4 Votes, Vote Versions and Vote Bits

13.4.1 The votes of a block

Every block at or above the stake validation height carries votes: vote transactions (also called SSGen) in its stake tree, whose structure is specified in Chapter 8. For this chapter a vote is reduced to a pair (version, bits) taken from its second output, an `OP_RETURN` script of the form `OP_RETURN` ‖ push opcode ‖ data.

RULE 13.2 Vote bits and vote version
Let `s` be the script of output 1 of a vote, indexed from 0. The vote bits are the unsigned 16-bit little-endian integer formed by the bytes `s[2:4]`. The vote version is the unsigned 32-bit little-endian integer formed by the bytes `s[4:8]` if $|s| \geq 8$, and 0 otherwise. The **votes of block B** are the (version, bits) pairs of all vote transactions in B's stake tree. Every vote counts once, in every tally of this chapter that covers B.

dcrd: `blockchain/stake/staketx.go · SSGenVoteBits, SSGenVersion, FindSpentTicketsInBlock`
`internal/blockchain/blockindex.go · populateTicketInfo`

The structural rules of Chapter 8 guarantee that `s[0]` is `OP_RETURN`, that `s[1]` is a single-byte push opcode of 2 to 75 bytes, and that $4 \leq |s| \leq 77$, so the two vote-bits bytes always exist. No consensus rule restricts the vote version: any value is valid, and votes whose version does not match a deployment are simply not counted for it. The vote version occupies the first four bytes of what Chapter 8 calls the extended vote bits; no rule of this chapter interprets the bytes after it.

13.4.2 Layout of the vote bits

Table 13.4 Use of the 16 vote bits

Bits	Use
0	Approval of the regular transaction tree of the parent block (`1` = approve). This bit is not part of any agenda; its consensus effect is specified in Chapter 10 and Chapter 8.
1–15	Agenda choices. Each deployment owns a mask of consecutive bits; the masks of the deployments sharing a deployment version are disjoint. Bits outside every mask of the vote's version carry no meaning.

Because masks are only required to be disjoint within one deployment version, different versions reuse the same bits: bits 1 and 2 (mask `0x0006`) carry `sdiffalgorithm` for version 4, `lnfeatures` for version 5, and so on. The vote version selects which deployments a vote addresses.

13.4.3 Choices

A deployment's vote has an ordered list of choices. Each has an ID, a bit pattern within the mask, and two flags: `IsAbstain` marks the abstain choice and `IsNo` marks the choice that rejects the change. A choice with neither flag is a passing choice. Every deployment defined on any network has exactly three choices: index 0 `abstain` (bits `0x0000`, `IsAbstain`), index 1 `no` (`IsNo`) and index 2 `yes`. A deployment could define several passing choices, and the tally supports that, but none has.

The tally does not look up a vote's bit pattern in the choice list. It decodes the vote bits to a choice *index*,

```
idx = (bits & mask) >> trailing_zeros16(mask)
```

where `trailing_zeros16` is the number of trailing zero bits of the 16-bit mask. A vote whose index is not smaller than the number of choices is *invalid* for that deployment and is ignored entirely; it counts neither as a choice nor toward the quorum. For this to agree with the bit patterns, the choice at index `i` must have bits `i << trailing_zeros16(mask)`; every deployment defined on every network satisfies this, as checked by the program that generated the tables of Section 13.9.

RULE 13.3 Well-formed deployments

The deployments of a network **MUST** satisfy all of the following; dcrd refuses to start with parameters that violate any of them.

1. Agenda IDs are unique across all deployment versions.
2. Within one deployment version, the masks are pairwise disjoint.
3. A mask is non-zero, does not include bit 0, and consists of one run of consecutive set bits.
4. The number of choices is at most 2^p, where p is the number of bits in the mask.
5. Every choice has a non-empty ID; choice IDs are unique within the vote, compared case-insensitively.
6. Every choice except the abstain choice has non-zero bits; every choice's bits are a subset of the mask.
7. No choice has both `IsAbstain` and `IsNo`; exactly one choice has `IsAbstain` and exactly one has `IsNo`.
8. A forced choice, if given, names an existing choice that is not the abstain choice. Forced choices are not permitted on the main network.
9. A historical activation (Table 13.8) names an agenda that has a deployment without a forced choice, and a winning choice that exists and is not the abstain choice. (An entry whose winning choice is the `no` choice would be ignored; every entry in Table 13.8 names `yes`.)

dcrd: `internal/blockchain/agendas.go` · `makeAgendas`, `validateDeploymentChoices`, `determineForcedThresholdState`
`internal/blockchain/agendas.go` · `makeHistoricalAgendas`

These conditions do not require the abstain choice to have bits `0x0000` or to come first, nor do they require the index property above; an implementation that adds deployments of its own must ensure the latter, or reproduce the index-based decoding exactly.

TEST VECTOR Decoding vote bits for the four main-network version-9 deployments

vote bits	revertttreasury-policy mask 0x0006	explicitver-upgrades mask 0x0018	autorevocations mask 0x0060	changesubsidy-split mask 0x0180
0x0001	0 abstain	0 abstain	0 abstain	0 abstain
0x0015	2 yes	2 yes	0 abstain	0 abstain
0x0155	2 yes	2 yes	2 yes	2 yes

```
0x00ab      1 no            1 no          1 no            1 no
0x0007      3 invalid       0 abstain     0 abstain       0 abstain
```

Each entry is the decoded index and the choice; an invalid index is ignored by the tally. Bit 0 (parent approval) is set in all five examples and plays no part in the decoding.

13.5 Stake Version

The header stake version records which vote version a qualified majority of the voters uses. Because the votes of earlier blocks are on the chain, the value is computable by every node, and once a 75 % majority of miners mine version 3 blocks the header field **MUST** equal the computed value. The threshold state machine then reads the computed value to decide whether the voters have upgraded for a deployment.

The computation for a block B with parent P proceeds in three steps.

1. Find the most recent stake version interval that ends before the interval containing B and in which one vote version accounts for at least 75 % of all votes. If no interval qualifies, the result is 0.
2. Return 0 unless 950 of the 1000 blocks ending with the *first* block of that interval have block version 3 or higher (main-network values).
3. Prevent regression: if at least 75 % of the headers in that interval already carry a stake version of at least the voter majority version, and a header stake version with a 75 % majority among those headers is higher still, return that higher header version instead.

ALGORITHM 13.3 **Required header stake version**

```python
def prior_svi_end(prev: Node) -> Node | None:
    # Final block of the stake version interval preceding the one that
    # contains the block after `prev`.
    h = prev.height + 1
    if h < SVH + SVI:
        return None
    return prev.ancestor(want_height(SVH, SVI, h))

def svi_blocks(end: Node) -> list[Node]:
    # The SVI blocks ending with `end` (all at or above SVH when used below).
    return [end.ancestor(end.height - i) for i in range(SVI)]

def voter_majority(end: Node) -> uint32 | None:
    total = 0                                   # int32 in dcrd; cannot overflow
    count = {}                                  # vote version -> number of votes
    for node in svi_blocks(end):
        total += len(node.votes)
        for vote in node.votes:
            count[vote.version] = count.get(vote.version, 0) + 1
    required = total * STAKE_MAJORITY_MULTIPLIER // STAKE_MAJORITY_DIVISOR
    for version, n in count.items():            # at most one can qualify
        if n >= required:
            return version
    return None

def voter_version(prev: Node) -> tuple[uint32, Node | None]:
    node = prior_svi_end(prev)
    while node is not None and node.height >= SVH:
        v = voter_majority(node)
        if v is not None:
            return (v, node)
        node = node.ancestor(node.height - SVI)  # one interval further back
    return (0, None)
```

```
def headers_at_least(min_version: uint32, end: Node) -> bool:
    # `end` is the final block of a stake version interval.
    n = sum(1 for node in svi_blocks(end) if node.header.stake_version >= min_version)
    return n >= SVI * STAKE_MAJORITY_MULTIPLIER // STAKE_MAJORITY_DIVISOR

def header_majority(end: Node) -> uint32:
    count = {}
    for node in svi_blocks(end):
        v = node.header.stake_version
        count[v] = count.get(v, 0) + 1
    required = SVI * STAKE_MAJORITY_MULTIPLIER // STAKE_MAJORITY_DIVISOR
    for version, n in count.items():
        if n >= required:
            return version
    return 0                                        # no header majority

def required_stake_version(prev: Node) -> uint32:
    # The stake version required in the header of the block after `prev`.
    version, end = voter_version(prev)
    if version == 0 or end is None:
        return 0
    first = end.ancestor(want_height(SVH, SVI, end.height) + 1)
    if not pow_majority(3, first, BLOCK_REJECT_NUM_REQUIRED):
        return 0
    if headers_at_least(version, end):
        prior = header_majority(end)
        if prior > version:
            version = prior
    return version
```

dcrd: internal/blockchain/stakeversion.go · findStakeVersionPriorNode, calcVoterVersionInterval, calcVoterVersion
internal/blockchain/stakeversion.go · isStakeMajorityVersion, calcPriorStakeVersion, calcStakeVersion

`first` is the first block of the interval ending at `end`, since `want_height(SVH, SVI, end.height)` is the final block of the interval before it. The interval end found by `voter_version` is always at height $\mathrm{SVH} + k \cdot \mathrm{SVI} - 1$ for some $k \geq 1$, so every window examined lies entirely at or above the stake validation height. On the main network a voter majority needs 75 % of at least 3×2016 votes (a block at or above the stake validation height carries at least three votes), and a header majority needs $\lfloor 2016 \times 3/4 \rfloor = 1512$ headers.

RULE 13.4 Header stake version

Let B be a block other than the genesis block and P its parent. If B's header version is at least 3 and `pow_majority(3, P, BlockEnforceNumRequired)` holds, the header stake version of B **MUST** equal `required_stake_version(P)` (`ErrBadStakeVersion`). Otherwise the header stake version is unconstrained.

dcrd: internal/blockchain/validate.go · checkBlockHeaderContext

On the main network the condition is that 750 of the 1000 blocks ending with the parent have version 3 or higher. Note the two different thresholds: the stake version is *enforced* once 75 % of recent blocks are version 3 (`BlockEnforceNumRequired`), but it is *non-zero* only once 95 % of the blocks ending with the start of the relevant stake version interval are (`BlockRejectNumRequired`).

IMPLEMENTATION PITFALL

`voter_majority` counts *all* votes, whatever their version, in the total, and requires count $\geq \lfloor 3 \text{ total } /4 \rfloor$ with truncating integer division. With at least three votes the qualifying version, if any, is unique, so the iteration order of the tally (a hash map in dcrd) does not matter. The same holds for `header_majority`, whose total is the fixed interval length.

IMPLEMENTATION PITFALL

The stake version never decreases while the header majority holds: once 75 % of an interval's headers carry version v, later intervals whose voter majority is a lower version u still yield v, because `headers_at_least(u, end)` holds and `header_majority(end)` returns $v > u$. dcrd's test `TestCalcStakeVersionCorners` exercises this sequence and its assertions are consistent with it: vote versions 2, 4, 2 (three intervals each), then 5 for two intervals, then 4 for two intervals; the required version at the end of these five phases is 2, 4, 4, 5, 5. A lower voter majority is adopted only if the higher header version itself has lost its 75 % header majority, which cannot happen while miners follow the rule.

TEST VECTOR Stake version on a simulated regression-network chain

```
network     regnet: SVH 144, SVI 112, RCI 320, BlockRejectNumRequired 75 of 100
blocks      every block has block version 12 and the required stake version;
            blocks at heights >= 144 carry 5 votes of version 12
result      required_stake_version(P) is 0 for blocks at heights 1 .. 255
            and 12 from height 256 on (first SVI window 144 .. 255)
```

Produced by running dcrd's `calcStakeVersion` over a chain of block index nodes. The threshold states on the same chain are given in Section 13.6.3.

13.6 The Threshold State Machine

13.6.1 States

Each deployment has, for every block, a threshold state together with a choice ID (Table 13.5). The state applies to whole rule change intervals: all blocks of one interval share it, and it is a function of the blocks up to and including the last block of the *preceding* interval.

Table 13.5 Threshold states

State	Meaning	Choice ID
`Defined`	Initial state. The start time has not passed or the upgrade conditions are not met. The genesis interval and every block below SVH + RCI are in this state.	empty
`Started`	Votes cast in this interval are tallied at its end.	empty
`LockedIn`	A passing choice won. The interval runs without further tallying.	winning choice
`Active`	The change is in force. Terminal.	winning choice
`Failed`	The vote expired or the `no` choice won. Terminal.	the `no` choice if it won; empty if expired

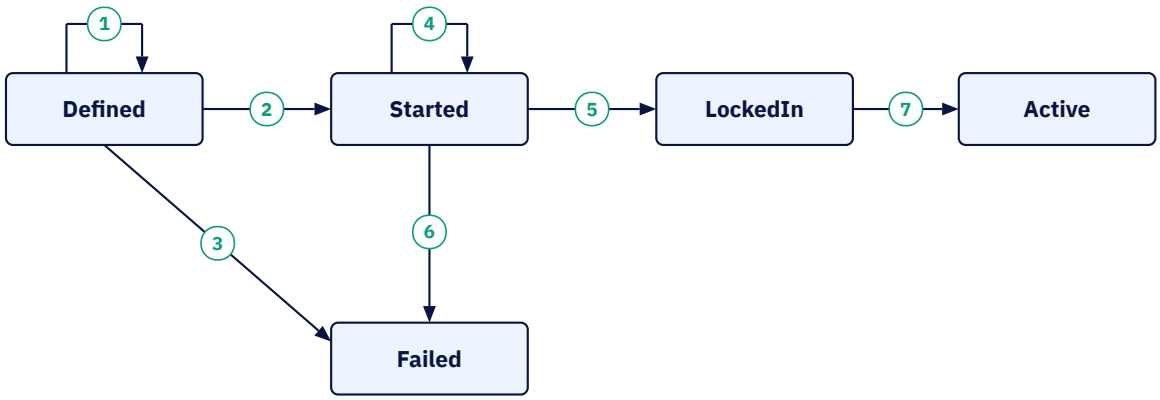


Figure 13.1 Threshold state transitions. Transitions happen only at the boundary between two rule change intervals; the numbers refer to Table 13.6.

13.6.2 When and over which blocks the state is evaluated

Let B be a block other than the genesis block and h its height.

- If $h < \text{SVH} + \text{RCI}$, every deployment is `Defined` for B (forced and historical cases aside, Section 13.7).
- Otherwise let W be the ancestor of B at height `want_height(SVH, RCI, h)`: the last block of the rule change interval before the one containing B. The state for B is the state *computed at* W, written $S(W)$ below.

$S(W)$ is defined recursively from $S(W')$, where W' is the ancestor of W one interval earlier (height $W.h - \text{RCI}$), by the transitions of Table 13.6. The recursion bottoms out when W' does not exist (negative height) or when the median time of W is before the start time. All times are compared as unsigned 64-bit integers: the **median time** of a block, written $\text{MTP}(W)$, is the median of the timestamps of W and its ten predecessors, as defined in Chapter 10, and the start and expire times are the deployment's `StartTime` and `ExpireTime`.

On the main network, W ranges over the heights $4095 + 8064k$. The first block whose state can differ from `Defined` is at height $4096 + 8064 = 12160$, judged at $W = 12159$.

Table 13.6 Transitions at the end W of a rule change interval, for a deployment of version v. Conditions in a row are tested in the order listed; the first row whose condition holds applies.

#	From	Condition at W	To
–	any	$\text{MTP}(W) < \text{StartTime}$ (no recursion)	`Defined`
1	`Defined`	$W.h < \text{SVH}$	`Defined`
3	`Defined`	$\text{MTP}(W) \geq \text{ExpireTime}$	`Failed`
1	`Defined`	`required_stake_version(W)` $< v$	`Defined`
1	`Defined`	not `pow_majority(v, W, BlockRejectNumRequired)`	`Defined`
2	`Defined`	otherwise ($\text{MTP}(W) \geq \text{StartTime}$ holds)	`Started`
6	`Started`	$\text{MTP}(W) \geq \text{ExpireTime}$	`Failed`
4	`Started`	non-abstaining votes in the interval ending at W fewer than `RuleChangeActivationQuorum`	`Started`
5	`Started`	first passing choice (in choice order) with at least the threshold of votes	`LockedIn`
6	`Started`	the `no` choice has at least the threshold of votes	`Failed`
4	`Started`	otherwise	`Started`
7	`LockedIn`	always	`Active`
–	`Active`, `Failed`	always	unchanged

ALGORITHM 13.4 Threshold state of a deployment

```
DEFINED, STARTED, LOCKED_IN, ACTIVE, FAILED = range(1, 6)

def deployment_state(d, B: Node) -> tuple[int, str]:
    # State and choice ID of deployment d for block B (B is not genesis).
    if B.height < SVH + RCI:
        return (DEFINED, "")
    W = B.parent.ancestor(want_height(SVH, RCI, B.height))
    return state_at(d, W)

def state_at(d, W: Node | None) -> tuple[int, str]:
    # State that applies to the RCI blocks following the interval end W.
    if W is None:
        return (DEFINED, "")
    if uint64(mtp(W)) < d.start_time:
        return (DEFINED, "")
    prev_state, prev_choice = state_at(d, W.ancestor(W.height - RCI))  # None if < 0
    return transition(d, prev_state, prev_choice, W)
```

```python
def transition(d, state: int, choice: str, W: Node) -> tuple[int, str]:
    t = uint64(mtp(W))
    if state == DEFINED:
        if W.height < SVH:
            return (DEFINED, "")
        if t >= d.expire_time:
            return (FAILED, "")
        if required_stake_version(W) < d.version:
            return (DEFINED, "")
        if not pow_majority(int32(d.version), W, BLOCK_REJECT_NUM_REQUIRED):
            return (DEFINED, "")
        if t >= d.start_time:                    # always true at this point
            return (STARTED, "")
        return (DEFINED, "")

    if state == STARTED:
        if t >= d.expire_time:
            return (FAILED, "")
        shift = trailing_zeros16(d.mask)
        counts = [0] * len(d.choices)            # uint32
        non_abstain = 0                          # uint32
        node = W
        for i in range(RCI):                     # the RCI blocks ending with W
            for vote in node.votes:
                if vote.version != d.version:
                    continue
                idx = (vote.bits & d.mask) >> shift
                if idx > len(d.choices) - 1:
                    continue                     # invalid choice: ignored
                counts[idx] += 1
                if not d.choices[idx].is_abstain:
                    non_abstain += 1
            node = node.parent
        if non_abstain < RULE_CHANGE_ACTIVATION_QUORUM:
            return (STARTED, "")
        threshold = non_abstain * RULE_CHANGE_ACTIVATION_MULTIPLIER \
            // RULE_CHANGE_ACTIVATION_DIVISOR
        for idx, c in enumerate(d.choices):
            if c.is_abstain or counts[idx] < threshold:
                continue
            if c.is_no:
                return (FAILED, c.id)
            return (LOCKED_IN, c.id)
        return (STARTED, "")

    if state == LOCKED_IN:
        return (ACTIVE, choice)
    return (state, choice)                       # ACTIVE and FAILED are terminal
```

dcrd: internal/blockchain/thresholdstate.go · nextThresholdState
internal/blockchain/blockindex.go · CalcPastMedianTime

RULE 13.5 Threshold state

For a deployment d that has no forced choice, the threshold state of d for a block B other than the genesis block is `deployment_state(d, B)` of Algorithm 13.4, using the vote fields of Rule 13.2, `required_stake_version` of Algorithm 13.3, `pow_majority` of Algorithm 13.2 and the median time of Chapter 10.

dcrd: internal/blockchain/thresholdstate.go · nextThresholdState, agendaState

The tally window of a `Started` interval is exactly that interval: if W is the end of an interval in the `Started` state, the blocks counted are $W.h - \mathrm{RCI} + 1$ through $W.h$, which are the blocks whose state was `Started`

(computed at the previous interval end). Votes for a deployment are therefore counted only while it is `Started`, and votes cast before voting started never count.

13.6.3 Quorum and majority

The quorum counts the valid votes of the deployment's version whose choice is not the abstain choice; that is, the votes for `yes` and `no` together. Abstentions, votes of other versions and votes with an invalid choice index are not counted. With N such votes and $N \geq$ `RuleChangeActivationQuorum`, a choice wins if its count c satisfies

$$c \geq \left\lfloor N \cdot \frac{\text{RuleChangeActivationMultiplier}}{\text{RuleChangeActivationDivisor}} \right\rfloor, \tag{13.1}$$

computed in unsigned 32-bit arithmetic with truncation. The product cannot overflow: N is at most RCI $\times$ TicketsPerBlock $= 40320$ on the main network. The threshold is the truncated product, not a rounded-up 75 %: with $N = 4033$ the threshold is 3024, which is 74.98 % of N.

TEST VECTOR Majority thresholds with main-network parameters (quorum 4032, 3/4)

non-abstaining votes N	4031	4032	4033	4035	40320
threshold floor(3N/4)	3023	3024	3024	3026	30240
quorum met	no	yes	yes	yes	yes

If the quorum is met but no choice reaches the threshold, the deployment stays `Started` and the next interval is tallied afresh; votes do not carry over between intervals. Since the threshold exceeds half of N whenever $N \geq 3$, at most one choice can reach it, so the choice order matters only in principle.

NOTE
dcrd's informational vote-count queries (the `getvoteinfo` RPC) count a vote with an invalid choice index as an abstention and include it in the total. The consensus tally ignores such votes entirely. Figures from the RPC are therefore not always the inputs of Algorithm 13.4.

IMPLEMENTATION PITFALL
Several details in Algorithm 13.4 are easy to get wrong:

- The expire time is tested *before* the tally. If MTP(W) is at or past the expire time, the deployment fails even if the votes of the interval ending at W reached the threshold.
- A deployment can fail without ever starting: the expire check in the `Defined` state comes before the version checks.
- The upgrade conditions (stake version and block version) are checked only in the `Defined` state. Once `Started`, a deployment keeps being tallied whatever later happens to the versions, and a later deployment version does not stop an earlier, undecided one; but votes of the later version no longer count for it, because the version comparison is exact.
- `required_stake_version(W)` is the stake version required for the block *after* W; it is computed from the stake version interval that ends at or before W. When an RCI boundary and an SVI boundary coincide, the interval ending exactly at W is used.
- `LockedIn` becomes `Active` unconditionally after exactly one interval: no expire check, no tally.
- The state machine depends only on the ancestors of the block. Two blocks at the same height on different branches can have different states.

NOTE
Median time is non-decreasing along a chain, because each block's timestamp must exceed the median time of its parent (Chapter 10). Consequently, for a deployment whose expire time is not before its start time (true of every deployment dcrd defines), the early return for MTP(W) $<$ StartTime in `state_at` never discards a state other than `Defined`. The early return is nevertheless part of the algorithm. dcrd evaluates the recursion

iteratively from the oldest uncomputed interval and caches the result for each interval-end block; the cache is an optimization only.

TEST VECTOR Threshold states on a simulated regression-network chain

```
network     regnet: SVH 144, SVI 112, RCI 320, quorum 160, threshold 3/4
blocks      block version 12, header stake version as required;
            blocks at heights >= 144 carry 5 votes each, vote version 12,
            vote bits 0x0005 (approve parent; maxtreasuryspend "yes")
required stake version     0 for heights 1..255, 12 from 256

maxtreasuryspend (v12)     Defined   heights    1 ..  463
                           Started   heights  464 ..  783   (judged at W = 463)
                           LockedIn  heights  784 .. 1103   (choice "yes")
                           Active    heights 1104 ..
blake3pow (v11)            Defined   heights    1 ..  463
                           Started   heights  464 ..        (no further change:
                                     votes of version 12 do not count for v11)
```

Produced by running dcrd's `nextThresholdState` over a chain of block index nodes. The first interval that can be `Started` begins at SVH + RCI = 464.

13.7 Agenda Activation

Agenda-dependent rules consult a single predicate: whether the agenda is active for the block being validated. It is normally the threshold state of the agenda's deployment, but dcrd also supports forced outcomes for test networks, default outcomes for agendas that a network does not define, and hard-coded historical activation points.

RULE 13.6 Agenda activity for a block

An agenda a is **active for** block B if and only if the first applicable case below says so:

1. If B is the genesis block, a is not active.
2. If a is `treasury` and B has height 1, a is not active.
3. If a has a **forced state** on the network (Table 13.7), a is active if and only if that state is `Active`.
4. If a has a **historical anchor** on the network (Table 13.8) with height h_A and hash H_A: if $B.h - 1 < h_A$, a is not active; if the ancestor of B at height h_A has hash H_A, a is active; otherwise continue with case 5.
5. a is active if and only if the threshold state of its deployment for B (Rule 13.5) is `Active`.

dcrd: internal/blockchain/agendas.go · isAgendaActive, isAgendaActivePositional, isTreasuryAgendaActive
internal/blockchain/agendas.go · makeAgendas
internal/blockchain/thresholdstate.go · agendaState

The choice ID of the winning choice does not enter the predicate: every agenda defined so far has a single passing choice.

Table 13.7 Forced states. A forced state replaces the threshold state for every block except the genesis block.

Source	Applies to	Forced state
`ForcedChoiceID` set to a passing choice	the deployment's agenda (not allowed on the main network)	`Active`
`ForcedChoiceID` set to the `no` choice	the deployment's agenda (not allowed on the main network)	`Failed`
agenda required by the software but with no deployment on the network	main network	`Defined`
agenda required by the software but with no deployment on the network	all other networks	`Active` (empty choice)

The agendas required by the software are the thirteen agendas of Table 13.9 other than `lnsupport`. In the current parameters the default case arises twice: `maxblocksize` has no deployment on the main network (so it is never active there) and none on testnet3 (so it is active there, without effect because testnet3 defines a single block size). Forced `yes` choices are used for every deployment on simnet and for `sdiffalgorithm` and `lnfeatures` on testnet3. Under case 2, the `treasury` agenda on simnet is active from height 2, every other forced agenda from height 1.

Table 13.8 Historical activation anchors. The anchor is the parent of the first block for which the agenda is active, so the activation height is the anchor height plus one. Hashes are in display form.

Agenda	Anchor	Active from	Anchor block hash
mainnet			
`sdiffalgorithm`	149247	149248	000000000000139582d056bc20bb352f4e9b248acbb202724f46000e59c9f75
`lnsupport`	149247	149248	000000000000139582d056bc20bb352f4e9b248acbb202724f46000e59c9f75
`lnfeatures`	189567	189568	000000000000005ca2ebe4ef0649128e1cd16d063f80c884cbd865d3af2300ae
`fixlnseqlocks`	342783	342784	000000000000000017c053b63c7ae9bb8e73ee1c13ef3da493042a7a05d7401c
`headercommitments`	431487	431488	000000000000000225b8f795704d4ebc484fad4ab09eec615fb44b4a95c3c26
`treasury`	552447	552448	000000000000000012d3e57c450f632a9877a28534dbffbf85d41a78aadb9acb
`reverttreasurypolicy`	657279	657280	000000000000000000669da8bbb90dad39a2fd889cbdda00bf259776af822fcaa
`explicitverupgrades`	657279	657280	000000000000000000669da8bbb90dad39a2fd889cbdda00bf259776af822fcaa
`autorevocations`	657279	657280	000000000000000000669da8bbb90dad39a2fd889cbdda00bf259776af822fcaa
`changesubsidysplit`	657279	657280	000000000000000000669da8bbb90dad39a2fd889cbdda00bf259776af822fcaa
`blake3pow`	794367	794368	0000000000000000c293d8c67409d05e960447ea25cdaf770e864d995c764ef0
`changesubsidysplitr2`	794367	794368	0000000000000000c293d8c67409d05e960447ea25cdaf770e864d995c764ef0
`maxtreasuryspend`	1052415	1052416	bc1b84dcfd532a01fbd42470d0668fcdbc7e6e4e9108218f55025d9e565dfaea
testnet3			
`fixlnseqlocks`	136847	136848	0000000004232de037b11ee39641910ae0fd0becb968f76f20a40cb7fa64f475
`headercommitments`	323327	323328	0000002438f7146f7dbfea248b4c94c4c35a8c714442a589110f4c7513a6b26a
`treasury`	560207	560208	0000004f5b6cefb4b5c0ae9dd7a92a431c2de697d22aed9215fe595ac8c1ec23
`reverttreasurypolicy`	867647	867648	0000000036af134886ea61aecea318c1f18f349d3acbf2a8c40ced62854b057b
`explicitverupgrades`	867647	867648	0000000036af134886ea61aecea318c1f18f349d3acbf2a8c40ced62854b057b
`autorevocations`	867647	867648	0000000036af134886ea61aecea318c1f18f349d3acbf2a8c40ced62854b057b
`changesubsidysplit`	877727	877728	000000000000e9528cd8a67898b2c02c854c825a3d09c0d83296f051d8c2adac
`blake3pow`	1170047	1170048	000000b396bfeaa6ae6fa9e3cee441d7215191630bdaa9b979a872985caed727
`changesubsidysplitr2`	1170047	1170048	000000b396bfeaa6ae6fa9e3cee441d7215191630bdaa9b979a872985caed727

Agenda	Anchor	Active from	Anchor block hash
`maxtreasuryspend`	1805087	1805088	`f3e25c6c40baaed62b27ad4f8c0ed6be89bc6890b415089e33acf7783ded248d`

Every anchor height is the last block of a rule change interval ($4095 + 8064k$ on the main network, $767 + 5040k$ on testnet3), as an `Active` state must begin at an interval boundary. On the chain that contains the anchor block, case 4 of Rule 13.6 gives exactly the result of the vote tally; the anchors record facts of the chain rather than changing any outcome, and let a node decide activation without the vote data of old blocks. Off that chain they differ in one respect: a block whose parent is below the anchor height is treated as not active even on a branch where the tally would say otherwise. Such a branch would have to fork from the main chain below the anchor height and lock the agenda in earlier than the main chain did. For anchors older than dcrd's fork-rejection checkpoint (two weeks of blocks before the hard-coded assumed-valid block, Chapter 10) such a fork is rejected anyway; for newer anchors, such as that of `maxtreasuryspend` at the pinned commit, case 4 is what decides. An implementation **MUST** apply case 4 as stated to agree with dcrd on every branch.

13.7.1 Which block's state each rule consults

Every agenda-dependent rule in dcrd asks whether the agenda is active for a specific block, always passing that block's parent to the query function, whose result is defined for the block *after* the node passed. In the terms of this document, Table 13.9 states the block B for which each rule evaluates Rule 13.6.

Table 13.9 Consumers of each agenda. B is the block being validated.

Agenda	Effect (specified in)	Active for
`maxblocksize`	Maximum block size becomes the second entry of `MaximumBlockSizes` if the network defines two (simnet, regnet); otherwise no effect (Chapter 10)	B
`sdiffalgorithm`	Stake difficulty algorithm of DCP-0001 for B's `SBits` (Chapter 9)	B
`lnsupport`	None; a non-binding poll	–
`lnfeatures`	`OP_SHA256` and `OP_CHECKSEQUENCEVERIFY` script flags; sequence-lock enforcement for both trees; lock-time finality judged against the median time of B's parent instead of B's timestamp (Chapter 6, Chapter 7, Chapter 10)	B
`fixlnseqlocks`	None in current dcrd: the corrected sequence-lock view of DCP-0004 is applied to every block (Chapter 7)	–
`headercommitments`	Combined merkle root and commitment root in the header (Chapter 10, Chapter 16)	B
`treasury`	Treasury transactions, opcodes, treasurybase and treasury-spend checks (Chapter 12)	B; and B's parent when disconnecting the parent's regular tree on disapproval
`reverttreasurypolicy`	Expenditure limit of DCP-0007 for the treasury spends in a treasury vote interval block B (Chapter 12)	B
`explicitverupgrades`	Maximum transaction and script versions (Section 13.8, Chapter 7)	B
`autorevocations`	Automatic revocations of DCP-0009 (Chapter 8)	B
`changesubsidysplit`	Subsidy split of DCP-0010 unless `changesubsidysplitr2` is also active (Chapter 11)	B
`blake3pow`	Proof-of-work hash and ASERT difficulty for B; the ASERT anchor is the last block for which the agenda is not active, or block 1 where the agenda is forced active (Chapter 9)	B
`changesubsidysplitr2`	Subsidy split of DCP-0012 (Chapter 11)	B
`maxtreasuryspend`	Expenditure limit of DCP-0013, taking precedence over DCP-0007 (Chapter 12)	B

IMPLEMENTATION PITFALL
The rule for a block is selected by the agenda's activity *for that block*, which by construction depends only on its ancestors. An implementation that evaluates the state "at" the block (including the block's own votes) or "at the tip" will activate rules one block, or one interval, early or late. The one rule that looks at a different block is the treasury check applied when a block disapproves its parent: the parent's regular tree is unwound under the treasury activity *for the parent*.

13.8 Explicit Version Upgrades (DCP-0008)

Before DCP-0008 [44], transactions of any version up to 65535 and regular-transaction output scripts of any script version up to 65535 were valid; unknown versions were treated like the newest known transaction version, and scripts of unknown version were considered valid without being executed. That permits soft forks, where old nodes accept data whose meaning they do not know, and it makes version numbers useless as an indication of which rules apply. Once the `explicitverupgrades` agenda is active for a block:

- every transaction in the block, in either tree and of every type, **MUST** have a version of at most 3 (`ErrTxVersionTooHigh`);
- every output of every regular (non-stake) transaction **MUST** have script version 0 (`ErrScriptVersionTooHigh`).

Stake transactions are exempt from the second rule only because their output script versions are already fixed individually by the rules of Chapter 8 and Chapter 12. The per-transaction statement of both rules is in Chapter 7. Transaction version 0 is also below the limit and remains valid. Versions 1, 2 and 3 are the original version, the version that enables relative lock times (DCP-0003) and the version used by treasury transactions (DCP-0006).

The effect is that any future change that needs a new transaction version or script version is necessarily a hard fork gated by a vote: the new deployment raises the maximum for blocks for which it is active. A transaction or script with a given version is then guaranteed to follow the rules defined for that version, so wallets, hardware signers and other software can rely on version numbers without tracking vote outcomes.

NOTE
The reference code in DCP-0008 reports a too-high script version with the transaction-version error kind; dcrd uses a separate error kind. The error kind is not consensus-relevant.

13.9 Deployments

The tables in this section were generated by a Go program that iterates over the `Deployments` of dcrd's `chaincfg` package for each network, so they reproduce the parameters exactly. On every network every deployment has the three choices `abstain` (index 0, bits `0x0000` , `IsAbstain`), `no` (index 1, `IsNo`) and `yes` (index 2); the tables give the bits of `no` and `yes` . Times are Unix seconds, with the UTC date for orientation.

Table 13.10 Main-network deployments

Ver.	Agenda ID / DCP	Mask	Choices	Start time	Expire time	Forced
4	`sdiffalgorithm` DCP-0001	`0x0006`	`no` `0x0002` `yes` `0x0004`	1493164800 2017-04-26	1524700800 2018-04-26	–
4	`lnsupport` –	`0x0018`	`no` `0x0008` `yes` `0x0010`	1493164800 2017-04-26	1508976000 2017-10-26	–
5	`lnfeatures` DCP-0002, DCP-0003	`0x0006`	`no` `0x0002` `yes` `0x0004`	1505260800 2017-09-13	1536796800 2018-09-13	–
6	`fixlnseqlocks` DCP-0004	`0x0006`	`no` `0x0002` `yes` `0x0004`	1548633600 2019-01-28	1580169600 2020-01-28	–

Ver.	Agenda ID / DCP	Mask	Choices	Start time	Expire time	Forced
7	`headercommitments` DCP-0005	`0x0006`	`no` `0x0002` `yes` `0x0004`	1567641600 2019-09-05	1599264000 2020-09-05	–
8	`treasury` DCP-0006	`0x0006`	`no` `0x0002` `yes` `0x0004`	1596240000 2020-08-01	1627776000 2021-08-01	–
9	`reverttreasurypolicy` DCP-0007	`0x0006`	`no` `0x0002` `yes` `0x0004`	1631750400 2021-09-16	1694822400 2023-09-16	–
9	`explicitverupgrades` DCP-0008	`0x0018`	`no` `0x0008` `yes` `0x0010`	1631750400 2021-09-16	1694822400 2023-09-16	–
9	`autorevocations` DCP-0009	`0x0060`	`no` `0x0020` `yes` `0x0040`	1631750400 2021-09-16	1694822400 2023-09-16	–
9	`changesubsidysplit` DCP-0010	`0x0180`	`no` `0x0080` `yes` `0x0100`	1631750400 2021-09-16	1694822400 2023-09-16	–
10	`blake3pow` DCP-0011	`0x0006`	`no` `0x0002` `yes` `0x0004`	1682294400 2023-04-24	1745452800 2025-04-24	–
10	`changesubsidysplitr2` DCP-0012	`0x0060`	`no` `0x0020` `yes` `0x0040`	1682294400 2023-04-24	1745452800 2025-04-24	–
11	`maxtreasuryspend` DCP-0013	`0x0006`	`no` `0x0002` `yes` `0x0004`	1762992000 2025-11-13	1826064000 2027-11-13	–

The main network has no `maxblocksize` deployment (the vote on it took place only on an earlier, since-replaced test network), so by Table 13.7 that agenda is never active on the main network.

Table 13.11 Testnet3 deployments

Ver.	Agenda ID / DCP	Mask	Choices	Start time	Expire time	Forced
5	`sdiffalgorithm` DCP-0001	`0x0006`	`no` `0x0002` `yes` `0x0004`	1493164800 2017-04-26	1524700800 2018-04-26	`yes`
6	`lnfeatures` DCP-0002, DCP-0003	`0x0006`	`no` `0x0002` `yes` `0x0004`	1505260800 2017-09-13	1536796800 2018-09-13	`yes`
7	`fixlnseqlocks` DCP-0004	`0x0006`	`no` `0x0002` `yes` `0x0004`	1548633600 2019-01-28	1580169600 2020-01-28	–
8	`headercommitments` DCP-0005	`0x0006`	`no` `0x0002` `yes` `0x0004`	1567641600 2019-09-05	1599264000 2020-09-05	–
9	`treasury` DCP-0006	`0x0006`	`no` `0x0002` `yes` `0x0004`	1596240000 2020-08-01	1627776000 2021-08-01	–
10	`reverttreasurypolicy` DCP-0007	`0x0006`	`no` `0x0002` `yes` `0x0004`	1631750400 2021-09-16	1694822400 2023-09-16	–
10	`explicitverupgrades` DCP-0008	`0x0018`	`no` `0x0008` `yes` `0x0010`	1631750400 2021-09-16	1694822400 2023-09-16	–
10	`autorevocations` DCP-0009	`0x0060`	`no` `0x0020` `yes` `0x0040`	1631750400 2021-09-16	1694822400 2023-09-16	–
10	`changesubsidysplit` DCP-0010	`0x0180`	`no` `0x0080` `yes` `0x0100`	1631750400 2021-09-16	1694822400 2023-09-16	–
11	`blake3pow` DCP-0011	`0x0006`	`no` `0x0002` `yes` `0x0004`	1682294400 2023-04-24	1745452800 2025-04-24	–
11	`changesubsidysplitr2` DCP-0012	`0x0060`	`no` `0x0020` `yes` `0x0040`	1682294400 2023-04-24	1745452800 2025-04-24	–
12	`maxtreasuryspend` DCP-0013	`0x0006`	`no` `0x0002` `yes` `0x0004`	1762992000 2025-11-13	1826064000 2027-11-13	–

Testnet3 has no `maxblocksize` deployment either; there the default is `Active`, which has no effect because testnet3 defines a single maximum block size. The forced choices on `sdiffalgorithm` and `lnfeatures` make those two agendas active from height 1, and the start and expire times listed for them are ignored.

Table 13.12 Simnet and regnet deployments. Both networks define the same thirteen deployments with start time 0 and expire time 9223372036854775807 ($2^{63}-1$, never reached); they differ only in the forced choice.

Ver.	Agenda ID / DCP	Mask	Choices	simnet forced	regnet forced
4	`maxblocksize` –	`0x0006`	`no 0x0002` `yes 0x0004`	`yes`	–
5	`sdiffalgorithm` DCP-0001	`0x0006`	`no 0x0002` `yes 0x0004`	`yes`	–
6	`lnfeatures` DCP-0002, DCP-0003	`0x0006`	`no 0x0002` `yes 0x0004`	`yes`	–
7	`fixlnseqlocks` DCP-0004	`0x0006`	`no 0x0002` `yes 0x0004`	`yes`	–
8	`headercommitments` DCP-0005	`0x0006`	`no 0x0002` `yes 0x0004`	`yes`	–
9	`treasury` DCP-0006	`0x0006`	`no 0x0002` `yes 0x0004`	`yes`	–
10	`reverttreasurypolicy` DCP-0007	`0x0006`	`no 0x0002` `yes 0x0004`	`yes`	–
10	`explicitverupgrades` DCP-0008	`0x0018`	`no 0x0008` `yes 0x0010`	`yes`	–
10	`autorevocations` DCP-0009	`0x0060`	`no 0x0020` `yes 0x0040`	`yes`	–
10	`changesubsidysplit` DCP-0010	`0x0180`	`no 0x0080` `yes 0x0100`	`yes`	–
11	`blake3pow` DCP-0011	`0x0006`	`no 0x0002` `yes 0x0004`	`yes`	–
11	`changesubsidysplitr2` DCP-0012	`0x0060`	`no 0x0020` `yes 0x0040`	`yes`	–
12	`maxtreasuryspend` DCP-0013	`0x0006`	`no 0x0002` `yes 0x0004`	`yes`	–

On simnet every agenda is active from height 1 (treasury from height 2). On regnet every agenda is subject to an ordinary vote that can start as soon as the upgrade conditions hold, at the earliest for the interval beginning at height $\mathrm{SVH} + \mathrm{RCI} = 464$, as in the simulated trace of Section 13.6.3.

13.10 Main-Network History

Table 13.13 records every main-network vote. The heights are those of the first block of the interval in the given state, as published in the "Voting Results" sections of the DCPs; the activation heights agree with the historical anchors of Table 13.8 (anchor plus one), and each of the three heights is the first block of a rule change interval ($4096 + 8064k$), which was checked with Algorithm 13.1.

Table 13.13 Main-network vote history. Sources: D = "Voting Results" section of the DCP listed; A = historical anchor in dcrd (`agendas.go`); W = consensus vote archive of the Decred documentation [37]; S = derived from the state machine (see text). Every listed vote passed. Percentages are as published in W and are shares of all votes including abstentions.

Agenda	Ver.	Started	Locked in	Active	Yes %	No %	Abst. %	Sources
`sdiffalgorithm`	4	133120	141184	149248	85.94	1.81	12.23	D DCP-0001, A, W
`lnsupport`	4	133120	141184	149248	76.34	1.07	22.58	S, A, W
`lnfeatures`	5	173440	181504	189568	61.98	0.30	37.70	D DCP-0002, DCP-0003, A, W
`fixlnseqlocks`	6	326656	334720	342784	54.46	0.00	45.52	D DCP-0004, A, W
`headercommitments`	7	415360	423424	431488	61.21	0.03	38.74	D DCP-0005, A, W
`treasury`	8	536320	544384	552448	83.18	0.08	16.72	D DCP-0006, A, W

Agenda	Ver.	Started	Locked in	Active	Yes %	No %	Abst. %	Sources
`reverttreasurypolicy`	9	641152	649216	657280	58.66	0.03	41.30	D DCP-0007, A, W
`explicitverupgrades`	9	641152	649216	657280	58.28	0.03	41.68	D DCP-0008, A, W
`autorevocations`	9	641152	649216	657280	58.27	0.03	41.69	D DCP-0009, A, W
`changesubsidysplit`	9	641152	649216	657280	60.14	0.15	39.79	D DCP-0010, A, W
`blake3pow`	10	778240	786304	794368	64.13	0.34	35.52	D DCP-0011, A, W
`changesubsidysplitr2`	10	778240	786304	794368	62.95	0.45	36.59	D DCP-0012, A, W
`maxtreasuryspend`	11	1036288	1044352	1052416	unconfirmed			D DCP-0013, A
`maxblocksize`	no main-network deployment; never voted on the main network							chaincfg

Every main-network vote that has been held passed, each in the first interval in which it was `Started` (the locked-in height is always one interval after the started height). No main-network vote has failed or expired.

`lnsupport` No DCP exists for this agenda, which asked stakeholders whether development of Lightning Network support should begin and has no consensus effect. Its activation height 149248 is given by the historical anchor. Its start and lock-in heights are derived: `Active` at 149248 requires `LockedIn` for the interval starting at 141184 and `Started` for the interval starting at 133120; and it could not have started earlier than `sdiffalgorithm`, whose DCP-0001 gives 133120, because both share the deployment version (hence the same version conditions) and the start time.

`maxtreasuryspend` The approval percentages are not stated in any of the sources used for this document.

Percentages The consensus majority test uses only non-abstaining votes. Computed that way the published figures give, for example, 97.9 % "yes" for `sdiffalgorithm` and 100.0 % for `fixlnseqlocks`. For vote versions 4, 5 and 6 the archive's "Voting End" column gives the first block of the lock-in interval (141184, 181504, 334720) where its entries for later versions give the last block of the voting interval (one less); the DCP heights are used here.

> **NOTE**
> DCP-0005 labels its expire time 1599264000 as "Jan 5th, 2020"; the value corresponds to 2020-09-05, which is also the date in dcrd's parameters. Only the numeric value matters.

The testnet3 activation heights of the voted agendas are given by Table 13.8. On testnet3, `changesubsidysplit` activated two intervals (10080 blocks) after the three other deployment-version-10 agendas.

13.11 Deriving Agenda Status in an Implementation

An implementation **MUST** derive the state of every agenda from the block chain by the rules above, not from a table of activation heights, except to the extent that Rule 13.6 itself uses the historical anchors. The inputs are, for each block: the header version, the header stake version, the header timestamp (for median times), the parent link, and the vote version and vote bits of each vote in the block. Only the last of these requires the block data rather than the header, so:

- A node that has only headers can determine the proof-of-work majorities, but neither the stake version requirement (it depends on the vote versions) nor the vote tallies. It can resolve agenda status from forced states and from the historical anchors (whose anchor hashes it can match against its header chain), and otherwise must wait for the block data. When activation cannot yet be determined, dcrd's early ("positional") checks accept merkle roots that are valid under *either* outcome of `headercommitments`, and check the difficulty bits with a heuristic described in Chapter 9; the exact outcome is enforced once the full context is available.
- The state is constant across a rule change interval and depends only on the interval-end blocks of the ancestry, so it can be computed once per interval-end block and deployment and cached by block hash. The same holds for the stake version requirement per stake version interval.

- Mempool acceptance and block template construction evaluate agenda status for the block that would extend the current tip. That is policy, but it must use the same predicate so that templates are valid.

When a new release adds a deployment version, blocks that an older release rejected may be valid under the new rules, since a change can relax a rule as well as tighten it. dcrd records the newest deployment version it has seen in its database and, on upgrade, clears the failed-validation mark of every block whose median time is at or after the start time of the first new deployment version, so that such blocks are considered again. This is an implementation matter, but any implementation that persists validation failures needs an equivalent.

On networks with forced deployments (simnet for every agenda, testnet3 for `sdiffalgorithm` and `lnfeatures`) and for agendas without a deployment, no tally is ever performed and the outcome is fixed for every block except the genesis block (and, for `treasury`, block 1). These cases are part of the consensus rules of those networks: a node that ignored a forced state would reject their chains from block 1.

CHAPTER 14

Network Parameters

A Decred **network** is an independent chain together with the peers that relay it: it has its own genesis block, its own consensus constants and its own encodings for addresses and keys. dcrd defines four networks, and every constant that differs between them is a field of the structure `chaincfg.Params`. The rest of this document names such a constant in the form `TicketMaturity` and quotes its main-network value; this chapter gives every field for every network, the quantities that follow from them, the genesis blocks, and the few network-specific rules that dcrd hard-codes outside the parameter structure.

14.1 The Four Networks

Table 14.1 summarizes the networks.

- The **main network** (`mainnet`) is the production network that carries the DCR currency.
- The **test network**, version 3 (`testnet3`), is a public network for testing software against realistic conditions with coins of no value. dcrd describes it as the third public iteration of the test network. Its predecessors are not defined in the current source; a remark in the block-size code notes that the `maxblocksize` vote took place on an earlier version of the test network that has since been replaced.
- The **simulation network** (`simnet`) is meant for private use by a group of people running full integration tests between applications such as wallets, voting service providers, pools and block explorers. It has no seeds, so its nodes connect only to peers named explicitly, and its parameters make blocks, ticket maturity and voting intervals very short.
- The **regression test network** (`regnet`) is meant for unit tests and RPC server tests. dcrd states that its values may change even when the change is a hard fork. When started on regnet, dcrd deletes any existing block and UTXO databases, so every run begins again from the genesis block.

Table 14.1 The networks at a glance

	mainnet	testnet3	simnet	regnet
`Name`	`mainnet`	`testnet3`	`simnet`	`regnet`
Magic (`Net`)	`0xd9b400f9`	`0xb194aa75`	`0x12141c16`	`0xdab500fa`
P2P port (`DefaultPort`)	9108	19108	18555	18655
RPC port (not in `chaincfg`)	9109	19109	19556	18656
Address prefix (`NetworkAddressPrefix`)	`D`	`T`	`S`	`R`
Target block time	5 min	2 min	1 s	1 s
Genesis timestamp (UTC)	2016-02-08	2018-08-06	2014-05-28	2018-10-03

Nodes of different networks are kept apart at three levels.

Network magic Every peer-to-peer message begins with the 4-byte network magic `Net`, a `uint32` serialized little-endian. A node reading a message whose magic differs from its own network's treats the message as coming from another network and fails the read (Chapter 15).

Genesis block Each network has its own genesis block (Section 14.6), so no block of one network can extend the chain of another: every chain is anchored to its network's genesis hash.

Encodings Addresses, WIF private keys and extended keys carry network-specific version bytes (Table 14.16), so that a string produced for one network fails to decode on another and funds are not sent to an address on the wrong network by mistake.

The RPC ports in Table 14.1 are defined by the dcrd application rather than by `chaincfg`. Like the peer-to-peer port, they are defaults that an operator may change. Source: `params.go`.

14.2 Reading the Parameter Tables

`chaincfg.Params` has 85 fields at the reference commit, including the unexported field `seeders`. Each appears exactly once in Table 14.2 to Table 14.16, grouped by theme, with its dcrd field name, a one-line meaning and its value on each network. A value that is the same on all four networks is printed once across the four value columns. Values too long for a column (hashes, scripts, keys, 256-bit integers) are given in separate tables referenced from the cell. The tables were produced by a program that imports dcrd's `chaincfg` package and walks the structure by reflection, refusing to run if any field is not covered; no value was transcribed by hand.

Beside each field name a tag says how dcrd uses it:

consensus read by the validation code; an implementation **MUST** use the value exactly.
node used for networking, synchronization or RPC; not part of consensus.
policy a default for mempool and relay policy (Chapter 17).
encoding used to encode and decode addresses and keys (Chapter 5).
unused not read anywhere in dcrd at the reference commit.

Durations are given in seconds, with the equivalent in larger units below; amounts are in atoms with the DCR equivalent below. Compact difficulty values ("bits", Chapter 9) are given in hexadecimal.

> **IMPLEMENTATION PITFALL**
> Several doc comments in `chaincfg/params.go` do not match how the field is used, and a field's "deprecated" marking does not mean it is unused. In particular: `WorkRewardProportion`, `StakeRewardProportion` and `BlockTaxProportion` are marked deprecated but define the original subsidy split, which still governs every block before the DCP-0010 agenda activated, and the treasury share in every era is still `BlockTaxProportion` divided by the sum of the three (Chapter 11); `BlockTaxProportion` is documented as an inverse percentage but is used as a proportion (1 of a total of 10); the `V2` proportions are not read at all; `WorkDiffWindowSize` is documented as a number of windows but is the number of blocks per window; and `ReduceMinDifficulty` is deprecated but still consensus-critical on testnet3. The tags in the tables reflect use, not comments.

14.3 Parameter Tables

14.3.1 Identity, Peer-to-Peer and Policy

Table 14.2 Identity, peer-to-peer and policy parameters

Field and meaning	mainnet	testnet3	simnet	regnet
Name NODE Human-readable identifier of the network.	mainnet	testnet3	simnet	regnet
Net NODE Network magic: the `uint32` that begins every peer-to-peer message header, serialized little-endian (byte order shown below the value). Consensus code also uses it to identify the network for the hard-coded exceptions of Section 14.5.	0xd9b400f9 wire: f9 00 b4 d9	0xb194aa75 wire: 75 aa 94 b1	0x12141c16 wire: 16 1c 14 12	0xdab500fa wire: fa 00 b5 da
DefaultPort NODE Default TCP port for peer-to-peer connections.	9108	19108	18555	18655
DNSSeeds UNUSED Deprecated list of DNS seed host names (with a flag for service-flag filtering support); dcrd no longer reads it. Listed in Table 14.3.	3 entries	3 entries	0 entries	0 entries

Field and meaning	mainnet	testnet3	simnet	regnet
seeders NODE Unexported list of HTTPS seeder host names that dcrd queries for peer addresses (accessor `Seeders()`). Listed in Table 14.3.	4 entries	3 entries	0 entries	0 entries
GenerateSupported NODE Whether the node's built-in CPU mining RPC is permitted.	false	true	true	true
AcceptNonStdTxs POLICY Default for the mempool option to accept and relay non-standard transactions.	false	true	true	true

Table 14.3 DNS seeds and HTTPS seeders

Field	Network	Value
DNSSeeds	mainnet	`mainnet-seed.decred.mindcry.org` (HasFiltering: true)
	mainnet	`mainnet-seed.decred.netpurgatory.com` (HasFiltering: true)
	mainnet	`mainnet-seed.decred.org` (HasFiltering: true)
	testnet3	`testnet-seed.decred.mindcry.org` (HasFiltering: true)
	testnet3	`testnet-seed.decred.netpurgatory.com` (HasFiltering: true)
	testnet3	`testnet-seed.decred.org` (HasFiltering: true)
	simnet	none
	regnet	none
seeders	mainnet	`mainnet-seed-1.decred.org`
	mainnet	`mainnet-seed-2.decred.org`
	mainnet	`mainnet-seed.jholdstock.uk`
	mainnet	`dcr-seed.jz.bz`
	testnet3	`testnet-seed-1.decred.org`
	testnet3	`testnet-seed-2.decred.org`
	testnet3	`testnet-seed.jholdstock.uk`
	simnet	none
	regnet	none

The seed host names are informative: they are how dcrd bootstraps peer discovery, not part of the protocol. The `DNSSeeds` list is retained in the structure for compatibility only.

14.3.2 Genesis

Table 14.4 Genesis parameters

Field and meaning	mainnet	testnet3	simnet	regnet
GenesisBlock CONSENSUS The block at height 0. Specified field by field in Section 14.6.	see Section 14.6			
GenesisHash CONSENSUS Hash of the genesis block header (display form in Table 14.23).	see Table 14.23			

14.3.3 Block Limits and Proof of Work

Table 14.5 Block limits and proof-of-work parameters

Field and meaning	mainnet	testnet3	simnet	regnet
MaximumBlockSizes CONSENSUS Maximum serialized block size in bytes. The first entry applies by default; the second, where present, applies once the `maxblocksize` agenda is active.	393216	1310720	1000000, 1310720	1000000, 1310720

Field and meaning	mainnet	testnet3	simnet	regnet
MaxTxSize CONSENSUS Maximum serialized transaction size in bytes.	393216	1000000	1000000	1000000
TargetTimePerBlock CONSENSUS Target interval between blocks.	300 s 5 min	120 s 2 min	1 s	1 s
PowLimit CONSENSUS Highest permitted proof-of-work target, as a 256-bit integer (full hexadecimal in Table 14.6).	2^224 - 1	2^232 - 1	2^255 - 1	2^255 - 1
PowLimitBits CONSENSUS PowLimit in compact form; because of the compact format's limited precision it is not exactly equal to PowLimit (expanded value in Table 14.6).	0x1d00ffff	0x1e00ffff	0x207fffff	0x207fffff
ReduceMinDifficulty CONSENSUS Deprecated. Whether the minimum-difficulty reduction of test networks applies (testnet3 only, and only below a hard-coded height; see Section 14.5).	false	true	false	false
MinDiffReductionTime CONSENSUS Deprecated. Time without a block after which the minimum-difficulty reduction applies, when ReduceMinDifficulty is set.	0 s	600 s 10 min	0 s	0 s
WorkDiffAlpha CONSENSUS Version 1 (BLAKE-256) difficulty algorithm: smoothing value of the weighted average.	1			
WorkDiffWindowSize CONSENSUS Version 1 algorithm: number of blocks in one window, which is also the retarget interval. (The doc comment in dcrd wrongly calls it a number of windows.)	144	144	8	8
WorkDiffWindows CONSENSUS Version 1 algorithm: number of windows in the weighted average.	20	20	4	4
TargetTimespan CONSENSUS Version 1 algorithm: target duration of one window, equal to WorkDiffWindowSize times TargetTimePerBlock.	43200 s 12 h	17280 s 288 min	8 s	8 s
RetargetAdjustmentFactor CONSENSUS Version 1 algorithm: bound on the factor by which the difficulty may change at one retarget. The original (version 1) stake difficulty algorithm uses the same bound.	4			
WorkDiffV2Blake3StartBits CONSENSUS Version 2 (ASERT, BLAKE3) algorithm: target, in compact form, used at the start of BLAKE3 proof of work.	0x1b00a5a6	0x1e00ffff	0x207fffff	0x207fffff
WorkDiffV2HalfLifeSecs CONSENSUS Version 2 algorithm: half-life in seconds; the difficulty halves or doubles for every multiple of this time the chain is behind or ahead of schedule.	43200 12 h	720 12 min	6	6

Table 14.6 `PowLimit`, and the compact values `PowLimitBits` and `WorkDiffV2Blake3StartBits` expanded, as 256-bit integers in big-endian hexadecimal

Field	Network	Value
PowLimit	mainnet	00000000ffffffffffffffffffffffffffffffffffffffffffffffffffffffff
	testnet3	000000ffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
	simnet	7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
	regnet	7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
PowLimitBits	mainnet	00000000ffff0000000000000000000000000000000000000000000000000000
	testnet3	000000ffff0000000000000000000000000000000000000000000000000000
	simnet	7fffff0000000000000000000000000000000000000000000000000000000000
	regnet	7fffff0000000000000000000000000000000000000000000000000000000000
WorkDiffV2Blake3StartBits	mainnet	000000000000a5a60000000000000000000000000000000000000000000000000
	testnet3	000000ffff0000000000000000000000000000000000000000000000000000
	simnet	7fffff0000000000000000000000000000000000000000000000000000000000
	regnet	7fffff0000000000000000000000000000000000000000000000000000000000

`MaximumBlockSizes` has a second entry only on simnet and regnet. dcrd selects the second entry for a block when the `maxblocksize` agenda is active for it and the list has more than one entry, and the first entry otherwise (`internal/blockchain/chain.go · maxBlockSize`). Since an agenda that has no deployment on a network other than mainnet is treated as always active (Section 14.3.6), testnet3 would select a second entry if it had one; it does not, so its limit is always 1310720 bytes. The proof-of-work algorithms that use these parameters are specified in Chapter 9.

14.3.4 Subsidy

Table 14.7 Subsidy and organization parameters

Field and meaning	mainnet	testnet3	simnet	regnet
BaseSubsidy CONSENSUS Block subsidy, in atoms, before any reduction.	3119582664 31.19582664 DCR	2500000000 25 DCR	50000000000 500 DCR	50000000000 500 DCR
MulSubsidy CONSENSUS Multiplier applied to the subsidy at each reduction.	100			
DivSubsidy CONSENSUS Divisor applied to the subsidy at each reduction (after the multiplication).	101			
SubsidyReductionInterval CONSENSUS Number of blocks between subsidy reductions.	6144	2048	128	128
WorkRewardProportion CONSENSUS Proof-of-work share of the original subsidy split, as a proportion of the sum `WorkRewardProportion` + `StakeRewardProportion` + `BlockTaxProportion`. Marked deprecated in dcrd but still read by the original split.	6			
StakeRewardProportion CONSENSUS Proof-of-stake (all votes of a block together) share of the original split, as a proportion of the same sum.	3			
BlockTaxProportion CONSENSUS Treasury share, as a proportion of the same sum, in every era: the later splits change only the proof-of-work and proof-of-stake shares. (The doc comment in dcrd describes it as an inverse percentage, which does not match its use.)	1			
WorkRewardProportionV2 UNUSED Proof-of-work proportion of the DCP-0010 split. Not read: the DCP-0010 and DCP-0012 splits are hard-coded in the subsidy code (Chapter 11).	1			

Field and meaning	mainnet	testnet3	simnet	regnet
StakeRewardProportionV2 UNUSED Proof-of-stake proportion of the DCP-0010 split. Not read, as for WorkRewardProportionV2.			8	
OrganizationPkScript CONSENSUS Output script that the treasury share of the coinbase must pay before the treasury agenda (DCP-0006) is active (Table 14.8).			see Table 14.8	
OrganizationPkScriptVersion CONSENSUS Script version required on that coinbase output.			0	
BlockOneLedger CONSENSUS Outputs (script version, script, amount) that the coinbase of block 1 must pay exactly. Contents in Chapter 11; count and total in Section 14.7.	3146 outputs	2 outputs	3 outputs	3 outputs

Table 14.8 Organization (pre-treasury) output script

Field	Network	Value
OrganizationPkScript	mainnet	a914f5916158e3e2c4551c1796708db8367207ed13bb87 OP_HASH160 f5916158e3e2c4551c1796708db8367207ed13bb OP_EQUAL address Dcur2mcGjmENx4DhNqDctW5wJCVyT3Qeqkx
	testnet3	a914d585cd7426d25b4ea5faf1e6987aacfeda3db94287 OP_HASH160 d585cd7426d25b4ea5faf1e6987aacfeda3db942 OP_EQUAL address TcrypGAcGCRVXrES7hWqVZb5oLJKCZEtoL1
	simnet	a914cbb08d6ca783b533b2c7d24a51fbca92d937bf9987 OP_HASH160 cbb08d6ca783b533b2c7d24a51fbca92d937bf99 OP_EQUAL address ScuQxvveKGfpG1ypt6u27F99Anf7EW3cqhq
	regnet	a9146913bcc838bd0087fb3f6b3c868423d5e300078d87 OP_HASH160 6913bcc838bd0087fb3f6b3c868423d5e300078d OP_EQUAL address RcQR65gasxuzf7mUeBXeAux6Z37joPuUwUN

The organization script is the pay-to-script-hash output to which the treasury share of every coinbase was paid before the decentralized treasury (DCP-0006, Chapter 12) replaced it with a treasury base. How the subsidy parameters are combined is specified in Chapter 11.

14.3.5 Proof of Stake

Table 14.9 Proof-of-stake parameters

Field and meaning	mainnet	testnet3	simnet	regnet
MinimumStakeDiff CONSENSUS Minimum ticket price (stake difficulty) in atoms.	200000000 2 DCR	20000000 0.2 DCR	20000 0.0002 DCR	20000 0.0002 DCR
TicketPoolSize CONSENSUS Ticket pool size in units of blocks: the stake difficulty algorithms target a live pool of TicketPoolSize times TicketsPerBlock tickets.	8192	1024	64	64
TicketsPerBlock CONSENSUS Number of tickets selected to vote on each block, which is also the maximum number of votes a block may contain.			5	
TicketMaturity CONSENSUS Number of blocks for a ticket to mature after the block that includes it (exact semantics in Chapter 8).	256	16	16	16
TicketExpiry CONSENSUS Number of blocks after maturity after which a ticket that has not been selected expires.	40960	6144	384	384
CoinbaseMaturity CONSENSUS Maturity depth, in blocks, for spending coinbase outputs; also the depth for vote and revocation outputs once the treasury agenda is active, and for treasury balance changes (Chapter 7, Chapter 12).	256	16	16	16

Field and meaning	mainnet	testnet3	simnet	regnet
SStxChangeMaturity CONSENSUS Maturity depth for ticket change outputs, and for vote and revocation outputs before the treasury agenda is active.			1	
TicketPoolSizeWeight CONSENSUS Weight applied to the pool-size deviation in version 1 of the stake difficulty algorithm; not read by version 2.			4	
StakeDiffAlpha CONSENSUS Smoothing value of the original (version 1) stake difficulty algorithm.			1	
StakeDiffWindowSize CONSENSUS Number of blocks in one stake difficulty window; the ticket price changes only at multiples of this height.	144	144	8	8
StakeDiffWindows CONSENSUS Number of windows in the weighted average of version 1 of the stake difficulty algorithm; not read by version 2.	20	20	8	8
MaxFreshStakePerBlock CONSENSUS Maximum number of ticket purchases a block may contain.			20	
StakeEnabledHeight CONSENSUS First height at which a ticket can be live (the earliest possible maturity).	512	32	32	32
StakeValidationHeight CONSENSUS First height whose block must contain votes (at least a majority of `TicketsPerBlock`). Those votes are on the block one below, which is the first block voted on and the first on which the ticket lottery runs. (The doc comment in dcrd says the block at this height contains no votes, which does not match the code.)	4096	768	144	144
StakeBaseSigScript CONSENSUS Required signature script of the stakebase input of every vote.	0000	0000	0000	7357

The stake system that these parameters configure is specified in Chapter 8; the two stake difficulty algorithms are in Chapter 9. Regnet's `StakeBaseSigScript` is `7357`, unlike the `0000` of the other networks; a vote on regnet **MUST** carry exactly those two bytes.

14.3.6 Voting, Versions and Upgrades

Table 14.10 Voting, version and upgrade parameters

Field and meaning	mainnet	testnet3	simnet	regnet
StakeVersionInterval CONSENSUS Number of blocks in a stake version interval.	2016	2016	112	112
StakeMajorityMultiplier CONSENSUS Numerator of the stake-version majority fraction.			3	
StakeMajorityDivisor CONSENSUS Denominator of the stake-version majority fraction.			4	
RuleChangeActivationQuorum CONSENSUS Minimum number of non-abstaining votes in a rule change interval for an agenda vote to count.	4032	2520	160	160
RuleChangeActivationMultiplier CONSENSUS Numerator of the fraction of non-abstaining votes a choice needs to win.			3	
RuleChangeActivationDivisor CONSENSUS Denominator of that fraction.			4	
RuleChangeActivationInterval CONSENSUS Number of blocks in a rule change interval.	8064	5040	320	320
Deployments CONSENSUS Agendas keyed by stake version. Summary in Table 14.11; full definitions in Chapter 13.	13 agendas	12 agendas	13 agendas	13 agendas

Field and meaning	mainnet	testnet3	simnet	regnet
BlockEnforceNumRequired CONSENSUS Legacy block-version upgrade: number of recent blocks at a version needed to enforce rules of that version.	750	51	51	51
BlockRejectNumRequired CONSENSUS Legacy block-version upgrade: number of recent blocks at a higher version needed to reject older block versions.	950	75	75	75
BlockUpgradeNumToCheck CONSENSUS Legacy block-version upgrade: number of most recent blocks examined.	1000	100	100	100

Table 14.11 summarizes `Deployments`. Each cell gives the stake version under which the agenda is defined and its vote mask, and below it either the median-time voting period (start and expiry dates, UTC), the forced choice, or "start 0, never expires" for agendas that are always open for voting. The agenda definitions in full (descriptions, choices and bits), the voting rules, and the outcome of every vote are owned by Chapter 13.

Table 14.11 Agendas per network: stake version and vote mask, then the voting period, forced choice or default

Agenda	mainnet	testnet3	simnet	regnet
maxblocksize	absent never active	absent always active	v4 0x0006 forced: yes	v4 0x0006 start 0, never expires
sdiffalgorithm	v4 0x0006 2017-04-26 to 2018-04-26	v5 0x0006 forced: yes	v5 0x0006 forced: yes	v5 0x0006 start 0, never expires
lnsupport no consensus effect	v4 0x0018 2017-04-26 to 2017-10-26	absent	absent	absent
lnfeatures	v5 0x0006 2017-09-13 to 2018-09-13	v6 0x0006 forced: yes	v6 0x0006 forced: yes	v6 0x0006 start 0, never expires
fixlnseqlocks	v6 0x0006 2019-01-28 to 2020-01-28	v7 0x0006 2019-01-28 to 2020-01-28	v7 0x0006 forced: yes	v7 0x0006 start 0, never expires
headercommitments	v7 0x0006 2019-09-05 to 2020-09-05	v8 0x0006 2019-09-05 to 2020-09-05	v8 0x0006 forced: yes	v8 0x0006 start 0, never expires
treasury	v8 0x0006 2020-08-01 to 2021-08-01	v9 0x0006 2020-08-01 to 2021-08-01	v9 0x0006 forced: yes	v9 0x0006 start 0, never expires
reverttreasurypolicy	v9 0x0006 2021-09-16 to 2023-09-16	v10 0x0006 2021-09-16 to 2023-09-16	v10 0x0006 forced: yes	v10 0x0006 start 0, never expires
explicitverupgrades	v9 0x0018 2021-09-16 to 2023-09-16	v10 0x0018 2021-09-16 to 2023-09-16	v10 0x0018 forced: yes	v10 0x0018 start 0, never expires
autorevocations	v9 0x0060 2021-09-16 to 2023-09-16	v10 0x0060 2021-09-16 to 2023-09-16	v10 0x0060 forced: yes	v10 0x0060 start 0, never expires
changesubsidysplit	v9 0x0180 2021-09-16 to 2023-09-16	v10 0x0180 2021-09-16 to 2023-09-16	v10 0x0180 forced: yes	v10 0x0180 start 0, never expires
blake3pow	v10 0x0006 2023-04-24 to 2025-04-24	v11 0x0006 2023-04-24 to 2025-04-24	v11 0x0006 forced: yes	v11 0x0006 start 0, never expires
changesubsidysplitr2	v10 0x0060 2023-04-24 to 2025-04-24	v11 0x0060 2023-04-24 to 2025-04-24	v11 0x0060 forced: yes	v11 0x0060 start 0, never expires
maxtreasuryspend	v11 0x0006 2025-11-13 to 2027-11-13	v12 0x0006 2025-11-13 to 2027-11-13	v12 0x0006 forced: yes	v12 0x0006 start 0, never expires

dcrd maintains a list of the agenda identifiers that influence consensus (every row except `lnsupport`, whose vote only requested development work). When such an agenda has no deployment in a network's parameters, dcrd creates a default one that is never active on the main network and always active on every other network; these are the cells marked "absent" with "never active" or "always active" (`internal/blockchain/agendas.go · requiredAgendaIDs`). A forced choice makes the agenda behave as if that choice had won, without a vote; simnet uses this for every agenda and testnet3 for its two oldest, to activate rules that mainnet adopted by vote. Regnet defines every agenda without a forced choice, so on regnet each rule change is put to an actual vote.

> **NOTE**
> On mainnet and testnet3 dcrd also records the outcome of past votes as historical facts (the block at which each agenda's activation was determined), which lets it decide activation without re-tallying votes. This list lives in `internal/blockchain/agendas.go · makeHistoricalAgendas` rather than in `chaincfg` and is described in Chapter 13.

14.3.7 Treasury

Table 14.12 Treasury parameters

Field and meaning	mainnet	testnet3	simnet	regnet
PiKeys CONSENSUS Public keys authorized to sign treasury spends (Table 14.13).	2 keys			
TreasuryVoteInterval CONSENSUS Treasury vote interval (TVI) in blocks: treasury spends may only be included at heights that are non-zero multiples of it.	288	60	48	4
TreasuryVoteIntervalMultiplier CONSENSUS Number of TVIs in a treasury spend voting window.	12	4	3	3
TreasuryVoteQuorumMultiplier CONSENSUS Numerator of the treasury spend quorum, as a fraction of the votes possible in the voting window.	1			
TreasuryVoteQuorumDivisor CONSENSUS Denominator of that quorum.	5			
TreasuryVoteRequiredMultiplier CONSENSUS Numerator of the fraction of yes votes a treasury spend requires.	3			
TreasuryVoteRequiredDivisor CONSENSUS Denominator of that fraction.	5			
TreasuryExpenditureWindow CONSENSUS Number of voting windows (TVI times multiplier) in one expenditure policy window.	2	4	4	4
TreasuryExpenditurePolicy CONSENSUS Original (DCP-0006) policy only: number of earlier expenditure windows averaged.	6	3	3	3
TreasuryExpenditureBootstrap CONSENSUS Original (DCP-0006) policy only: average expenditure, in atoms, assumed when none of those windows contains a treasury spend.	1600000000000 16000 DCR	1000000000000 10000 DCR	10000000000 100 DCR	10000000000 100 DCR

Table 14.13 Sanctioned Politeia keys (compressed secp256k1 public keys)

Field	Network	Value
PiKeys[0]	mainnet	03f6e7041f1cf51ee10e0a01cd2b0385ce3cd9debaabb2296f7e9dee9329da946c
PiKeys[1]	mainnet	0319a37405cb4d1691971847d7719cfce70857c0f6e97d7c9174a3998cf0ab86dd
PiKeys[0]	testnet3	03beca9bbd227ca6bb5a58e03a36ba2b52fff09093bd7a50aee1193bccd257fb8a
PiKeys[1]	testnet3	03e647c014f55265da506781f0b2d67674c35cb59b873d9926d483c4ced9a7bbd3
PiKeys[0]	simnet	02a36b785d584555696b69d1b2bbeff4010332b301e3edd316d79438554cacb3e7
PiKeys[1]	simnet	02b2c110e7b560aa9e1545dd18dd9f7e74a3ba036297a696050c0256f1f69479d7
PiKeys[0]	regnet	03b459ccf3ce4935a676414fd9ec93ecf7c9dad081a52ed6993bf073c627499388
PiKeys[1]	regnet	02e3af1209f4d39dd8b448ef0a5375befa85bbc50be0aa0936379d67444184a2c3

The treasury rules are specified in Chapter 12. All three expenditure policies measure spending over a policy window whose length in blocks is the product of `TreasuryVoteInterval`, `TreasuryVoteIntervalMultiplier` and `TreasuryExpenditureWindow`. Of the remaining parameters, only the original DCP-0006 policy uses `TreasuryExpenditurePolicy` and `TreasuryExpenditureBootstrap`; the DCP-0007 policy uses neither; the DCP-0013 policy has no parameters of its own in `chaincfg` and derives a floor for its limit from `BaseSubsidy` (Table 14.18). For simnet and regnet, the private keys belonging to the two Pi keys are published in the dcrd source, so anyone can sign treasury spends on those networks.

14.3.8 Checkpoints and Synchronization

Table 14.14 Checkpoint and synchronization parameters

Field and meaning	mainnet	testnet3	simnet	regnet
Checkpoints UNUSED Deprecated. Formerly the manual checkpoints; empty on every network and not read.			`empty`	
AssumeValid NODE Hash of an externally verified block; some checks are skipped for its ancestors, and it anchors the rejection of old forks (Chapter 10). The fork rejection decides which blocks a node accepts, so an implementation that omits it accepts side-chain blocks that dcrd rejects; the check uses this hard-coded value, not an operator's override, and `--allowoldforks` disables it. All zero means not set.	see Table 14.15	see Table 14.15	`zero (unset)`	`zero (unset)`
MinKnownChainWork NODE Minimum total chain work known at release time, used during initial synchronization. Unset on simnet and regnet.	see Table 14.15	see Table 14.15	`nil (unset)`	`nil (unset)`

Table 14.15 Assume-valid block (display form) and minimum known chain work (hexadecimal)

Field	Network	Value
AssumeValid	mainnet	458d6a8e11c916d4149ca8bc5c7aaaaf16cc61971b0c20764c07edf85df44eb6
	testnet3	e5d29f04aa33d19dfe4935ca2e70d8f640a7263b653f57ce600b6829d6f2cf48
MinKnownChainWork	mainnet	0000000000000000000000000000000000000000000243868232c14b8224643b6
	testnet3	0000000000000000000000000000000000000000000000000f377240e146195df

None of the networks has manual checkpoints any longer. The assume-valid block and the minimum known chain work are refreshed with dcrd releases and are not consensus rules in themselves; how the assume-valid block anchors the rejection of old forks, and that this rejection is disabled when `AssumeValid` is zero (simnet and regnet), is specified in Chapter 10.

IMPLEMENTATION PITFALL
Since the `blake3pow` agenda (DCP-0011) the proof of work is checked against a BLAKE3 hash of the header, while the block hash that identifies a block is still the BLAKE-256 hash (Chapter 9). Block hashes of blocks

mined since then in general do not begin with zeros, as the assume-valid hashes in Table 14.15 show. An implementation that sanity-checks block hashes for leading zeros rejects the current chain.

14.3.9 Address and Key Encoding

Table 14.16 Address and key encoding parameters

Field and meaning	mainnet	testnet3	simnet	regnet
NetworkAddressPrefix ENCODING First character of every address on the network. Not read by dcrd itself; the character follows from the address prefixes.	D	T	S	R
PubKeyAddrID ENCODING Two-byte prefix of pay-to-pubkey addresses (all three signature types).	0x1386	0x28f7	0x276f	0x25e5
PubKeyHashAddrID ENCODING Prefix of pay-to-pubkey-hash addresses, secp256k1 ECDSA.	0x073f	0x0f21	0x0e91	0x0e00
PKHEdwardsAddrID ENCODING Prefix of pay-to-pubkey-hash addresses, Ed25519.	0x071f	0x0f01	0x0e71	0x0de0
PKHSchnorrAddrID ENCODING Prefix of pay-to-pubkey-hash addresses, secp256k1 Schnorr.	0x0701	0x0ee3	0x0e53	0x0dc2
ScriptHashAddrID ENCODING Prefix of pay-to-script-hash addresses.	0x071a	0x0efc	0x0e6c	0x0ddb
PrivateKeyID ENCODING Prefix of WIF-encoded private keys.	0x22de	0x230e	0x2307	0x22fe
HDPrivateKeyID ENCODING Version bytes of serialized extended private keys.	0x02fda4e8	0x04358397	0x0420b903	0xeab40448
HDPublicKeyID ENCODING Version bytes of serialized extended public keys.	0x02fda926	0x043587d1	0x0420bd3d	0xeab4f987
SLIP0044CoinType ENCODING SLIP-0044 coin type used in hierarchical deterministic derivation paths. Read by wallets, not by dcrd.	42	1	1	1
LegacyCoinType ENCODING Coin type used by older wallets, kept for backwards compatibility. Read by wallets, not by dcrd.	20	11	115	1

Table 14.17 shows the leading characters that each prefix produces. They were computed by Base58-encoding the prefix followed by an all-zero and by an all-`0xff` payload and checksum of the correct length and keeping the characters common to both, so every valid string of that type begins with them; the encoded length is given in parentheses. The payload is 20 bytes for the hash-based addresses, 33 bytes (a type byte and a 32-byte coordinate or key) for pay-to-pubkey addresses and WIF keys, and 74 bytes after the 4-byte version for extended keys. The encodings themselves are specified in Chapter 5.

Table 14.17 Address and key prefixes with the leading characters they produce

Encoding and field	mainnet	testnet3	simnet	regnet
Pay-to-pubkey (33-byte payload) **PubKeyAddrID**	0x1386 Dk... (53 chars)	0x28f7 Tk... (53 chars)	0x276f Sk... (53 chars)	0x25e5 Rk... (53 chars)
P2PKH, secp256k1 ECDSA **PubKeyHashAddrID**	0x073f Ds... (35 chars)	0x0f21 Ts... (35 chars)	0x0e91 Ss... (35 chars)	0x0e00 Rs... (35 chars)
P2PKH, Ed25519 **PKHEdwardsAddrID**	0x071f De... (35 chars)	0x0f01 Te... (35 chars)	0x0e71 Se... (35 chars)	0x0de0 Re... (35 chars)
P2PKH, secp256k1 Schnorr **PKHSchnorrAddrID**	0x0701 DS... (35 chars)	0x0ee3 TS... (35 chars)	0x0e53 SS... (35 chars)	0x0dc2 RS... (35 chars)
Pay-to-script-hash **ScriptHashAddrID**	0x071a Dc... (35 chars)	0x0efc Tc... (35 chars)	0x0e6c Sc... (35 chars)	0x0ddb Rc... (35 chars)
WIF private key (type byte + 32-byte key) **PrivateKeyID**	0x22de Pm... (53 chars)	0x230e Pt... (53 chars)	0x2307 Ps... (53 chars)	0x22fe Pr... (53 chars)

Encoding and field	mainnet	testnet3	simnet	regnet
Extended private key `HDPrivateKeyID`	`0x02fda4e8` dprv... (111 chars)	`0x04358397` tprv... (111 chars)	`0x0420b903` sprv... (111 chars)	`0xeab40448` rprv1... (112 chars)
Extended public key `HDPublicKeyID`	`0x02fda926` dpub... (111 chars)	`0x043587d1` tpub... (111 chars)	`0x0420bd3d` spub... (111 chars)	`0xeab4f987` rpub1... (112 chars)

14.4 Derived Quantities

Table 14.18 lists quantities that follow arithmetically from the parameters and that implementers frequently need. T is `TargetTimePerBlock` in seconds. The two rows marked "Check" verify relations that the dcrd comments state between parameters. The treasury rows follow the computations in `internal/blockchain/treasury.go · checkTSpendHasVotes` and `internal/blockchain/chain.go · New`: a treasury spend voting window spans `TreasuryVoteInterval` × `TreasuryVoteIntervalMultiplier` blocks, the number of votes possible in it is that times `TicketsPerBlock`, and the DCP-0013 floor on the spending limit is $\lfloor \text{BaseSubsidy} / 10 \rfloor$ times the window length. The exact use of each quantity is specified in the chapter that owns it.

Table 14.18 Derived quantities. Times assume blocks arrive exactly at the target rate.

Quantity and formula	mainnet	testnet3	simnet	regnet
Blocks per day $86400/T$	288	720	86400	86400
Expected blocks in two weeks $\lfloor 14 \cdot 86400/T \rfloor$	4032 14 d	10080 14 d	1209600 14 d	1209600 14 d
Check: `TargetTimespan` $=$ `WorkDiffWindowSize` $\cdot T$			true	
V1 PoW retarget interval `WorkDiffWindowSize`	144 12 h	144 4.8 h	8 8 s	8 8 s
V1 PoW averaging span `WorkDiffWindowSize` × `WorkDiffWindows`	2880 10 d	2880 4 d	32 32 s	32 32 s
ASERT half-life in blocks `WorkDiffV2HalfLifeSecs` $/T$	144	6	6	6
Subsidy reduction interval `SubsidyReductionInterval`	6144 21.33 d	2048 2.84 d	128 2.13 min	128 2.13 min
Original split PoW / PoS / treasury proportions over their sum			6/3/1 of 10	
Target live ticket pool `TicketPoolSize` × `TicketsPerBlock`	40960	5120	320	320
Ticket price window `StakeDiffWindowSize`	144 12 h	144 4.8 h	8 8 s	8 8 s
Ticket price averaging span (version 1 algorithm only) `StakeDiffWindowSize` × `StakeDiffWindows`	2880 10 d	2880 4 d	64 1.07 min	64 1.07 min
Ticket maturity `TicketMaturity`	256 21.33 h	16 32 min	16 16 s	16 16 s
Ticket expiry `TicketExpiry`	40960 142.22 d	6144 8.53 d	384 6.4 min	384 6.4 min
Check: `TicketExpiry` $\geq$ `StakeEnabledHeight` + `StakeValidationHeight`			true	
Coinbase maturity `CoinbaseMaturity`	256 21.33 h	16 32 min	16 16 s	16 16 s
Stake enabled height `StakeEnabledHeight`	512 1.78 d	32 1.07 h	32 32 s	32 32 s
Stake validation height `StakeValidationHeight`	4096 14.22 d	768 1.07 d	144 2.4 min	144 2.4 min
Stake version interval `StakeVersionInterval`	2016 7 d	2016 2.8 d	112 1.87 min	112 1.87 min

Quantity and formula	mainnet	testnet3	simnet	regnet
Rule change interval `RuleChangeActivationInterval`	8064 28 d	5040 7 d	320 5.33 min	320 5.33 min
Votes possible per rule change interval `RuleChangeActivationInterval` × `TicketsPerBlock`	40320	25200	1600	1600
Rule change quorum as a share of those `RuleChangeActivationQuorum` / previous row			10 %	
Treasury vote interval (TVI) `TreasuryVoteInterval`	288 1 d	60 2 h	48 48 s	4 4 s
Treasury spend voting window TVI × `TreasuryVoteIntervalMultiplier`	3456 12 d	240 8 h	144 2.4 min	12 12 s
Votes possible in a voting window window × `TicketsPerBlock`	17280	1200	720	60
Treasury spend quorum (votes) possible votes × `TreasuryVoteQuorumMultiplier` / `TreasuryVoteQuorumDivisor`, rounded down	3456	240	144	12
Treasury spend yes fraction `TreasuryVoteRequiredMultiplier` / `TreasuryVoteRequiredDivisor`			60 %	
Expenditure policy window voting window × `TreasuryExpenditureWindow`	6912 24 d	960 1.33 d	576 9.6 min	48 48 s
DCP-0006 averaging span policy window × `TreasuryExpenditurePolicy`	41472 144 d	2880 4 d	1728 28.8 min	144 2.4 min
DCP-0013 spend limit floor (atoms) `BaseSubsidy` /10 (rounded down) × voting window	1078127767296 10781.27767296 DCR	60000000000 600 DCR	720000000000 7200 DCR	60000000000 600 DCR
Block-one ledger total sum of the `BlockOneLedger` amounts	1680000 DCR 168000000000000 atoms	100000 DCR 10000000000000 atoms	300000 DCR 30000000000000 atoms	300000 DCR 30000000000000 atoms

14.5 Network-Specific Rules Outside the Parameters

A few consensus rules depend on the network but are not expressed through `chaincfg.Params`. dcrd selects them by comparing `Net` with the magic of a particular network. They are listed here so that an implementation supporting several networks does not miss them; each is specified in the chapter given.

Table 14.19 Hard-coded network-specific rules

Network	Rule	Specified in
testnet3	The minimum-difficulty reduction of `ReduceMinDifficulty` applies only to blocks below height 962928. From that height a maximum difficulty applies: the target is never allowed below `PowLimit` shifted right by 6 bits, i.e. `00000003ffffffffffffffffffffffffffffffffffffffffffffffffffffffff`, compact `0x1d03ffff`, and blocks at that maximum difficulty must be at least one minute apart. The block at height 962928 **MUST** have hash `0000004fd1b267fd39111d456ff557137824538e6f6776168600e56002e23b93` (display form); any other block at that height is rejected.	Chapter 9
mainnet, testnet3	Exemptions from the DCP-0005 block-version check for blocks that were accepted in violation of it: five specific version-6 blocks on mainnet (heights 413762 to 430191), and every version-7 block up to height 323282 on testnet3.	Chapter 13
mainnet; others	The latest block version below which blocks become rejectable by majority is 11 on mainnet and 12 on every other network.	Chapter 13
mainnet; others	The lowest block version for which dcrd's header-stage difficulty check considers the ASERT result before the agenda state is known is 10 on mainnet and 11 on every other network. This is not itself a consensus rule: the proof-of-work rules of Chapter 9 do not depend on the block version.	Chapter 9
mainnet, testnet3	A list of specific outputs (six on mainnet, one on testnet3) that were spent by a stake transaction in the same block under rules that were later tightened, and that remain acceptable.	Chapter 7

Network	Rule	Specified in
mainnet	A deployment with a forced choice is refused on the main network (`ErrForcedMainNetChoice`).	Chapter 13
mainnet, testnet3	Historical agenda activation anchors (Section 14.3.6).	Chapter 13

14.6 Genesis Blocks

The genesis block is the block at height 0. It has no parent and is not produced by mining: each network's genesis block is a constant of the protocol, written out field by field in `chaincfg` and given here in full.

RULE 14.1 Genesis block
The block at height 0 of a network **MUST** be exactly the genesis block given in this section for that network, whose serialization is given at the end of this section. The genesis block is valid by definition: none of the validation rules of Chapter 10 or Chapter 7 is applied to it, including proof of work, Merkle root and size checks. A block at height 1 **MUST** name the genesis block's hash as its previous block.

dcrd: internal/blockchain/chainio.go · createChainState
internal/blockchain/validate.go · checkBlockHeaderPositional,checkBlockPositional

RULE 14.2 Genesis outputs are not spendable
The transactions of the genesis block are never connected: the UTXO set of a chain whose tip is the genesis block is empty. An input that references an output of the genesis coinbase therefore references an output that does not exist, and the transaction containing it **MUST** be rejected as for any missing input (Chapter 7).

dcrd: internal/blockchain/chainio.go · createChainState
internal/blockchain/utxoviewpoint.go · FetchUtxoView

Besides serving as the parent of block 1, the genesis header's values that matter to later blocks are its timestamp, which is the median time of the chain while the genesis block is the tip, and its difficulty bits, which are the starting point of the proof-of-work difficulty computation (Chapter 9). Agendas are never active for the genesis block. The version 2 compact filter of the genesis block is computed by the ordinary algorithm with no spent scripts and contains one item (Chapter 16).

Table 14.20 lists every header field (Chapter 4 specifies the header layout) and Table 14.21 every field of the single coinbase transaction. No genesis block has stake transactions.

Table 14.20 Genesis block headers

Field	mainnet	testnet3	simnet	regnet
Version	1	6	1	1
PrevBlock	zero			
MerkleRoot	66aa7491... full value below	e7dfbcea... full value below	5c9488e2... full value below	66aa7491... full value below
StakeRoot	zero			
VoteBits	0			
FinalState	000000000000			
Voters	0			
FreshStake	0			
Revocations	0			
PoolSize	0			
Bits	0x1b01ffff	0x1e00ffff	0x207fffff	0x207fffff
SBits	200000000 2 DCR	20000000 0.2 DCR	0 0 DCR	0 0 DCR
Height	0			
Size	0			
Timestamp	1454954400 2016-02-08 18:00:00 UTC	1533513600 2018-08-06 00:00:00 UTC	1401292357 2014-05-28 15:52:37 UTC	1538524800 2018-10-03 00:00:00 UTC
Nonce	0x00000000	0x18aea41a	0x00000000	0x00000000
ExtraData	zero			
StakeVersion	0	6	0	0

Table 14.21 Genesis coinbase transactions

Field	mainnet	testnet3	simnet	regnet
SerType	0 full			
Version	1			
inputs	1			
in: PreviousOutPoint.Hash	zero			
in: PreviousOutPoint.Index	0xffffffff			
in: PreviousOutPoint.Tree	0			
in: Sequence	0xffffffff			
in: ValueIn	-1	-1	0	-1
in: BlockHeight	0x00000000			
in: BlockIndex	0xffffffff	0xffffffff	0x00000000	0xffffffff
in: SignatureScript	0000 2 bytes	0000 2 bytes	77 bytes full value below	0000 2 bytes
outputs	1			
out: Value	0			
out: Version	0			
out: PkScript	32 bytes full value below	32 bytes full value below	67 bytes full value below	32 bytes full value below
LockTime	0			
Expiry	0			

Table 14.22 Genesis coinbase scripts (hexadecimal and dcrd disassembly)

Field	Network	Value
SignatureScript	mainnet	0000 0 0
	testnet3	0000 0 0
	simnet	04ffff001d0104455468652054696d65732030332f4a616e2f32303039204368616e63656c6c6f72206f6e206272696e6b206f66207365636f6e64206261696c6f757420666f722062616e6b73 ffff001d 04 5468652054696d65732030332f4a616e2f32303039204368616e63656c6c6f72206f6e206272696e6b206f66207365636f6e64206261696c6f757420666f722062616e6b73
	regnet	0000 0 0
PkScript	mainnet	801679e98561ada96caec2949a5d41c4cab3851eb740d951c10ecbcf265c1fd9 OP_LEFT 79e98561ada96caec2949a5d41c4cab3851eb740d951 OP_TADD [error] (parse failure: opcode OP_DATA_14 requires 15 bytes, but script only has 7 remaining) class: nonstandard
	testnet3	801679e98561ada96caec2949a5d41c4cab3851eb740d951c10ecbcf265c1fd9 OP_LEFT 79e98561ada96caec2949a5d41c4cab3851eb740d951 OP_TADD [error] (parse failure: opcode OP_DATA_14 requires 15 bytes, but script only has 7 remaining) class: nonstandard
	simnet	4104678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5fac 04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f OP_CHECKSIG class: pubkey
	regnet	801679e98561ada96caec2949a5d41c4cab3851eb740d951c10ecbcf265c1fd9 OP_LEFT 79e98561ada96caec2949a5d41c4cab3851eb740d951 OP_TADD [error] (parse failure: opcode OP_DATA_14 requires 15 bytes, but script only has 7 remaining) class: nonstandard

Table 14.23 Genesis block hashes (display form)

Field	Network	Value
block hash	mainnet	298e5cc3d985bfe7f81dc135f360abe089edd4396b86d2de66b0cef42b21d980
	testnet3	a649dce53918caf422e9c711c858837e08d626ecfcd198969b24f7b634a49bac
	simnet	6bef82c645999585f7255cb02672921ac2f5492820090cd635fe3a59d16b4f87
	regnet	2ced94b4ae95bba344cfa043268732d230649c640f92dce2d9518823d3057cb0
MerkleRoot	mainnet	66aa7491b9adce110585ccab7e3fb5fe280de174530cca10eba2c6c3df01c10d
	testnet3	e7dfbceac9fccd6025c70a1dfa9302b3e7b5aa22fa51c98a69164ad403d60a2c
	simnet	5c9488e2f3425de9951ee31f4ad5803c4487a8f4a2719d60c3bf2b58c5295692
	regnet	66aa7491b9adce110585ccab7e3fb5fe280de174530cca10eba2c6c3df01c10d
coinbase TxHash	mainnet	e7dfbceac9fccd6025c70a1dfa9302b3e7b5aa22fa51c98a69164ad403d60a2c
	testnet3	e7dfbceac9fccd6025c70a1dfa9302b3e7b5aa22fa51c98a69164ad403d60a2c
	simnet	a216ea043f0d481a072424af646787794c32bcefd3ed181a090319bbf8a37105
	regnet	e7dfbceac9fccd6025c70a1dfa9302b3e7b5aa22fa51c98a69164ad403d60a2c
coinbase TxHashFull	mainnet	66aa7491b9adce110585ccab7e3fb5fe280de174530cca10eba2c6c3df01c10d
	testnet3	66aa7491b9adce110585ccab7e3fb5fe280de174530cca10eba2c6c3df01c10d
	simnet	5c9488e2f3425de9951ee31f4ad5803c4487a8f4a2719d60c3bf2b58c5295692
	regnet	66aa7491b9adce110585ccab7e3fb5fe280de174530cca10eba2c6c3df01c10d

Table 14.24 Genesis Merkle root provenance and proof of work

Property	mainnet	testnet3	simnet	regnet
Merkle root field equals coinbase	TxHashFull	TxHash	TxHashFull	TxHashFull
Block hash $\leq$ target of its own `Bits`	no	no	yes	yes

Several properties of the genesis blocks are unusual and follow from their being constants rather than validated blocks.

- **The mainnet, testnet3 and regnet coinbases are the same transaction.** Its single input has the null outpoint (zero hash, index `0xffffffff`, tree 0) and the null witness values (value in −1, block height 0, block index `0xffffffff`); its signature script is the two bytes `0000`. Its single output has value 0 and version 0, and its 32-byte output script does not even parse: after `OP_LEFT`, a 22-byte push and `OP_TADD`, a 14-byte push runs past the end of the script. It is not a standard script of any kind and could never be satisfied.
- **The simnet coinbase is different.** Its signature script pushes the bytes `ffff001d`, the byte `04`, and the ASCII text "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks", the coinbase message of Bitcoin's genesis block; its output is a 65-byte uncompressed public key followed by `OP_CHECKSIG`, again with value 0. Its input witness leaves the value in and block index at zero instead of the null values.
- **Merkle root.** The Merkle root of a block with a single regular transaction is that transaction's full hash (Chapter 4). The testnet3 genesis header instead carries the transaction's prefix hash (`TxHash`); the dcrd source acknowledges this as a mistake that cannot be corrected without changing the testnet3 genesis hash, and notes that the field is not validated. Mainnet and regnet share a Merkle root because they share the coinbase.
- **Header fields.** The `Size` field is 0 although each block is larger, and the stake root is zero. On mainnet and testnet3 the block hash does not satisfy the target encoded in the block's own `Bits`; on simnet and regnet, whose targets admit about half of all hashes, it happens to.
- **Stake fields.** The testnet3 genesis header has block version 6 and stake version 6, unlike the version 1 and stake version 0 of the other networks, and its `SBits` equals `MinimumStakeDiff` as on mainnet. Simnet and regnet have `SBits` 0.

The serialized blocks below were produced by serializing the `chaincfg` genesis blocks with dcrd's `wire` package, checked by deserializing them again, and hashed with dcrd; the hashes match `GenesisHash`. The first 180 bytes of each are the block header, whose BLAKE-256 hash is the block hash (Chapter 3).

TEST VECTOR mainnet genesis block: serialized (300 bytes; the first 180 are the header)

```
0100000000000000000000000000000000000000000000000000000000000000000000000000000000dc101dfc3c6a2eb10ca0c5374e10d28feb53f
7eabcc850511ceadb99174aa660000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000ffff011b00c2eb0b0000000000000000000000000a0d7b856000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000010100000010000000000000000000000000000000000000000000000000000000000000000000000000ffff
ffff00ffffffff0100000000000000000000020801679e98561ada96caec2949a5d41c4cab3851eb740d951c10ecbcf265c1fd900000000
0000000001ffffffffffffffff00000000ffffffff02000000
```

TEST VECTOR testnet3 genesis block: serialized (300 bytes; the first 180 are the header)

```
0600000000000000000000000000000000000000000000000000000000000000000000000000000002c0ad603d44a16698ac951fa22aab5e7b30293
fa1d0ac72560cdfcc9eabcdfe7000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000ffff001e002d3101000000000000000000000000808f675b1aa4ae1800000000000000000000000000000000000000000000000000000
0000000000000000000000060000000101000000010000000000000000000000000000000000000000000000000000000000000000000000000ffff
ffff00ffffffff0100000000000000000000020801679e98561ada96caec2949a5d41c4cab3851eb740d951c10ecbcf265c1fd900000000
0000000001ffffffffffffffff00000000ffffffff02000000
```

TEST VECTOR simnet genesis block: serialized (410 bytes; the first 180 are the header)

```
0100000000000000000000000000000000000000000000000000000000000000000000000000000000925629c5582bbfc3609d71a2f4a887443c80d5
4a1fe31e95e95d42f3e288945c0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000ffff7f2000000000000000000000000000000000004506865300000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000010100000010000000000000000000000000000000000000000000000000000000000000000000000000ffff
ffff00ffffffff0100000000000000000000434104678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6
bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5fac0000000000000000010000000000000000000000000000000000
004d04ffff001d0104455468652054696d65732030332f4a616e2f323030392043686616e63656c6c6f72206f6e206272696e6b206f6620
7365636f6e64206261696c6f757420666f722062616e6b7300
```

TEST VECTOR regnet genesis block: serialized (300 bytes; the first 180 are the header)

```
0100000000000000000000000000000000000000000000000000000000000000000000000000000000dc101dfc3c6a2eb10ca0c5374e10d28feb53f
7eabcc850511ceadb99174aa66000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000ffff7f2000000000000000000000000000000000008006b45b000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000010100000001000000000000000000000000000000000000000000000000000000000000000000000000ffff
ffff00ffffffff0100000000000000000000020801679e98561ada96caec2949a5d41c4cab3851eb740d951c10ecbcf265c1fd900000000
0000000001ffffffffffffffff00000000ffffffff02000000
```

14.7 Block-One Ledger

The coinbase of the block at height 1 does not pay the ordinary subsidy; instead it must pay exactly the outputs listed in `BlockOneLedger` (the premine on mainnet). Table 14.25 gives the number of outputs and their total for each network. The outputs themselves, and the rules that enforce them, are given in Chapter 11.

Table 14.25 Block-one ledger summary

Network	Outputs	Total (atoms)	Total	Script types
mainnet	3146	168000000000000	1680000 DCR	all P2PKH (secp256k1 ECDSA), script version 0
testnet3	2	10000000000000	100000 DCR	all P2PKH (secp256k1 ECDSA), script version 0
simnet	3	30000000000000	300000 DCR	all P2PKH (secp256k1 ECDSA), script version 0
regnet	3	30000000000000	300000 DCR	all P2PKH (secp256k1 ECDSA), script version 0

PART IV

Network

How nodes find each other, exchange blocks and transactions, serve light clients, decide what to relay and mine, and mix coins.

CHAPTER 15

The Peer-to-Peer Protocol

Decred nodes exchange blocks, transactions, addresses and a few kinds of ephemeral state over a peer-to-peer network of TCP connections. This chapter specifies that protocol: the framing of every message, the encoding of every message type, the protocol versions and how two peers agree on one, the connection handshake, how a node synchronizes the chain from its peers and relays new data, and how the reference implementation manages connections and punishes misbehaviour.

The wire protocol descends from Bitcoin's, and many command names are the same, but the two are not interoperable and differ in ways that are easy to miss: the message checksum uses BLAKE-256 rather than double SHA-256, unknown commands are a bannable offence rather than being ignored, blocks are announced by header, and there are Decred-specific messages for the stake system, compact filters and coin mixing. Nothing in this chapter should be inferred from Bitcoin's behaviour.

15.1 Normative Status

The peer-to-peer protocol is not part of consensus. Whether a block is valid never depends on how it was received, and two nodes that implement the messages in this chapter differently will still agree on the chain as long as they can exchange blocks. For that reason this chapter contains no rule boxes: the numbered rules of this document are reserved for block validity.

The protocol is nevertheless a wire format, and a node that encodes a message incorrectly, or sends one that its peer does not expect, will be disconnected and usually banned by every node running the reference implementation. The requirements in this chapter therefore come in two strengths:

- **Interoperability requirements**, written with **MUST** and **MUST NOT**, describe what a peer has to do so that dcrd will talk to it: byte layouts, size limits, version gating and message ordering. Violating one of them causes dcrd to disconnect the peer and, in most cases, to ban its IP address for a day (Section 15.9.3).
- **Implementation behaviour**, described in ordinary prose and labelled as policy, records what dcrd itself chooses to do: how many peers it connects to, which peer it syncs from, when it requests data, how long it waits. An independent implementation is free to choose differently, but it should understand these choices because its peers will apply them to it.

Where dcrd tolerates input that it would never produce itself, the text says so, since an implementation that wishes to interoperate with older or unusual peers needs to know the exact boundaries.

15.2 Transport

Peers communicate over TCP. A connection carries a sequence of messages in each direction; there is no other framing, no encryption and no authentication at the protocol level. Connections to Tor onion services and connections through SOCKS5 proxies are supported by the reference implementation as a transport detail and do not change the protocol.

The party that opens the TCP connection is the **outbound** side and the party that accepts it is the **inbound** side. The distinction matters for the handshake (Section 15.6) and for several policies, because the addresses a

node connects to itself are chosen by it, while inbound peers are chosen by others and are treated with more suspicion.

Each network has a default TCP port and a four-byte **network magic** that begins every message. A message whose magic does not match the local network is rejected (Section 15.3). The values are listed in Table 15.1; the parameter tables of Chapter 14 give them as `DefaultPort` and `Net`.

Table 15.1 Network identifiers. The magic is a `uint32` serialized little-endian, so the bytes on the wire are the reverse of the hexadecimal value.

Network	Magic (`uint32`)	Bytes on the wire	Default port
Main network	`0xd9b400f9`	`f9 00 b4 d9`	9108
Test network 3	`0xb194aa75`	`75 aa 94 b1`	19108
Simulation network	`0x12141c16`	`16 1c 14 12`	18555
Regression network	`0xdab500fa`	`fa 00 b5 da`	18655

The simulation and regression networks are meant for private testing. On those networks dcrd neither answers nor processes address messages and never contacts seeders, so that they cannot grow into public networks.

15.3 The Message Envelope

Every message consists of a fixed 24-byte header followed by a payload whose length the header gives. The header layout is in Table 15.2.

Table 15.2 Message header (`MessageHeaderSize` = 24 bytes)

Field	Type	Bytes	Description
`magic`	`uint32`	4	Network magic from Table 15.1, little-endian.
`command`	`char[12]`	12	ASCII command name identifying the message type, padded on the right with `0x00` bytes to 12 bytes (`CommandSize`).
`length`	`uint32`	4	Length of the payload in bytes, little-endian.
`checksum`	`byte[4]`	4	The first four bytes of BLAKE-256 of the payload.

The checksum is computed as

$$\text{checksum} = \text{BLAKE256}(\text{payload})[0:4], \tag{15.1}$$

a single application of BLAKE-256 (Chapter 3) over exactly the `length` payload bytes, keeping the first four bytes of the digest in the order the hash function outputs them. An empty payload therefore has checksum `716f6e86`, the first four bytes of BLAKE-256 of the empty string.

> **IMPLEMENTATION PITFALL**
> Bitcoin uses the first four bytes of double SHA-256 here. Decred uses *single* BLAKE-256. It is also not the hash used for block or transaction identifiers in any particular serialization; it is simply BLAKE-256 of the raw payload bytes.

15.3.1 Command Strings

A command name is between 1 and 12 bytes of printable ASCII. When sending, the name is copied into the 12-byte field and the remainder is filled with zero bytes. When receiving, dcrd strips trailing zero bytes and requires every remaining byte to be in the **strict ASCII** range `0x20` to `0x7e` inclusive. A command that contains a zero byte followed by a non-zero byte is therefore malformed. Commands are case-sensitive and compared exactly; there is no normalization. All command strings defined at the reference commit are listed in Table 15.12.

15.3.2 Size Limits

Two limits bound a payload. The global limit `MaxMessagePayload` is $32 \times 1024 \times 1024 = 33554432$ bytes. In addition, each message type has its own maximum, which may depend on the negotiated protocol version; the values at the current protocol version are in Table 15.12, and a value of zero means that the message has no payload or is not valid at that version. A sender **MUST NOT** send a payload that exceeds either limit, and the receiver checks both *before* reading the payload, using only the length in the header, so that a peer cannot make it allocate memory for an oversized message.

15.3.3 Receiving a Message

Algorithm 15.1 gives the complete sequence of checks dcrd applies when it reads a message. The order matters for interoperability in one respect: the checks in steps 1 to 5 happen before the payload is read, so after any of those failures the stream position is unknown and the connection cannot continue.

ALGORITHM 15.1 Reading one message from a connection

```
MAX_MESSAGE_PAYLOAD = 32 * 1024 * 1024

def read_message(stream, pver: uint32, net: uint32):
    hdr = stream.read_exact(24)               # EOF here: connection closed
    magic    = le32_decode(hdr[0:4])
    command  = hdr[4:16]
    length   = le32_decode(hdr[16:20])
    checksum = hdr[20:24]

    if length > MAX_MESSAGE_PAYLOAD:                    # 1
        fail(ErrPayloadTooLarge)
    if magic != net:                                    # 2
        fail(ErrWrongNetwork)
    name = command.rstrip(b"\x00")
    if any(b < 0x20 or b > 0x7e for b in name):          # 3
        fail(ErrMalformedCmd)
    if name not in KNOWN_COMMANDS:                      # 4 (empty name included)
        fail(ErrUnknownCmd)
    if length > max_payload_length(name, pver):         # 5
        fail(ErrPayloadTooLarge)

    payload = stream.read_exact(length)                 # short read: I/O error
    if blake256(payload)[0:4] != checksum:              # 6
        fail(ErrPayloadChecksum)
    msg, consumed = decode(name, payload, pver)         # 7: per-message rules
    if consumed != length:                              # 8
        fail(ErrTrailingBytes)
    return msg
```

dcrd: wire/message.go · ReadMessageN

Every failure labelled with an `Err...` name in Algorithm 15.1, and every decoding failure of the same kind inside step 7 (an oversized count, a non-canonical variable-length integer, a non-ASCII string, a message type that is not valid at the negotiated version, and so on), is a **wire protocol violation**. dcrd responds to a wire protocol violation by banning the remote peer and closing the connection (Section 15.9.3). A payload that ends before the decoder has read every field it needs is an ordinary I/O error: the connection is closed but the peer is not banned. A read that waits longer than the idle timeout also closes the connection (Section 15.9.2).

The payload must be consumed exactly: trailing bytes after the last field are a violation, even in the `version` message whose optional fields would otherwise suggest that extra data is tolerated.

IMPLEMENTATION PITFALL

Unknown commands are fatal. Bitcoin nodes ignore messages with commands they do not recognize. dcrd treats an unknown command as a wire protocol violation and bans the sender. This includes `reject`,

which dcrd's message reader no longer recognizes at any protocol version (Section 15.7.16). Likewise, a known message that is not valid at the negotiated protocol version (for example `addr` after both peers have negotiated version 12) causes a ban. A node **MUST NOT** send any message type that the negotiated protocol version does not define.

TEST VECTOR Envelope of a payload-less message (`verack`, main network)

```
f9 00 b4 d9                                     magic 0xd9b400f9
76 65 72 61 63 6b 00 00 00 00 00 00             "verack" + six 0x00
00 00 00 00                                     length 0
71 6f 6e 86                                     BLAKE-256("")[0:4]
```

TEST VECTOR Envelope of `ping` with nonce `0x1122334455667788` (main network)

```
f9 00 b4 d9                                     magic
70 69 6e 67 00 00 00 00 00 00 00 00             "ping"
08 00 00 00                                     length 8
ec 16 34 63                                     checksum
88 77 66 55 44 33 22 11                         payload: nonce (uint64 LE)
```

15.4 Common Encodings

Integers in payloads are little-endian unless stated otherwise. The variable-length integer (`varint`), length-prefixed byte string (`varbytes`) and the block header and transaction serializations are defined in Chapter 4; in particular, a varint that could have been encoded in fewer bytes is rejected as non-canonical, which here is a wire protocol violation. This section defines the remaining building blocks that several messages share.

15.4.1 Strings

A `varstring` is a varint length followed by that many bytes. Where a field is described as **strict ASCII**, every byte of the string **MUST** lie in `0x20`–`0x7e` this applies to the user agent of `version`, the requested types of `getinitstate`, and the command and reason of the obsolete `reject` message. A varstring length larger than `MaxMessagePayload` is rejected before any allocation, and individual fields impose smaller limits that are stated with each message.

15.4.2 Timestamps

Three timestamp encodings occur, all counting seconds since the Unix epoch:

- `uint32` timestamps, in the legacy network address (`addr` message).
- `int64` timestamps, in the `version` message.
- `uint64` timestamps, in the version 2 network address (`addrv2` message).

For the two 64-bit encodings the receiver reads the eight bytes as an unsigned integer t and rejects the message (a wire protocol violation) when $t > 2^{63} - 1 - 62135596800$, the largest value that still fits a signed 64-bit count of seconds after shifting the epoch to the year 1. In effect, negative `int64` timestamps are also rejected.

15.4.3 Legacy Network Address

The legacy network address, used by `addr` and by the two address fields of `version`, can only describe IPv4 and IPv6 hosts. It has a timestamped form (30 bytes) used in `addr`, and an untimestamped form (26 bytes) used in `version`.

Table 15.3 Legacy network address (`NetAddress`)

Field	Type	Bytes	Description
`timestamp`	`uint32`	4	Last time the address was seen. Present only in the timestamped form.
`services`	`uint64`	8	Service flags of the host (Table 15.9).
`ip`	`byte[16]`	16	IPv6 address in network byte order. An IPv4 address *a.b.c.d* is written as the IPv4-mapped IPv6 address `::ffff:a.b.c.d`, that is ten `0x00` bytes, two `0xff` bytes and the four IPv4 bytes.
`port`	`uint16`	2	TCP port, **big-endian**.

IMPLEMENTATION PITFALL
The port of the legacy network address is big-endian while every other integer in the protocol is little-endian, and the port of the version 2 address (Table 15.4) is little-endian. The legacy timestamp is a `uint32` and wraps in the year 2106.

15.4.4 Version 2 Network Address

Protocol version 12 introduced a variable-length network address that also carries Tor version 3 onion services. A one-byte type selects the length of the encoded address.

Table 15.4 Version 2 network address (`NetAddressV2`)

Field	Type	Bytes	Description
`timestamp`	`uint64`	8	Last time the address was seen, subject to the range check above.
`services`	`uint64`	8	Service flags of the host.
`type`	`uint8`	1	Address type from Table 15.5.
`addr`	`byte[n]`	4/16/32	Encoded address; its length n is fixed by the type.
`port`	`uint16`	2	TCP port, **little-endian**.

Table 15.5 Version 2 network address types

Value	Name	Length	Encoded address
0	Unknown	–	Not encodable. Receiving it is a wire protocol violation (`ErrUnknownNetAddrType`).
1	IPv4	4	The four IPv4 address bytes in network order.
2	IPv6	16	The sixteen IPv6 address bytes in network order. **MUST NOT** be an IPv4-mapped address (see below).
3	Tor v3	32	The 32-byte Ed25519 public key of the onion service. The `.onion` name is the lowercase base32 encoding of `pubkey || checksum || 0x03`, where `checksum` is the first two bytes of SHA3-256 of the ASCII string `.onion checksum`, the public key and the byte `0x03`.
4–255	–	–	Reserved. Receiving any of them is a wire protocol violation.

The type and the bytes have to agree in canonical form. When dcrd converts a received address it reduces an address of type IPv6 that is an IPv4-mapped address to its four-byte form and then finds that the derived type, IPv4, does not match the claimed type; it bans the sender (Section 15.7.3).

15.4.5 Inventory Vectors

An **inventory vector** names one object a peer has, wants or does not have. It is 36 bytes.

Table 15.6 Inventory vector (`InvVect`)

Field	Type	Bytes	Description
`type`	`uint32`	4	Object type from Table 15.7.
`hash`	`byte[32]`	32	Identifier of the object in internal byte order (not the reversed display form).

Table 15.7 Inventory types

Value	Name	Meaning of the hash
0	`ERROR`	Unused.
1	`MSG_TX`	Transaction hash (Chapter 4). Used for every kind of transaction, including tickets, votes, revocations and treasury transactions.
2	`MSG_BLOCK`	Block hash.
3	`MSG_FILTERED_BLOCK`	Defined but unused; dcrd never sends or serves it.
4	`MSG_MIX`	Hash of a mixing message (Chapter 18). Valid from protocol version 10.

The wire decoder does not restrict the type field; the restrictions are applied by the message handlers and are described with `inv`, `getdata` and `notfound` in Section 15.7.

15.4.6 Block Locators

A **block locator** is a list of block hashes that lets a peer find the most recent block two nodes have in common without either knowing the other's chain. It starts at a chosen block and walks back towards the genesis block, densely at first and then with exponentially growing gaps, so its length grows only logarithmically with the height. The `getheaders` and `getblocks` messages carry a locator and a **stop hash**. A locator **MUST NOT** contain more than `MaxBlockLocatorsPerMsg` $= 500$ hashes, a bound that the construction below never approaches.

Algorithm 15.2 gives dcrd's construction. The first entry is the block itself; the next eleven are its eleven immediate ancestors; after that the distance between entries doubles each time; and the genesis block is always the last entry. When the starting block is on a side chain, the walk follows that side chain's ancestors.

ALGORITHM 15.2 **Block locator construction**

```
def block_locator(node) -> list[Hash]:
    locator = []
    step = 1
    while True:
        locator.append(node.hash)
        if node.height == 0:            # genesis reached
            break
        height = max(node.height - step, 0)
        node = ancestor(node, height)   # ancestor on node's own branch
        if len(locator) > 10:           # after 11 entries, double the step
            step *= 2
    return locator
```

dcrd: internal/blockchain/chainview.go · blockLocator

TEST VECTOR Heights of the blocks in a locator, by starting height

```
start 0        : [0]
start 5        : [5 4 3 2 1 0]
start 12       : [12 11 10 9 8 7 6 5 4 3 2 1 0]
start 13       : [13 12 11 10 9 8 7 6 5 4 3 2 0]
start 1000000  : [1000000 999999 999998 999997 999996 999995 999994 999993
                  999992 999991 999990 999989 999987 999983 999975 999959
                  999927 999863 999735 999479 998967 997943 995895 991799
                  983607 967223 934455 868919 737847 475703 0]   (31 entries)
```

The responder interprets a locator against its own best chain with Algorithm 15.3. Two special cases are part of the protocol: an empty locator is a request for the single block named by the stop hash, and a locator with no hash on the responder's best chain is answered from the block after genesis. A stop hash of all zeros, or

any hash not on the best chain at or after the starting point, imposes no limit other than the per-message maximum.

ALGORITHM 15.3 Resolving a locator and stop hash

```
def locate(locator: list[Hash], stop: Hash, max_entries: int) -> list[Node]:
    stop_node = lookup(stop)                # any known header, or None
    if len(locator) == 0:
        return [] if stop_node is None else [stop_node]

    start = genesis
    for h in locator:                       # first hash on the best chain wins
        n = lookup(h)
        if n is not None and best_chain_contains(n):
            start = n
            break
    start = best_chain_next(start)          # block after the common one
    if start is None:                       # common block is the tip
        return []

    total = tip.height - start.height + 1
    if (stop_node is not None and best_chain_contains(stop_node)
            and stop_node.height >= start.height):
        total = stop_node.height - start.height + 1
    total = min(total, max_entries)
    return [best_chain_at(start.height + i) for i in range(total)]
```

dcrd: internal/blockchain/chain.go · locateInventory

> **NOTE**
> With an empty locator the stop block is returned even if it is not on the best chain, because only the lookup, not best-chain membership, is checked in that branch.

15.5 Protocol Versions and Services

15.5.1 Version History

The protocol version is a small integer that each peer advertises in its `version` message. It is incremented whenever a message is added, removed or changed. Table 15.8 lists every version defined by the reference implementation and what it introduced.

Table 15.8 Protocol versions

Version	dcrd name	Change
1	`InitialProcotolVersion`	Initial protocol. (The name is misspelled in dcrd.)
2	`NodeBloomVersion`	Adds the `SFNodeBloom` service flag. Bloom filtering was never implemented; the flag is unused.
3	`SendHeadersVersion`	Adds `sendheaders`.
4	`MaxBlockSizeVersion`	Raises the maximum `block` and `tx` payload from 1000000 to 1310720 bytes (`MaxBlockPayloadV3` to `MaxBlockPayload`).
5	`FeeFilterVersion`	Adds `feefilter`.
6	`NodeCFVersion`	Adds the `SFNodeCF` service flag and the version 1 committed filter messages `getcfilter`, `cfilter`, `getcfheaders`, `cfheaders`, `getcftypes`, `cftypes`.
7	`CFilterV2Version`	Adds `getcfilterv2` and `cfilterv2`; the version 1 filter messages are deprecated from this version.
8	`InitStateVersion`	Adds `getinitstate` and `initstate`, which replace `getminings` and `miningstate`.
9	`RemoveRejectVersion`	Removes `reject`.

Version	dcrd name	Change
10	`MixVersion`	Adds the eight mixing messages and inventory type `MSG_MIX`.
11	`BatchedCFiltersV2Version`	Adds `getcfsv2` and `cfiltersv2`.
12	`AddrV2Version`	Adds `addrv2` and removes `addr`. The handshake becomes strict: `verack` must immediately follow `version`.

The current version, `ProtocolVersion` in the wire package and the maximum advertised by dcrd, is **12**. The **minimum version dcrd accepts** from a peer is **9** (`RemoveRejectVersion`); a peer advertising anything lower is disconnected during the handshake. Versions 1 to 8 therefore matter only for reading the history; in particular, `reject`, `getminings` and `miningstate` can never be validly exchanged with a current dcrd node. (dcrd's message reader does not recognize `reject` at all, at any version, so it is treated as an unknown command.)

15.5.2 Negotiation

Each peer starts with its own maximum version as its working version. When it receives the remote `version` message, it sets

$$\text{negotiated} = \min(\text{local maximum}, \text{remote advertised}), \tag{15.2}$$

and from that moment encodes and decodes every message at the negotiated version. Both sides arrive at the same value because each computes the minimum of the same two numbers. The one subtlety is the inbound side: dcrd reads the remote `version` before writing its own, so the version an inbound dcrd advertises is already the negotiated value rather than its maximum (Section 15.6). An implementation **MUST** therefore not infer a peer's maximum version from an inbound peer's `version` message.

The negotiated version determines, for each message type, whether it may be sent at all and, for a few types, its maximum size. The gating is summarized in the "Versions" column of Table 15.12.

15.5.3 Service Flags

The `services` field of `version` and of each network address is a 64-bit bit field advertising optional capabilities.

Table 15.9 Service flags

Bit	Value	Name	Meaning
0	`0x1`	`SFNodeNetwork`	Full node: stores the full chain and serves blocks, headers and filters.
1	`0x2`	`SFNodeBloom`	Bloom filtering (never implemented; unused).
2	`0x4`	`SFNodeCF`	Version 1 committed filters (obsolete).

dcrd advertises exactly `SFNodeNetwork`. It requires `SFNodeNetwork` of every outbound peer and asks HTTPS seeders only for such peers; it does not require any service of inbound peers. A peer without `SFNodeNetwork` is never chosen to sync from and is skipped when relaying the early announcement of a new tip block (Section 15.8.6). Unknown service bits are carried and stored but otherwise ignored.

15.6 The Connection Handshake

Before anything else, the two peers exchange `version` messages and acknowledge each other's with `verack`. The handshake fixes the negotiated protocol version and tells each side the other's services, best block height, user agent and whether it wants transaction announcements.

15.6.1 The `version` Message

Table 15.10 `version` payload (maximum 358 bytes)

Field	Type	Bytes	Description
`version`	`int32`	4	Protocol version of the sender: its maximum for an outbound sender, the negotiated value for an inbound dcrd sender.
`services`	`uint64`	8	Service flags of the sender.
`timestamp`	`int64`	8	Sender's current Unix time in seconds.
`addr_you`	`NetAddress`	26	Address of the receiver as seen by the sender, untimestamped form (Table 15.3).
`addr_me`	`NetAddress`	26	Address of the sender, untimestamped form. Optional on receipt.
`nonce`	`uint64`	8	Random value used to detect connections to self. Optional on receipt.
`user_agent`	`varstring`	1+	Software identification, strict ASCII, at most `MaxUserAgentLen` = 256 bytes. Optional on receipt.
`last_block`	`int32`	4	Height of the sender's best block. Optional on receipt.
`relay`	`bool`	1	`0x01` if the sender wants transaction and mix announcements, `0x00` if not. Any non-zero byte reads as true. Optional on receipt.

The maximum payload is $4 + 8 + 8 + 2 \times 30 + 8 + 9 + 256 + 4 + 1 = 358$ bytes (the limit uses the 30-byte timestamped address size and the 9-byte maximum varint, so it exceeds the largest encodable message, 344 bytes).

Decoding. The first four fields are mandatory. Each of the remaining fields, in order, is decoded only if at least one byte of the payload remains, and a field that is absent takes the value zero (empty user agent, `relay` true). Once a field has started it must be complete: a payload that ends inside a field is an I/O error. After the last field nothing may remain. A user agent that is longer than 256 bytes or contains a byte outside `0x20` – `0x7e` is a wire protocol violation. The receiver ignores `addr_me` and the services in `addr_you`; it uses only the IP address and port of `addr_you` for discovering its own external address (Section 15.9.6).

A sender **MUST** include all nine fields. The optional-field decoding exists only for compatibility with very old software.

What dcrd sends. dcrd fills the fields as follows:

- `version`: 12 when outbound; the negotiated version when inbound.
- `services`: `SFNodeNetwork`.
- `timestamp`: the local clock, whole seconds.
- `addr_you`: the remote IP address and port of the connection with the remote peer's services (zero for outbound connections, whose services are not yet known). When dcrd connects through a proxy and the address would otherwise reveal the proxy, it sends the IPv4 address `0.0.0.0` and port 0 instead. For an outbound connection to a Tor onion service, which has no IP address, the 16 IP bytes are all zero.
- `addr_me`: services `SFNodeNetwork`, IP address all zeros, port 0. The local address is advertised separately with an address message, because a connection from a node does not prove that the node accepts connections.
- `nonce`: a fresh random 64-bit value per connection. dcrd remembers the 50 most recent nonces it has sent.
- `user_agent`: a sequence of `/name:version/` components, oldest layer first. The wire library contributes `/dcrwire:1.0.0/` and dcrd appends `dcrd:MAJOR.MINOR.PATCH/`, adding `(comment; comment)` before the closing slash for a pre-release tag. At the reference commit the string is `/dcrwire:1.0.0/dcrd:2.2.0(pre)/`. The format is a convention; receivers do not parse it.
- `last_block`: the height of the local best chain tip.
- `relay`: false only when dcrd runs in blocks-only mode, true otherwise.

15.6.2 Validation of the Remote `version`

When dcrd receives the first message on a connection it applies the following checks in order. Any failure aborts the handshake and closes the connection; none of them leads to a ban unless the message was also a wire protocol violation.

1. The message **MUST** be a `version` message.
2. Its nonce **MUST NOT** be one of the 50 nonces most recently sent by the local node; otherwise the connection is to the node itself.
3. The negotiated version is set as in Section 15.5.2, and the remote services, best height, user agent and clock offset are recorded.
4. For outbound connections on the main and test networks, the services recorded for the address in the address manager are updated. This happens even if the peer is rejected below.
5. The advertised version **MUST** be at least 9.
6. **Policy, outbound only.** dcrd wants at least $\min(3, T)$ of its outbound peers to support mixing (version 10 or later), where T is the target number of outbound peers (8 by default). It rejects an outbound peer with a version below 10 when it has fewer such peers than that and accepting the peer would leave too few free outbound slots to reach the quota, that is when $\text{outbound} + \min(3, T) \geq T$.
7. **Policy, outbound only.** The peer **MUST** advertise `SFNodeNetwork`.

If all checks pass, the receiver records `addr_you` as a candidate for its own external address, notes whether the peer disabled relay, and adds the remote `timestamp` to the network time samples (Section 15.6.5).

15.6.3 Message Order

The outbound side speaks first. The full sequence, as seen on the wire, is in Table 15.11.

Table 15.11 Handshake order. A peer sends `verack` only after it has accepted the other side's `version`.

Step	Direction	Message and condition
1	outbound → inbound	`version` (outbound peer's maximum version).
2	inbound → outbound	`version` (negotiated version), sent after the inbound side validated step 1.
3	inbound → outbound	`verack`, sent immediately after step 2.
4	outbound → inbound	`verack`, sent after the outbound side validated step 2 and received step 3.

The rules each side enforces are:

- The first message received **MUST** be `version`.
- At a negotiated version of 12, the second message received **MUST** be `verack`. At negotiated versions 9 to 11, dcrd reads up to three messages looking for the `verack`; up to two other messages may precede it, and they are buffered and processed, in order, after the handshake completes. If the third message is not `verack` the handshake fails.
- The whole handshake **MUST** complete within 30 seconds of its start (`negotiateTimeout`); each individual read is also subject to the idle timeout.
- After the handshake, a second `version` or a second `verack` causes an immediate disconnect.

A node that receives a `version` it does not accept simply closes the connection without sending `verack` or any explanation. There is no negative-acknowledgement message.

15.6.4 After the Handshake

As soon as the handshake succeeds, dcrd sends or schedules the following, which an implementation should expect to receive right away:

1. `sendheaders`, to every peer, asking for block announcements as headers.
2. For outbound peers on the main and test networks: an `addr` or `addrv2` (by negotiated version) containing the local node's best external address, if the node accepts inbound connections, believes

it is synced, and the address is routable; then `getaddr` if the address manager holds fewer than 1000 addresses.
3. `getheaders`, to discover the peer's best chain (Section 15.8). During the initial header sync it is sent only to the chosen sync peer; afterwards it is sent to every new peer that advertises `SFNodeNetwork`.
4. `getinitstate` (at version 8 or later), once the local node believes it is synced, to learn the current tip candidates, their votes and the treasury spends in the remote mempool (Section 15.7.17).
5. `ping` every 107 seconds thereafter.

15.6.5 Network-Adjusted Time

The `timestamp` of each accepted `version` message is a sample of the remote peer's clock. dcrd combines these samples into an offset that is added to the local clock to form the **network-adjusted time**, which the block validation code uses when it compares header timestamps against the present (Chapter 10). The procedure, Algorithm 15.4, deliberately reproduces a quirk of Bitcoin Core: the offset is recomputed only when the number of samples is odd, and the sample limit, 200, is even, so once 200 samples have been collected the offset never changes again for the life of the process.

ALGORITHM 15.4 **Adding a time sample**

```
MAX_ENTRIES = 200
MAX_OFFSET  = 70 * 60          # seconds
offsets = []                   # in arrival order
known_sources = set()
offset_secs = 0                # current adjustment

def add_time_sample(source_id: str, remote_time: int):
    global offset_secs
    if source_id in known_sources:       # one sample per remote "host:port"
        return
    known_sources.add(source_id)
    offset = tdiv_seconds(remote_time - now_seconds())   # truncated to seconds
    if len(offsets) == MAX_ENTRIES:
        offsets.pop(0)                   # drop the oldest
    offsets.append(offset)
    n = len(offsets)
    if n < 5 or n % 2 != 1:
        return
    median = sorted(offsets)[n // 2]
    offset_secs = median if abs(median) < MAX_OFFSET else 0

def adjusted_time() -> int:
    return now_seconds() + offset_secs
```

dcrd: internal/blockchain/mediantime.go · AddTimeSample

15.6.6 Worked Example

The following `version` message was produced by dcrd's wire package. The sender advertises protocol version 12 and `SFNodeNetwork`, its clock reads 2025-01-01T00:00:00Z (`0x67748580`), it sees the receiver at `192.0.2.10` port 9108, its nonce is `0x0123456789abcdef`, its user agent is that of dcrd at the reference commit, its best height is 1000000 and it wants transaction announcements.

TEST VECTOR `version` payload (117 bytes)

```
0c 00 00 00                                      version = 12
01 00 00 00 00 00 00 00                          services = SFNodeNetwork
80 85 74 67 00 00 00 00                          timestamp = 1735689600
00 00 00 00 00 00 00 00                          addr_you.services = 0
00 00 00 00 00 00 00 00 00 00 ff ff c0 00 02 0a  addr_you.ip = ::ffff:192.0.2.10
```

```
23 94                                           addr_you.port = 9108 (big-endian)
01 00 00 00 00 00 00 00                         addr_me.services = SFNodeNetwork
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 addr_me.ip = ::
00 00                                           addr_me.port = 0
ef cd ab 89 67 45 23 01                         nonce = 0x0123456789abcdef
1f                                              user_agent length = 31
2f 64 63 72 77 69 72 65 3a 31 2e 30 2e 30 2f    "/dcrwire:1.0.0/"
64 63 72 64 3a 32 2e 32 2e 30 28 70 72 65 29 2f "dcrd:2.2.0(pre)/"
40 42 0f 00                                     last_block = 1000000
01                                              relay = true
```

TEST VECTOR `version` header (main network) and checksum

```
BLAKE-256(payload) = 588c8c560b1ab70e42bfad8f39b53db8d1dad0004115631aec8156d73b52a4dd

f9 00 b4 d9                                     magic
76 65 72 73 69 6f 6e 00 00 00 00 00             "version"
75 00 00 00                                     length = 117
58 8c 8c 56                                     checksum = digest[0:4]
```

The complete 141-byte message is the header followed by the payload above.

15.7 Message Reference

Table 15.12 lists every command defined at the reference commit. The “Versions” column gives the range of negotiated protocol versions at which the wire layer accepts the message; outside that range sending it is a wire protocol violation. The “Max payload” column is the per-type limit at version 12, or at the last version at which the message is valid.

Table 15.12 Message types. “Sent” and “Handled” describe dcrd’s behaviour: whether it ever originates the message, and what it does on receipt.

Command	Versions	Max payload	Sent	Handled on receipt
`version`	all	358	yes	Handshake only; later: disconnect.
`verack`	all	0	yes	Handshake only; later: disconnect.
`getaddr`	all	0	yes	Answered once, inbound peers only.
`addr`	1–11	30003	yes	Stored in the address manager.
`addrv2`	12–	51003	yes	Stored in the address manager.
`inv`	all	1800003	yes	Requests needed transactions and mix messages.
`getdata`	all	1800003	yes	Served asynchronously.
`notfound`	all	1800003	yes	Clears requests; ban score.
`getblocks`	all	16039	no	Answered with `inv` (legacy).
`getheaders`	all	16039	yes	Answered with `headers`.
`headers`	all	362003	yes	Header sync and announcements.
`block`	all	1310720	yes	Processed only if requested.
`tx`	all	1310720	yes	Processed (also if unsolicited).
`mempool`	all	0	no	Answered with `inv`; ban score.
`ping`	all	8	yes	Answered with `pong`.
`pong`	all	8	no[a]	Updates round-trip statistics.
`sendheaders`	3–	0	yes	Switches announcements to headers.
`feefilter`	5–	8	no	Ignored.

Command	Versions	Max payload	Sent	Handled on receipt
`reject`	none[c]	–	no	Unknown command: ban.
`getminings`	all[b]	0	no	Ban.
`miningstate`	all[b]	1546	no	Ban.
`getinitstate`	8–	1057	yes	Answered once with `initstate`.
`initstate`	8–	1763	yes	Requests the advertised data, once.
`getcfilterv2`	7–	32	no	Answered with `cfilterv2`.
`cfilterv2`	7–	263210	yes	Ignored (dcrd never requests it).
`getcfsv2`	11–	64	no	Answered with `cfiltersv2`.
`cfiltersv2`	11–	26321001	yes	Ignored (dcrd never requests it).
`getcfilter`	6–[b]	33	no	Ban.
`cfilter`	6–[b]	262182	no	Ignored.
`getcfheaders`	6–[b]	16036	no	Ban.
`cfheaders`	6–[b]	64036	no	Ignored.
`getcftypes`	6–[b]	0	no	Ban.
`cftypes`	6–[b]	259	no	Ignored.
`mixpairreq`	10–	8476848	yes	Chapter 18.
`mixkeyxchg`	10–	17815	yes	Chapter 18.
`mixcphrtxt`	10–	552584	yes	Chapter 18.
`mixslotres`	10–	17318030	yes	Chapter 18.
`mixfactpoly`	10–	49291	yes	Chapter 18.
`mixdcnet`	10–	20988047	yes	Chapter 18.
`mixconfirm`	10–	1016520	yes	Chapter 18.
`mixsecrets`	10–	70831	yes	Chapter 18.

[a] dcrd sends `pong` only in reply to `ping`. [b] Accepted by the wire layer at these versions but not acted on by dcrd at any version it negotiates: `getminings`, `miningstate` and the three version 1 filter requests are answered with a ban (a plain disconnect if banning is disabled), and the three version 1 filter replies are ignored; see the individual entries. [c] Defined for versions 1–8 (maximum payload `MaxMessagePayload`), but dcrd's message reader has no decoder for it and reports an unknown command at every version.

The remainder of this section specifies each message. Unless a message is described as having no payload, its payload is exactly the fields listed, in order.

15.7.1 `getaddr`

Requests known peer addresses. No payload.

dcrd sends `getaddr` to outbound peers after the handshake when its address manager holds fewer than 1000 addresses. It answers a `getaddr` only if the request comes from an inbound peer, only the first time on that connection, and never on the simulation and regression networks; otherwise it silently ignores it. The answer is an `addr` (versions 9 to 11) or `addrv2` (version 12) message described in Section 15.9.5. A requester should not expect a reply.

15.7.2 `addr`

Announces peer addresses using the legacy encoding. Valid only at negotiated versions below 12.

Table 15.13 `addr` payload (maximum 30003 bytes)

Field	Type	Bytes	Description
`count`	`varint`	1-3	Number of addresses, at most `MaxAddrPerMsg` = 1000.
`addresses`	`NetAddress[count]`	30 each	Timestamped legacy addresses (Table 15.3).

An `addr` with zero addresses decodes successfully but dcrd bans its sender. Receipt is described in Section 15.9.5.

15.7.3 `addrv2`

Announces peer addresses using the version 2 encoding. Valid only at negotiated version 12 or later.

Table 15.14 `addrv2` payload (maximum 51003 bytes)

Field	Type	Bytes	Description
`count`	`varint`	1-3	Number of addresses, from 1 to `MaxAddrPerV2Msg` = 1000. Zero is a wire protocol violation.
`addresses`	`NetAddressV2[count]`	21-51 each	Version 2 addresses (Table 15.4).

The per-type limit assumes the largest address (51 bytes, Tor v3). On receipt, every address is converted to canonical form; if the type does not match the canonical form of the bytes (an IPv4-mapped address sent as type IPv6) the whole message is discarded and the sender is banned.

TEST VECTOR `addrv2` payload with an IPv4 and an IPv6 address

```
02                                                  count = 2
80 85 74 67 00 00 00 00                             timestamp = 1735689600 (uint64)
01 00 00 00 00 00 00 00                             services = SFNodeNetwork
01                                                  type = IPv4
c0 00 02 0a                                         192.0.2.10
94 23                                               port = 9108 (little-endian)
80 85 74 67 00 00 00 00                             timestamp
01 00 00 00 00 00 00 00                             services
02                                                  type = IPv6
20 01 0d b8 00 00 00 00 00 00 00 00 00 00 00 01     2001:db8::1
94 23                                               port = 9108
```

15.7.4 `inv`

Announces objects the sender has.

Table 15.15 `inv`, `getdata` and `notfound` payload (maximum 1800003 bytes)

Field	Type	Bytes	Description
`count`	`varint`	1-3	Number of inventory vectors, at most `MaxInvPerMsg` = 50000.
`inventory`	`InvVect[count]`	36 each	Inventory vectors (Table 15.6).

dcrd announces transactions and mixing messages with `inv`. It never announces blocks with `inv` to a peer that sent `sendheaders`, which every dcrd peer does; it does still produce block inventory in answer to `getblocks`.

On receipt, an empty `inv` causes a ban. Otherwise each vector is processed by type:

- `MSG_TX`: recorded as known to the peer. If the local node believes it is synced and needs the transaction (Section 15.8.7), it is requested with `getdata`.
- `MSG_MIX`: likewise for mixing messages.
- `MSG_BLOCK`: recorded as known to the peer, and the last such vector is used to update the peer's best announced block. Nothing is requested; blocks are fetched on the basis of headers.
- Any other type: ignored.

In blocks-only mode, dcrd disconnects a peer that announces a transaction or mixing message, since it told the peer in its `version` not to.

15.7.5 `getdata`

Requests objects by inventory vector. The payload has the same layout as `inv` (Table 15.15). The responder sends, for each vector it can serve and in the order requested, a `tx`, `block` or mixing message, and then a single `notfound` listing every vector it could not serve.

dcrd serves:

- `MSG_TX` from a cache of transactions it announced in the last 45 seconds (up to 4500 entries), falling back to its mempool. Transactions already mined into blocks are not served unless they happen to still be in that cache.
- `MSG_BLOCK` for any block whose data it has, on the best chain or not.
- `MSG_MIX` from the mixing pool's recent messages.

A vector with any other type is skipped with no response at all, not even a `notfound`. An empty `getdata` causes a ban.

Limits (policy). dcrd serves `getdata` asynchronously but bounds the work a peer can queue. Each request adds $\lfloor 99n/50000 \rfloor$ to the peer's decaying ban score, where n is the number of vectors. A request for fewer than 506 objects therefore adds nothing, while requesting about 50000 objects twice within a minute crosses the ban threshold. A peer is disconnected (not banned) when accepting a new `getdata` would leave more than 1000 of its `getdata` messages, or more than 100000 individual objects, waiting to be served.

15.7.6 `notfound`

Lists requested objects the sender does not have. Same layout as `inv` (Table 15.15).

dcrd sends `notfound` as described under `getdata`. On receipt, a vector with a type other than `MSG_TX`, `MSG_BLOCK` or `MSG_MIX` causes a disconnect. For the others, the receiver removes the matching outstanding requests to that peer so the objects can be fetched elsewhere, and increases the peer's ban score: by 20 **persistent** points per block, and by 10 **decaying** points per transaction or mixing message. Because the block penalty never decays, a peer is banned at the latest when it has listed a sixth block in `notfound` messages over the life of the connection.

15.7.7 `getblocks`

Requests the hashes of best-chain blocks following a locator. This is the legacy synchronization mechanism; dcrd never sends it but still answers it.

Table 15.16 `getblocks` and `getheaders` payload (maximum 16039 bytes)

Field	Type	Bytes	Description
`version`	`uint32`	4	Protocol version of the requester. Ignored by the receiver; dcrd sends 0 in `getheaders`.
`count`	`varint`	1-3	Number of locator hashes, at most 500.
`locator`	`byte[32][count]`	32 each	Block locator (Section 15.4.6).
`stop`	`byte[32]`	32	Stop hash; all zeros for no stop.

The response is a single `inv` with up to `MaxBlocksPerMsg` $= 500$ block vectors for the blocks returned by Algorithm 15.3, omitting blocks the requester is already known to have. If no vectors remain, nothing is sent.

When the response contains exactly 500 vectors, dcrd remembers the last hash. When the requester later asks for that block with `getdata`, dcrd sends the block followed by an `inv` containing the single hash of its current best block, which prompts a legacy client to send the next `getblocks`.

15.7.8 `getheaders`

Requests the headers of best-chain blocks following a locator. The payload is identical to `getblocks` (Table 15.16).

The response is always exactly one `headers` message, possibly empty, with the headers of the blocks returned by Algorithm 15.3 with a maximum of `MaxBlockHeadersPerMsg` $= 2000$. If the cumulative proof of work of

the responder's own best chain is below the network's minimum known chain work (`MinKnownChainWork`), it answers with an empty `headers` message instead, because its chain is evidently not yet synced and its headers would be of no use.

dcrd suppresses a `getheaders` it is about to send if its first locator hash and stop hash equal those of the previous `getheaders` it sent to the same peer.

15.7.9 `headers`

Delivers block headers, in answer to `getheaders` or unsolicited as a block announcement.

Table 15.17 `headers` payload (maximum 362003 bytes)

Field	Type	Bytes	Description
`count`	`varint`	1-3	Number of headers, at most 2000.
`headers`	`entry[count]`	181 each	Each entry is a 180-byte serialized block header (Chapter 4) followed by a varint transaction count that **MUST** be zero (a single `0x00` byte).

A non-zero transaction count is a wire protocol violation. How the receiver processes headers is the core of synchronization and is specified in Section 15.8.4.

15.7.10 `block`

Delivers a full block, serialized as in Chapter 4. The maximum payload is `MaxBlockPayload` $= 1310720$ bytes (1000000 at versions 3 and below), and each of the two transaction trees may contain at most $\lfloor\lfloor 1310720/15 \rfloor/2\rfloor + 1 = 43691$ transactions (33334 at versions 3 and below), where 15 is the size of the smallest possible transaction. A count above that bound is a wire protocol violation. These are wire limits only; the consensus size limit is stricter (Chapter 10).

dcrd processes a `block` only if it requested that block from that peer; an unrequested block causes an immediate disconnect. There is consequently no unsolicited block propagation in Decred: new blocks are announced by header and fetched by `getdata`.

15.7.11 `tx`

Delivers one transaction, serialized as in Chapter 4, with maximum payload 1310720 bytes (1000000 at versions 3 and below).

For historical reasons dcrd accepts unsolicited transactions: a `tx` that was not requested is processed like any other. Transactions are submitted to the mempool, whose acceptance rules are in Chapter 17; rejected transactions are remembered (in a probabilistic filter of about 62500 entries) and neither requested nor reprocessed until the next block extends the best chain. In blocks-only mode received transactions are ignored.

15.7.12 `mempool`

Requests announcements of the responder's mempool. No payload.

dcrd responds by queuing an `inv` announcement for every transaction in its mempool through the normal trickling mechanism (Section 15.8.9), skipping transactions the requester is known to have. Each `mempool` adds 33 to the sender's decaying ban score, so more than three in quick succession lead to a ban. dcrd never sends `mempool`.

15.7.13 `ping` and `pong`

Table 15.18 `ping` and `pong` payload (8 bytes)

Field	Type	Bytes	Description
`nonce`	`uint64`	8	Arbitrary value chosen by the sender of `ping` and echoed in `pong`.

A peer **MUST** answer every `ping` with a `pong` carrying the same nonce. dcrd sends a `ping` with a random nonce every 107 seconds (the default idle timeout of 120 seconds less 13), so that an otherwise idle but healthy connection always has traffic before the idle timeout expires. A `pong` whose nonce matches the most recent outstanding `ping` updates the round-trip statistics; any other `pong` is ignored. dcrd does not set a response deadline for `ping`.

15.7.14 `sendheaders`

Asks the receiver to announce new blocks with `headers` messages instead of `inv`. No payload. Valid from version 3.

dcrd sends `sendheaders` to every peer immediately after the handshake and honours it from any peer for the rest of the connection.

15.7.15 `feefilter`

Table 15.19 `feefilter` payload (8 bytes)

Field	Type	Bytes	Description
`min_fee`	`int64`	8	Minimum fee rate, in atoms per kilobyte, below which the sender does not want transaction announcements.

Valid from version 5. dcrd neither sends `feefilter` nor acts on it: a received `feefilter` is decoded and discarded, and transaction announcements are not filtered by fee.

15.7.16 `reject` (obsolete)

Defined only for versions below 9 (`RemoveRejectVersion`), and therefore never at a version dcrd negotiates. In fact dcrd's message reader no longer maps the command string `reject` to any message type, so a received `reject` fails with an unknown-command violation at every protocol version, before its length or payload is examined, and leads to a ban. The layout, which the wire library can still encode and decode at versions 1 to 8, is given for completeness.

Table 15.20 `reject` payload (versions 1 to 8; maximum `MaxMessagePayload`)

Field	Type	Bytes	Description
`command`	`varstring`	1+	Command of the rejected message, strict ASCII.
`code`	`uint8`	1	Reason code: `0x01` malformed, `0x10` invalid, `0x11` obsolete, `0x12` duplicate, `0x40` non-standard, `0x41` dust, `0x42` insufficient fee, `0x43` checkpoint.
`reason`	`varstring`	1+	Human-readable reason, strict ASCII.
`hash`	`byte[32]`	0 or 32	Present only when `command` is `block` or `tx`: the hash of the rejected object.

15.7.17 `getinitstate` and `initstate`

These two messages, introduced at version 8, let a node that has just come online learn ephemeral state that is not part of the chain: which blocks at the tip are currently viable to build on, the votes cast on them, and the treasury spend transactions awaiting votes. Without them a miner would have to wait for the next block, and a voting wallet for new treasury spends to be relayed, before learning this information.

Table 15.21 `getinitstate` payload (maximum 1057 bytes)

Field	Type	Bytes	Description
`count`	`varint`	1	Number of requested types, at most `MaxInitStateTypes` $= 32$.
`types`	`varstring[count]`	1-33 each	Requested types: strict ASCII strings of at most `MaxInitStateTypeLen` $=$ 32 bytes each.

Table 15.22 Initial state types

Type string	Requested data
`headblocks`	Hashes of the blocks at the tip's height (the tip and any competing chain tips) that have enough votes to be built upon.
`headblockvotes`	Hashes of the votes for those blocks.
`tspends`	Hashes of the treasury spend transactions in the responder's mempool.

Table 15.23 `initstate` payload (maximum 1763 bytes)

Field	Type	Bytes	Description
`block_count`	`varint`	1	Number of block hashes, at most `MaxISBlocksAtHeadPerMsg` = 8.
`blocks`	`byte[32][block_count]`	32 each	Block hashes.
`vote_count`	`varint`	1	Number of vote hashes, at most `MaxISVotesAtHeadPerMsg` = 40.
`votes`	`byte[32][vote_count]`	32 each	Vote transaction hashes.
`tspend_count`	`varint`	1	Number of treasury spend hashes, at most `MaxISTSpendsAtHeadPerMsg` = 7.
`tspends`	`byte[32][tspend_count]`	32 each	Treasury spend transaction hashes.

Requesting. dcrd sends one `getinitstate` per connection, when the local node first believes it is synced (immediately for peers that connect after that point). It always requests `tspends` and, unless mining state synchronization is disabled by configuration, also `headblocks` and `headblockvotes`, in the order `tspends`, `headblocks`, `headblockvotes`. It expects the `initstate` within the stall timeout (Section 15.9.2) and disconnects a peer that does not answer.

Responding. A peer **MUST NOT** send more than one `getinitstate` per connection; dcrd bans a peer that does. Unknown type strings are ignored and duplicate strings are harmless. dcrd answers as follows:

- If its best chain height is below `StakeValidationHeight` −1, an `initstate` with all three lists empty.
- Otherwise, if `headblocks` or `headblockvotes` was requested, it takes the current tip and every other known chain tip at the same height, keeps those for which its mempool holds at least $\lfloor$`TicketsPerBlock` $/2\rfloor + 1$ votes (3 on the main network), sorts them by number of votes, most first (the order among equal counts is unspecified, except that if the current tip has as many votes as the first entry it is swapped into first place), and truncates the list to 8. If `headblockvotes` was requested, the vote hashes for every block in that list are concatenated in list order. The block list itself is sent only if `headblocks` was requested.
- If `tspends` was requested, all treasury spend hashes in its mempool, which holds at most 7.

Receiving. A peer **MUST NOT** send more than one `initstate` (or `miningstate`) per connection; dcrd bans a peer that does. On receipt, dcrd requests with a single `getdata` every listed block it neither has nor has already requested, followed by every listed vote and treasury spend it does not already have.

15.7.18 `getminings` and `miningstate` (obsolete)

The predecessors of `getinitstate` and `initstate`, used before version 8. Note that the command string of the request is `getminings` (ten bytes), not `getminingstate`. The wire layer accepts both at every version, but dcrd bans any peer that sends either at a negotiated version of 8 or later, which covers every version it accepts. `getminings` has no payload; `miningstate` is:

Table 15.24 `miningstate` payload (maximum 1546 bytes)

Field	Type	Bytes	Description
`version`	`uint32`	4	Message version; dcrd used 1.

Field	Type	Bytes	Description
height	uint32	4	Height of the sender's best chain tip. (dcrd's handler intended an empty reply of height 0 while its best height was below StakeValidationHeight −1, but its send helper returns without sending when the block list is empty, so no reply is produced in that case.)
block_count	varint	1	At most MaxMSBlocksAtHeadPerMsg = 8.
blocks	byte[32][block_count]	32 each	Block hashes.
vote_count	varint	1	At most MaxMSVotesAtHeadPerMsg = 40.
votes	byte[32][vote_count]	32 each	Vote hashes.

15.7.19 Version 2 Compact Filter Messages

Version 2 compact filters let light clients test whether a block may concern them without downloading it. Their construction and the header commitment that authenticates them are specified in Chapter 16; this section gives only the messages.

Table 15.25 `getcfilterv2` payload (32 bytes, version 7 and later)

Field	Type	Bytes	Description
block_hash	byte[32]	32	Block whose filter is requested.

Table 15.26 `cfilterv2` payload (maximum 263210 bytes)

Field	Type	Bytes	Description
block_hash	byte[32]	32	Block the filter belongs to.
data	varbytes	1+	Serialized filter, at most MaxCFilterDataSize = 262144 bytes.
proof_index	uint32	4	Index of the filter's leaf in the header commitment tree; dcrd always sends 0.
proof_count	varint	1	Number of proof hashes, at most MaxHeaderProofHashes = 32.
proof	byte[32][proof_count]	32 each	Inclusion proof hashes, leaf to root.

dcrd answers `getcfilterv2` with one `cfilterv2` for any block whose filter it has, on the best chain or not, and silently ignores requests for unknown blocks.

Table 15.27 `getcfsv2` payload (64 bytes, version 11 and later)

Field	Type	Bytes	Description
start_hash	byte[32]	32	First block of the range.
end_hash	byte[32]	32	Last block of the range, inclusive.

Table 15.28 `cfiltersv2` payload (maximum 26321001 bytes)

Field	Type	Bytes	Description
count	varint	1	Number of filters, at most MaxCFiltersV2PerBatch = 100.
filters	cfilterv2[count]	var	Each entry is a complete `cfilterv2` payload (Table 15.26), in ascending height order from `start_hash` to `end_hash`.

dcrd answers `getcfsv2` with one `cfiltersv2` when both blocks are known, the start block is the end block or an ancestor of it (on any branch), and the range contains at most 100 blocks, that is $\text{end.height} - \text{start.height} + 1 \leq 100$, and it has the filter of every block in the range. Otherwise the request is ignored without a response.

15.7.20 Version 1 Compact Filter Messages (obsolete)

Protocol version 6 introduced a first generation of committed filters with the messages below, and version 7 deprecated them. The wire layer still decodes them at version 6 and later, but dcrd does not implement version 1 filters and does not advertise `SFNodeCF`: it bans any peer that sends `getcfilter`, `getcfheaders` or `getcftypes` at a negotiated version of 6 or higher (and disconnects it at lower versions), and ignores `cfilter`, `cfheaders` and `cftypes`. The layouts, for completeness:

Table 15.29 Version 1 filter message payloads. `filter_type` is a `uint8`: 0 regular, 1 extended.

Command	Payload
`getcfilter`	`block_hash` (32), `filter_type` (1).
`cfilter`	`block_hash` (32), `filter_type` (1), `data` (varbytes, at most 262144 bytes).
`getcfheaders`	`count` (varint, at most 500), `locator` (32 each), `stop` (32), `filter_type` (1).
`cfheaders`	`stop` (32), `filter_type` (1), `count` (varint, at most 2000), `filter_headers` (32 each).
`getcftypes`	No payload.
`cftypes`	`count` (varint, at most 256), `filter_types` (1 each).

15.7.21 Mixing Messages

Protocol version 10 added eight messages that implement the StakeShuffle coin-mixing protocol: `mixpairreq` (pair request), `mixkeyxchg` (key exchange), `mixcphrtxt` (ciphertexts), `mixslotres` (slot reservation), `mixfactpoly` (factored polynomial), `mixdcnet` (DC-net), `mixconfirm` (confirmation) and `mixsecrets` (revealed secrets). Their payloads, their signatures and hashes, and their validation are specified in Chapter 18; their per-type maximum payloads are in Table 15.12.

At the peer-to-peer layer they behave like transactions: they are announced with `inv` vectors of type `MSG_MIX` whose hash is the message hash, requested with `getdata`, and relayed to peers that negotiated version 10 or later and did not disable relay in their `version` message. A peer **MUST NOT** send them at a lower negotiated version. dcrd bans peers whose mixing messages fail validation in ways the mixing chapter marks as bannable.

15.8 Synchronization and Relay

This section describes how dcrd obtains the chain from its peers and how new blocks, transactions, votes and other data spread through the network. All of it is implementation behaviour rather than a wire requirement, but a node that does not follow the same request-and-response discipline will be disconnected, because dcrd refuses unsolicited blocks and disconnects peers that do not answer requests in time.

Figure 15.1 gives an overview; the subsections below specify each step.

15.8.1 State Used by Synchronization

dcrd distinguishes three views of the chain:

- The **best header** is the tip of the header chain with the most cumulative proof of work that is not known to be invalid. Headers are validated and stored as soon as they arrive, before the blocks themselves.
- The **best chain** is the chain of fully validated blocks with the most cumulative work (Chapter 10). It lags behind the best header while blocks are being downloaded.
- The node is **current** when its best chain has at least `MinKnownChainWork` of cumulative work, has caught up with the best header, and its tip timestamp is at most 24 hours older than the network-adjusted time. Once current, the state latches and is lost again only if the tip becomes more than 24 hours old.

The synchronization manager additionally tracks a **sync height**, the highest height any sync peer has claimed or any accepted header or block has reached, and considers itself synced once the chain is current and its best height has reached the sync height.

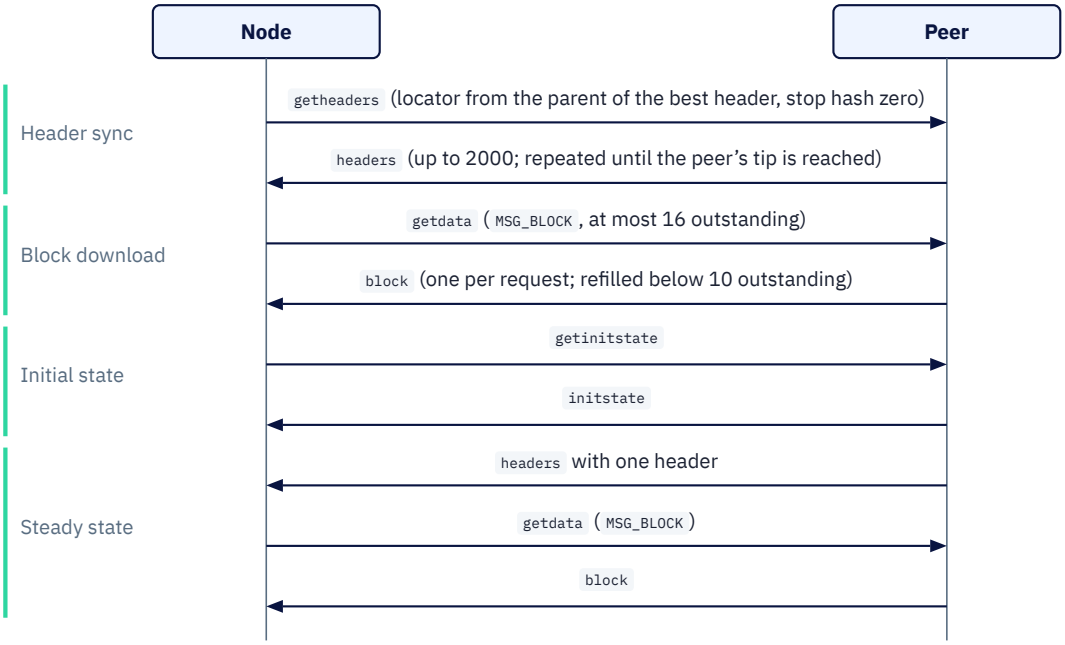


Figure 15.1 Messages exchanged with the sync peer during initial synchronization, and the announcement of a new block afterwards. In the steady state the announcing peer may be any peer. dcrd announces with `inv` (`MSG_BLOCK`) only to a peer that has not sent `sendheaders`, and does not itself request blocks announced that way.

For each peer it tracks the **best announced block**, the announced block with the most cumulative work that is also known locally, and the most recently announced block that did not connect to any known header (an **orphan announcement**), which is promoted to best announced block if its header later becomes known.

15.8.2 Choosing a Sync Peer

Exactly one peer at a time is the **sync peer**. A peer is a candidate if it advertises `SFNodeNetwork` and the best height it reported (initially the `last_block` of its `version` message, later updated from its announcements) is at least the local best header height. Among the candidates the one with the greatest reported height is chosen. A new sync peer is chosen when the current one disconnects, or, if there is none, when a new peer connects.

15.8.3 Initial Header Synchronization

A node that is not yet synced first downloads all headers from the sync peer and only then starts downloading blocks. To begin, and whenever a new sync peer is chosen during this phase, it sends the sync peer a `getheaders` whose locator starts at the *parent* of the local best header and whose stop hash is zero. Starting one block back guarantees a non-empty answer even when both peers already share the same tip, which lets the node learn the peer's real best block. It then sets the sync height to the larger of the peer's reported height and the local best header height.

Each `headers` reply is processed as in Section 15.8.4. If it contains the maximum of 2000 headers, the node immediately asks for more with a `getheaders` whose locator starts at the last received header. The initial header synchronization is complete when the sync peer delivers a `headers` message whose final header's height plus 6 exceeds the sync height.

While this phase lasts, every `headers` message from the sync peer that raises the best header height resets a **stall timer** of $(3 + 2000/1000) \times 2 = 10$ seconds. If the timer fires, the sync peer is disconnected and another is chosen.

When the phase completes, the node sends a `getheaders` (from the parent of its best header) to every peer that serves data and whose best announced block is still unknown, so that it learns their chains as well.

15.8.4 Processing `headers`

Algorithm 15.5 gives the processing of a received `headers` message. Its handling of headers that do not connect deserves attention: a short message whose first header has an unknown parent is treated as an announcement of a block the local node is missing parents for, while a long one is treated as evidence that the peer is on an unrelated chain.

ALGORITHM 15.5 Processing a headers message

```
MAX_ANNOUNCE_HEADERS = 12        # fewer than this looks like an announcement
MAX_ORPHAN_MSGS      = 10        # consecutive non-connecting announcements

def on_headers(peer, headers):
    if len(headers) == 0:
        return                                  # peer has nothing more
    first = headers[0]
    if not have_header(first.prev_block):
        if len(headers) < MAX_ANNOUNCE_HEADERS:
            peer.orphan_msgs += 1
            if peer.orphan_msgs >= MAX_ORPHAN_MSGS:
                disconnect(peer); return
            if initial_header_sync_done:
                peer.push_getheaders(locator(best_header), ZERO_HASH)
            peer.orphan_announcement = hash(headers[-1])
            if not initial_header_sync_done:
                peer.reported_height = max(peer.reported_height, headers[-1].height)
            return
        if initial_header_sync_done:
            disconnect(peer)                    # unrelated chain
        return                                  # ignored during initial sync

    for i in range(1, len(headers)):            # must form a single chain
        if (headers[i].prev_block != hash(headers[i - 1])
                or headers[i].height != headers[i - 1].height + 1):
            disconnect(peer); return

    prev_best_height = best_header.height
    for h in headers:
        if not process_block_header(h):         # header validation, blocks chapter
            if not initial_header_sync_done and peer is sync_peer:
                sync_height = best_header.height
            disconnect(peer); return
        peer.known_inventory.add((MSG_BLOCK, hash(h)))

    if (not initial_header_sync_done and peer is sync_peer
            and best_header.height > prev_best_height):
        reset_header_sync_stall_timer()         # 10 seconds
    peer.orphan_msgs = 0
    update_best_announced(peer, headers[-1])    # also updates reported height
    sync_height = max(sync_height, best_header.height)

    full = len(headers) == 2000
    if (not chain_is_current() and not peer.inbound and not full
            and chain_work(hash(headers[-1])) < MIN_KNOWN_CHAIN_WORK):
        disconnect(peer); return                # outbound peer on a weak chain
    if full:
        peer.push_getheaders(locator(hash(headers[-1])), ZERO_HASH)
    maybe_finish_initial_header_sync(peer, headers[-1])
    if chain_is_current():
        request_blocks_not_held_or_requested(peer, headers)   # getdata
    if initial_header_sync_done and sync_peer is not None:
        fetch_next_blocks(sync_peer)             # Algorithm below
```

dcrd: internal/netsync/manager.go · OnHeaders

The header validation performed by `process_block_header`, including the proof-of-work, difficulty, timestamp and checkpoint checks, belongs to Chapter 10. A header that fails validation causes a disconnect without a ban.

15.8.5 Block Download

Once the headers are synced, blocks are downloaded from the sync peer along the branch leading to the best header, oldest first, with a small window of outstanding requests (Algorithm 15.6). The node keeps a look-ahead list of up to 512 needed block hashes, refreshed when the best header changes or when the list runs short.

ALGORITHM 15.6 Requesting the next blocks from the sync peer

```
MAX_IN_FLIGHT = 16       # per peer
MIN_IN_FLIGHT = 10       # refill threshold

def fetch_next_blocks(peer):
    in_flight = count(r for r in requested_blocks.values() if r == peer)
    if in_flight >= MAX_IN_FLIGHT:
        return
    needed = next_needed_blocks()    # blocks on the path from the best chain tip
                                     # to the best header whose data is missing
    want = min(len(needed), MAX_IN_FLIGHT - in_flight)
    msg = []
    for h in needed[:want]:
        consume(h)
        if h in requested_blocks:    # already in flight from some peer
            continue
        requested_blocks[h] = peer
        msg.append(InvVect(MSG_BLOCK, h))
    if msg:
        peer.send(getdata(msg))

# After a requested block from the sync peer has been processed:
#     if len(requested_blocks) < MIN_IN_FLIGHT: fetch_next_blocks(sync_peer)
```

dcrd: internal/netsync/manager.go · fetchNextBlocks

A block received from a peer **MUST** have been requested from that peer; otherwise dcrd disconnects the peer (Section 15.7.10). Each received block is fully validated and connected (Chapter 10); the use of checkpoints and of the assume-valid block to skip script validation for old blocks is also specified there and has no effect on the messages exchanged. If a block is rejected and it was the block for the best header, the node resets its sync height to the new best header and sends `getheaders` to every peer that serves data, to rediscover a valid best chain.

The initial chain synchronization is complete when the chain becomes current. At that point the node requests the initial state from every peer (Section 15.7.17).

When a peer disconnects, every block, transaction and mixing message that was requested from it and not yet received is re-requested from another peer that is known to have announced it, or forgotten if there is none. If the peer was the sync peer, a new one is chosen and block download resumes from it.

> **NOTE**
> dcrd has no orphan block pool. Because blocks are only requested once their headers have been validated and connected, a block whose parent is unknown can only arrive unsolicited, and unsolicited blocks are refused. Missing ancestry is always resolved at the header level, by `getheaders`.

15.8.6 Steady-State Block Relay

New blocks spread by header announcement:

1. When the node is current and a block whose parent is the current tip has passed its sanity and context checks (Chapter 10), and before it is fully connected, dcrd announces it immediately (not trickled) to every peer that advertises `SFNodeNetwork`. Relaying before the comparatively expensive connection checks is safe in practice because a block with valid proof of work rarely fails them.
2. When the block has been accepted into the block chain, dcrd announces it again to every peer, provided the node is current (or configured to mine while not synced). A peer that already received the announcement in step 1 is skipped, because dcrd suppresses an announcement of the same block it announced last to that peer.
3. To a peer that sent `sendheaders`, the announcement is a `headers` message containing that single header. To any other peer it is an `inv` with one `MSG_BLOCK` vector.
4. The receiver processes the header (Algorithm 15.5). Because it is current, it requests the block with `getdata` from the peer whose announcement arrived first. Side-chain blocks are requested in the same way, so that competing tips are available for the stakeholders to vote on.

15.8.7 Transaction Relay

Transactions, of every type, propagate by `inv` announcement followed by `getdata` and `tx`. When a transaction is accepted to the mempool, from a peer, from RPC or because it resolved an orphan, dcrd queues an announcement to every connected peer that:

- has not disabled relay in its `version` message; and
- is not already known to have the transaction (it announced or sent it, or dcrd announced it to the peer, within the last 15 minutes, among the last 1000 such objects).

Mixing messages are announced under the same conditions and only to peers that negotiated version 10 or later.

When an announcement arrives, dcrd requests the transaction only if all of the following hold: the local node is synced, the transaction is not in the mempool or orphan pool, it is not among the transactions received from peers and rejected since a block received from a peer last extended the best chain, it was not confirmed in one of the most recent blocks (about 23000 transactions are remembered), and it is not already being requested from some peer. At most 50000 outstanding transaction requests are tracked; beyond that a random outstanding entry is forgotten. Orphan transactions, those spending outputs not yet known, are kept in the mempool's orphan pool tagged with the peer that sent them and evicted when that peer disconnects (Chapter 17).

Transactions submitted with `sendrawtransaction`, other than votes, are kept in a rebroadcast set until they are mined, and the whole set is re-announced through the normal relay path five minutes after the node starts and thereafter at intervals chosen uniformly at random below 30 minutes. Each time a block is connected or disconnected, tickets whose price no longer equals the next stake difficulty or that have expired are removed from the set, as are revocations whose ticket is not in the live ticket set. (The code comment describes the last condition as a ticket that "has been revived", but the test is that the ticket is *not* live, which is the normal state of a ticket being revoked, so a revocation is in practice dropped from the set at the first connected block.)

15.8.8 Votes, Tickets and Treasury Spends

Ticket purchases, votes, revocations and treasury spends are transactions of the stake tree (Chapter 8, Chapter 12) and use the ordinary transaction relay; there is no separate inventory type. Votes are time-critical because a block cannot be built on the current tip until enough votes for it are available, which is why new blocks are announced immediately and why a node that comes online asks for the current tip candidates and their votes with `getinitstate` (Section 15.7.17). Treasury spends wait in the mempool for many blocks while stakeholders vote on them, so a newly started node learns of them through the `tspends` initial state type rather than by waiting for a rebroadcast.

15.8.9 Inventory Trickling

dcrd does not announce transactions and mixing messages one by one. Each peer has a queue of pending inventory that is flushed after a random delay uniformly chosen between 100 and 500 milliseconds, re-chosen after every flush. A flush sends as many `inv` messages as needed, each with at most 1000 vectors, skipping items that became known to the peer in the meantime. Block announcements bypass the queue and are sent at once.

15.9 Peer Management

Everything in this section is policy of the reference implementation. It is specified because other nodes will apply it to any implementation that connects to them, and because its constants explain the limits an implementation will encounter.

15.9.1 Connection Limits

Table 15.30 Connection limits (dcrd defaults)

Limit	Default	Meaning
Maximum peers	125	Total of inbound, automatic outbound and one-off manual connections, including pending ones. Persistent connections and inbound connections from whitelisted addresses do not count.
Target outbound	8	Automatic outbound connections maintained (reduced to the maximum peers if that is smaller).
Per host	5	Connections of any kind, including persistent ones, with the same IP address. Loopback and whitelisted addresses are exempt.
Persistent peers	8	Manually configured peers that are kept connected and retried. They count towards neither the maximum peers nor the outbound target.
Outbound per group	1	An automatic outbound connection is not made to a network group (below) that already has an outbound connection of any kind: automatic, manual or a registered persistent peer, pending ones included.

Outbound selection. To fill an outbound slot dcrd draws candidate addresses from the address manager, which only returns routable addresses, and makes up to 100 draws. It skips an address whose network group already has an outbound connection; during the first 30 draws it also skips addresses attempted in the last ten minutes, and during the first 50 it skips addresses whose port is not the network's default port. Tor addresses are eligible only when an onion-capable proxy is configured. After 25 consecutive failures (to find a suitable address or to connect), automatic outbound connections pause for 5 seconds before each further attempt until one succeeds. The **network group** of an address is its /16 prefix for IPv4 (also for IPv6 addresses that embed an IPv4 address: 6to4, Teredo and the translation prefixes), its /32 prefix for other IPv6 addresses (/36 inside the Hurricane Electric block 2001:470::/32), and the low four bits of the first byte of the public key for Tor v3.

Persistent peers are retried with exponential backoff. When a dial fails, or a connection ends, less than 5 seconds after the dial that produced it was started, the retry count r is incremented and the next attempt waits a random time uniformly distributed in $[b/2, b)$, where $b = \min(5\text{ s} \times 2^{r-1}, 5\text{ min})$. Otherwise r is reset to zero and the peer is redialed at once; this includes a dial that takes 5 seconds or longer to fail, such as one that runs into the 30-second dial timeout.

Inbound acceptance. An inbound connection is accepted and then immediately closed if any of the following applies, unless the address is whitelisted or loopback where noted:

- The remote IP address is banned.
- **Rate limit** (not for whitelisted or loopback): each inbound group may open on average one connection every 5 seconds, with bursts of 3. The inbound group is the full IPv4 address or the /64 IPv6 prefix, coarsened to /24 and /56 while a flood is detected, that is while more than 300 connections were allowed through the rate limit in the last 60 seconds. During a flood, connections that pass the rate limit are

additionally dropped at random with a probability rising smoothly from 0.2 to 0.85 as the flood intensity grows to 1200 allowed connections per minute.
- Another connection with the same IP address *and* port exists.
- The per-host limit (not for whitelisted or loopback addresses) or the maximum peers limit (not for whitelisted addresses) would be exceeded.

15.9.2 Timeouts

Table 15.31 Timeouts and periodic actions

Name	Value	Effect
Handshake	30 s	The `version` / `verack` exchange must finish within this time.
Idle	120 s	Each read waits at most this long for a complete message; on expiry the peer is disconnected. Configurable, minimum 15 s.
Ping interval	107 s	A `ping` is sent at this interval. Fixed; it does not follow a configured idle timeout.
Response stall	30 s	Deadline for every object requested with `getdata` (answered by the object or `notfound`) and for the `initstate` answering `getinitstate`. Checked every 15 s; time spent processing messages from the peer extends the deadlines.
Write stall	20 s + 1 s per 256 KiB	Deadline for writing one message to the socket.
Header sync stall	10 s	Maximum time without header progress from the sync peer during initial header synchronization.
Dial	30 s	Outbound connection attempt timeout.

Two queue limits complement the timeouts. A peer is disconnected if the messages queued for sending to it exceed 40 MiB in total, and when the local node is about to send it a `getdata` that would raise the number of objects requested from that peer and not yet delivered (by the object or a `notfound`) above 75000. No response deadline is set for `ping`, `mempool`, `getblocks` or `getheaders`, because a peer with nothing to report may legitimately not answer them, or may answer after a long backlog.

15.9.3 Misbehaviour and Banning

dcrd responds to misbehaviour in one of three ways:

- **Ban**: the peer's IP address (without port) is banned for 24 hours and the connection is closed. New connections to or from a banned address are closed as soon as they are established. A ban is not applied to whitelisted peers, and banning can be disabled by configuration.
- **Ban score**: a score is increased; when it exceeds the threshold (default 100) the peer is banned.
- **Disconnect**: the connection is closed with no lasting consequence.

The ban score has a **persistent** part, which only grows, and a **decaying** part, which halves every 60 seconds. Algorithm 15.7 specifies it. This is one of the few places where dcrd uses floating-point arithmetic, which is acceptable because the score is local policy.

ALGORITHM 15.7 **Dynamic ban score**

```
HALFLIFE = 60                          # seconds
LIFETIME = 1800                        # seconds after which the decaying part is 0
LAMBDA   = ln(2) / HALFLIFE

persistent: uint32 = 0
transient:  float  = 0.0
last_unix:  int    = 0

def increase(p: uint32, t: uint32, now_unix: int) -> uint32:
    persistent += p
    dt = now_unix - last_unix
```

```
    if t > 0:
        if dt > LIFETIME:
            transient = 0.0
        elif transient > 1 and dt > 0:
            transient *= exp(-LAMBDA * dt)
        transient += t
        last_unix = now_unix
    return persistent + uint32(transient)      # truncation toward zero

# After an increase: if score > THRESHOLD / 2 a warning is logged, and if
# score > THRESHOLD (default 100) the peer is banned unless whitelisted.
```

dcrd: internal/connmgr/dynamicbanscore.go · DynamicBanScore

Table 15.32 lists the ban score increments and Table 15.33 the immediate consequences.

Table 15.32 Ban score increments

Event	Persistent	Decaying
`mempool` received	0	33
`getdata` with n vectors	0	$\lfloor 99n/50000 \rfloor$
`notfound` listing b blocks	$20b$	0
`notfound` listing t transactions	0	$10t$
`notfound` listing m mixing messages	0	$10m$

Table 15.33 Misbehaviour with an immediate consequence

Behaviour	Consequence
Any wire protocol violation (Section 15.3.3): wrong magic, oversized payload, malformed or unknown command, bad checksum, trailing bytes, non-canonical varint, invalid string, count above a limit, message invalid for the negotiated version, out-of-range timestamp, unknown address type	Ban
Empty `inv`, empty `getdata`, empty `addr`	Ban
`addrv2` address whose type does not match its canonical form	Ban
Second `getinitstate`, or second `initstate` / `miningstate`	Ban
`getminings` or `miningstate` at version 8 or later	Ban
`getcfilter`, `getcfheaders` or `getcftypes` at version 6 or later	Ban
Mixing message failing validation in a bannable way (Chapter 18)	Ban
Second `version` or `verack`; first message not `version`; `verack` not where required; self-connection; version below 9; handshake timeout	Disconnect
Unrequested `block`	Disconnect
`notfound` containing an unknown inventory type	Disconnect
`headers` not forming a chain, containing an invalid header, or not connecting after the initial sync and not looking like an announcement; 10 consecutive non-connecting announcements	Disconnect
Outbound peer whose best chain has less than `MinKnownChainWork` during initial sync	Disconnect
Response, write or header-sync stall; idle timeout; queue limits exceeded	Disconnect
Transaction or mixing message announced to a blocks-only node	Disconnect

15.9.4 The Address Manager

dcrd keeps the addresses it has learned in an **address manager** modelled on Bitcoin Core's. Its exact bucketing is local policy, but its structure explains which addresses a node shares and connects to.

Addresses are held in two tables. The **new** table, 1024 buckets of at most 64 addresses, holds addresses that have been heard of but never successfully connected to; an address is placed in a bucket chosen by a keyed hash of its network group and the network group of the peer that reported it, and a frequently heard address

may appear in up to 8 buckets. The **tried** table, 64 buckets of at most 256 addresses, holds addresses to which an outbound connection has succeeded, placed by a keyed hash of the address and its group. The hash key is random per node, so an attacker cannot predict the placement. Unroutable addresses (private, link-local, documentation, loopback, unspecified and similar reserved ranges) are never stored.

To choose an outbound address, the manager picks the tried or new table with equal probability (when both have a matching entry), then a random bucket and a random entry within it, and accepts the entry with probability $\min(1, f \cdot c)$, retrying otherwise. Here f starts at 1 and is multiplied by 1.2 after every rejected draw, and $c = \max(1.5^{-a}, 0.01)$, where a is the number of connection attempts since the last successful handshake, except that $c = 0.01$ for an address that was never attempted or was attempted in the last ten minutes. The address's timestamp plays no part in the choice.

An address is **bad**, and is neither shared nor preferred, if it was not attempted in the last minute and any of the following holds: its timestamp is more than 10 minutes in the future; its timestamp is more than 30 days old; it was attempted at least 3 times and never succeeded; or it has not succeeded in the last 7 days and has failed at least 5 times. The manager reports that it needs more addresses while it holds fewer than 1000, and saves its state to disk every 10 minutes and on shutdown.

15.9.5 Address Relay

dcrd's address relay is deliberately conservative, to make it hard for others to learn which addresses a node knows:

- **Answering `getaddr`.** Only for inbound peers, only once per connection. dcrd takes every known address of a type the negotiated version can encode (IPv4 and IPv6 below version 12, also Tor v3 at version 12) that is not bad and has connected successfully at least once, selects a random 23% of them rounded up (at most 2500), removes those the peer is already known to have, and sends the remainder, truncated at random to 1000, in one `addr` or `addrv2` message. If nothing remains, nothing is sent.
- **Self-advertisement.** After the handshake with an outbound peer, as described in Section 15.6.4.
- **Receiving addresses.** Every address in a received `addr` or `addrv2` whose timestamp is more than 10 minutes in the future has its timestamp replaced by the current time minus 5 days. The addresses are recorded as known to the sending peer and added to the address manager with that peer as their source.
- **No forwarding.** Received addresses are never relayed to other peers. They reach other nodes only when those nodes send `getaddr`.

dcrd tracks, per connection, up to 10000 addresses the peer is known to have (in a probabilistic filter with a false-positive rate of 1 in 1000), so that it does not send them back.

15.9.6 External Address Discovery

A node that accepts inbound connections needs to know its own public address to advertise it. Besides addresses configured explicitly or found on local interfaces, dcrd learns it from the `addr_you` field of its peers' `version` messages. Each report by an outbound peer of a routable, non-local address from which that peer's own address is reachable gives that IP address (the port is ignored) a point. Reports from inbound peers are intended only to add points to an address already reported by an outbound peer, but they are looked up under the address *and port* while outbound reports are stored under the bare IP address, so at the reference commit an inbound report never matches and has no effect. When, after an outbound report, the best-scoring address of a family (IPv4 or IPv6) has at least 60% of the outbound target rounded up (5 with the default target of 8), it is added as a local address, provided it matches a listening interface or is a plausible external endpoint for one. This discovery is disabled when a proxy is used, when an external address is configured or discovery is disabled by configuration, when the node does not listen, and on the simulation and regression networks.

15.9.7 Peer Discovery and Seeding

A node with an empty address manager finds its first peers through **HTTPS seeders**, web services that crawl the network and return addresses of reachable full nodes. The seeder host names for each network are in Chapter 14; the simulation and regression networks have none. The legacy DNS seeds that also appear in the

network parameters are deprecated and no longer queried. Peers can also be configured manually; configuring `--connect` peers disables automatic outbound connections entirely.

At start-up, unless seeding is disabled, dcrd queries every seeder in parallel. The request is

```
GET https://<seeder>/api/addrs?services=1
```

where `services` is the decimal value of the required service flags. Two further query parameters are defined by the client but not used by dcrd: `ipversion` (4 or 6) and `pver` (a protocol version). The connection uses the same dialer as peer connections, and so the same proxy settings, and the request times out after one minute. Any status other than 200 is a failure.

The response body is a sequence of JSON objects, each of the form

```
{"host": "<ip>:<port>", "services": <uint64>, "pver": <uint32>}
```

separated by optional whitespace. It is *not* a JSON array; an array is a parse failure. dcrd reads at most 4096 bytes and at most 16 objects; a truncated or malformed object fails the whole query, while an empty body is a successful query that yields no addresses. Entries whose `host` is not an IP literal with a valid port are skipped. Each accepted address is added to the address manager with the reported services, a timestamp chosen at random between 3 and 7 days in the past, and, as its source, the seeder's own IP address on port 443. The `pver` field is parsed but not used.

If every seeder fails and the address manager still needs addresses, the round is repeated after a delay that starts at 1 second and grows by one second per round up to 10 seconds.

> **NOTE**
> Summary of the main departures from Bitcoin's protocol, for readers who know it: BLAKE-256 single-hash checksum; network magic and ports of their own; unknown commands and version-invalid messages are banned rather than ignored; no `reject` at any accepted version; minimum protocol version 9; `verack` must directly follow `version` at version 12; the inbound peer answers with the negotiated version; block announcements by header and no unsolicited blocks; `addrv2` with a little-endian port and no address forwarding; `getinitstate` / `initstate` for tip, vote and treasury spend state; version 2 compact filters with header-commitment proofs; mixing messages; HTTPS seeders instead of DNS seeds.

CHAPTER 16

Compact Filters and Light Clients

A full node downloads and validates every block. A *light client*, such as a wallet on a phone, cannot afford to, yet it still needs to find the transactions that pay to it or spend its coins. Decred serves such clients with **compact block filters**: for every block, a small probabilistic summary of the scripts the block creates and spends, from which a client can decide locally, without revealing anything about its addresses, whether the block may concern it. Only when a filter matches does the client fetch the whole block.

Since the activation of the `headercommitments` agenda (DCP-0005 [45]), every block header commits to its block's filter. A filter therefore can no longer be forged or withheld by the peer that serves it, but it has also become part of the consensus rules: every full node **MUST** compute each block's filter bit for bit as specified here, because a block whose header carries a different commitment is invalid.

This chapter specifies the keyed hash function SipHash-2-4 as used by the filters (Section 16.2), the Golomb-coded set that encodes a filter (Section 16.3), exactly which data from a block enters its filter (Section 16.4), the header commitment and its inclusion proofs (Section 16.5), and, informatively, how light clients use all of this (Section 16.6). Test vectors produced by dcrd close the chapter (Section 16.7). The block header and the variable-length integer encoding are defined in Chapter 4; BLAKE-256 and the generic Merkle tree construction in Chapter 3; how the activation of an agenda is determined in Chapter 13; and the encodings of the peer-to-peer messages named here in Chapter 15.

> **IMPLEMENTATION PITFALL**
> Decred's filters resemble Bitcoin's BIP158 filters [46] only in outline. The parameters, the reduction function, the serialization of the item count, the removal of duplicates, the encoding of the empty filter, the key, the contents and the commitment scheme all differ. Nothing in this chapter should be implemented from BIP158 or BIP157 [47].

16.1 Overview and Constants

A filter is built in three steps. First, a list of byte strings (*items*), almost all of them transaction scripts, is collected from the block and from the outputs its inputs spend (Section 16.4). Second, each item is hashed with SipHash-2-4 under a 128-bit key taken from the block header, and the 64-bit hashes are mapped into a range sized so that two different items collide with probability about $1/M$. Third, the sorted values are compressed with Golomb–Rice coding into a bit string (Section 16.3). Hashing the serialized filter with BLAKE-256 gives the value the header commits to (Section 16.5).

Table 16.1 lists every constant used. None of them is a per-network parameter: the filter construction is identical on all networks.

Table 16.1 Constants of the compact filter system

Name	Value	Meaning
`B`	19	Golomb–Rice parameter: bit size of the remainder code, bin size 2^{19} (`blockcf2.B`).
`M`	784931	Inverse of the false-positive rate for version 2 block filters (`blockcf2.M`).

Name	Value	Meaning
`KeySize`	16	Size in bytes of a SipHash key (`gcs.KeySize`).
`MaxScriptSize`	16384	Largest script, in bytes, that may enter a filter (`txscript.MaxScriptSize`).
`HeaderCmtFilterIndex`	0	Leaf index of the filter hash in the header commitment Merkle tree.
`MaxCFiltersV2PerBatch`	100	Most filters in one batched reply (`wire` ; policy of the wire protocol).
`MaxCFilterDataSize`	262144	Largest filter accepted in a filter message, 256×1024 bytes (`wire`).
`MaxHeaderProofHashes`	32	Most proof hashes accepted in a filter message (`wire`).

The last three constants belong to the wire protocol, not to consensus. They are listed because a light client needs them.

RATIONALE
For a given false-positive rate $1/M$ the size of a Golomb-coded set is minimized near $M \approx 1.497137 \cdot 2^B$, not at the traditional $M = 2^B$; $M = 784931 = \lceil 1.497137 \cdot 2^{19} \rceil$ is that optimum for $B = 19$. DCP-0005 reports that, measured on the main chain up to block 382280, this pair gave the smallest total of filter bytes plus bytes wasted downloading blocks after false positives, among the alternatives tried.

16.2 SipHash-2-4

Every item is hashed with SipHash-2-4 [48], a keyed pseudorandom function designed by Aumasson and Bernstein that maps a 128-bit key and a message of any length to a 64-bit value. Keying the hash with a per-block value prevents an attacker from crafting scripts that collide in every block's filter and so cause false positives for a chosen victim everywhere.

The key is supplied as 16 bytes. It is split into two 64-bit words, each read **little-endian**: $k_0 = \mathrm{LE}^{-1}(\mathrm{key}[0:8])$ and $k_1 = \mathrm{LE}^{-1}(\mathrm{key}[8:16])$. The result is used as an unsigned 64-bit integer, not as a byte string. Algorithm 16.1 is the complete function; it is the standard SipHash-2-4 with 2 compression rounds per message word and 4 finalization rounds, and an implementation **MAY** use any conforming library provided the key and output conventions just stated are respected.

ALGORITHM 16.1 SipHash-2-4 with 64-bit output

```
def rotl64(x: uint64, b: int) -> uint64:
    return uint64((x << b) | (x >> (64 - b)))

def int_le(b: bytes) -> int:          # little-endian value of 0..8 bytes
    return sum(b[i] << (8 * i) for i in range(len(b)))

def siphash24(k0: uint64, k1: uint64, m: bytes) -> uint64:
    v0 = k0 ^ 0x736f6d6570736575
    v1 = k1 ^ 0x646f72616e646f6d
    v2 = k0 ^ 0x6c7967656e657261
    v3 = k1 ^ 0x7465646279746573

    def siround():                    # all additions wrap modulo 2^64
        nonlocal v0, v1, v2, v3
        v0 = uint64(v0 + v1); v1 = rotl64(v1, 13); v1 ^= v0; v0 = rotl64(v0, 32)
        v2 = uint64(v2 + v3); v3 = rotl64(v3, 16); v3 ^= v2
        v0 = uint64(v0 + v3); v3 = rotl64(v3, 21); v3 ^= v0
        v2 = uint64(v2 + v1); v1 = rotl64(v1, 17); v1 ^= v2; v2 = rotl64(v2, 32)

    n = len(m)
    full = n - n % 8
    for i in range(0, full, 8):       # each complete 8-byte word
        w = int_le(m[i:i + 8])
        v3 ^= w
```

```
        siphround(); siphround()
        v0 ^= w
    # Final word: the 0..7 remaining bytes, little-endian, with the message
    # length modulo 256 in the most significant byte.
    w = ((n % 256) << 56) | int_le(m[full:n])
    v3 ^= w
    siphround(); siphround()
    v0 ^= w
    v2 ^= 0xff
    siphround(); siphround(); siphround(); siphround()
    return v0 ^ v1 ^ v2 ^ v3
```

dcrd: github.com/dchest/siphash · Hash (called from gcs/gcs.go · newFilter)

TEST VECTOR SipHash-2-4

```
key       000102030405060708090a0b0c0d0e0f
k0, k1    0x0706050403020100, 0x0f0e0d0c0b0a0908
message                                  result (uint64)
(empty)                                  0x726fdb47dd0e0e31
00                                       0x74f839c593dc67fd
00010203040506                           0xab0200f58b01d137
0001020304050607                         0x93f5f5799a932462
0001020304050607 08                      0x9e0082df0ba9e4b0
000102030405060708090a0b0c0d0e           0xa129ca6149be45e5
00 01 02 ... 3e (63 bytes)               0x958a324ceb064572
```

The 15-byte case is the reference vector of the SipHash paper, whose output bytes `e5 45 be 49 61 ca 29 a1` are the little-endian encoding of the integer shown.

16.3 Golomb-Coded Sets

A Golomb-coded set (GCS) is a compressed, sorted list of small hash values. To test whether an item is in the set, one hashes it the same way and looks for the value in the list. Items in the set always match; an item not in the set matches with probability about $1/M$. The construction here is the one dcrd calls a **version 2** GCS; the obsolete version 1 is described in Section 16.3.5.

A version 2 GCS is parameterized by the remainder bit size B ($0 \le B \le 32$), the false-positive parameter M (an unsigned 64-bit integer), a 16-byte SipHash key and the list of items. Block filters always use $B = 19$ and $M = 784931$ (Rule 16.1); the general algorithm is given because the code is written for general parameters and because the test vectors exercise them.

16.3.1 Construction

ALGORITHM 16.2 **Version 2 GCS construction**

```
def gcs_build(B: int, M: uint64, key: bytes, items: list[bytes]) -> bytes:
    if len(items) > 2**31 - 1:          # counted before any item is dropped
        fail("ErrNTooBig")
    k0 = int_le(key[0:8]); k1 = int_le(key[8:16])

    # 1. Hash the non-empty items, keeping only the first occurrence of each
    #    distinct 64-bit hash value.
    hashes, seen = [], set()
    for x in items:
        if len(x) == 0:
            continue
        h = siphash24(k0, k1, x)
```

```
        if h in seen:
            continue
        seen.add(h)
        hashes.append(h)
    N = len(hashes)
    if N == 0:
        return bytes()                    # the empty filter is zero bytes long

    # 2. Map each hash into [0, N*M) with the multiply-and-shift reduction:
    #    the high 64 bits of the full 128-bit product.
    NM = uint64(N * M)
    values = sorted((h * NM) >> 64 for h in hashes)

    # 3. Golomb-Rice code the differences between consecutive values.
    bits = []
    prev = 0
    for v in values:
        d = v - prev                      # d >= 0; d == 0 is possible
        prev = v
        q = d >> B                        # quotient
        r = d & ((1 << B) - 1)            # remainder
        bits += [1] * q + [0]             # quotient in unary: q ones, then a zero
        bits += [(r >> i) & 1 for i in range(B - 1, -1, -1)]  # B bits, MSB first
    while len(bits) % 8 != 0:
        bits.append(0)                    # pad with zero bits to a whole byte
    body = bytes(sum(bits[8*j + i] << (7 - i) for i in range(8))
                 for j in range(len(bits) // 8))  # first bit -> bit 0x80

    # 4. Prefix the number of distinct hashes.
    return varint(N) + body
```

dcrd: gcs/gcs.go · newFilter | gcs/bits.go · bitWriter

Several details of Algorithm 16.2 are easy to get wrong and are spelled out here.

Empty items Zero-length items are discarded before hashing; they never affect the result.

Duplicates Duplicates are removed by comparing the 64-bit SipHash values, before reduction. Identical items therefore count once, and so do distinct items whose SipHash values happen to collide. Two distinct hashes that reduce to the same value in $[0, NM)$ are **not** merged: both are coded, the second with a difference of zero (a single `0` bit followed by B zero bits). N is the number of distinct 64-bit hashes.

Item order Because the hash values are sorted and duplicates are removed by value, the output depends only on the set of non-empty items, not on their order or multiplicity. An implementation may collect items in any order.

Reduction $N \cdot M$ is computed as an unsigned 64-bit product. For block filters it cannot wrap, since $N < 2^{31}$ and $M < 2^{20}$. The reduced value is $\lfloor h \cdot NM / 2^{64} \rfloor$, computed exactly with a 128-bit product; it is not $h \bmod NM$.

Bit order Bits are written into each byte starting at the most significant bit (`0x80`) and ending at the least significant bit (`0x01`), and bytes are filled in order. The remainder is written most significant bit first.

Unary polarity The quotient q is written as q one-bits followed by one zero-bit. A quotient of zero is the single bit `0`.

Padding The final partial byte is filled with zero bits. No length or terminator other than N is stored.

Count N is written with the variable-length integer encoding of Section 4.2.2, in its canonical (shortest) form. For $N < 253$ it is a single byte.

Empty set If no item survives step 1, the serialized filter is the empty byte string. In particular it is **not** the one-byte string `00` that `varint(0)` would give.

The `ErrNTooBig` limit applies to the raw number of items, before empty items and duplicates are removed. It cannot be reached by a block: even a block made entirely of one-byte outputs is far smaller than 2^{31} items. If it were reached, dcrd would reject the block (Rule 16.2 describes the same failure path).

Table 16.2 Golomb–Rice code word for a difference d with $B = 19$. A filter body is the concatenation of the code words of all N differences, followed by zero padding to a byte boundary.

Part	Length (bits)	Content	Example: $d = 1110803$
Quotient	$q + 1$	q ones, then one zero, where $q = \lfloor d/2^{19} \rfloor$	$q = 2$: `110`
Remainder	$B = 19$	$r = d \bmod 2^{19}$, most significant bit first	$r = 62227$: `0001111001100010011`

16.3.2 Filter Hash

The **hash** of a serialized filter F is

$$\mathrm{FilterHash}(F) = \begin{cases} 32 \text{ zero bytes} & \text{if } |F| = 0 \\ \mathrm{BLAKE256}(F) & \text{otherwise} \end{cases} \tag{16.1}$$

where BLAKE-256 is applied to the complete serialization, including the leading item count. The hash is a 32-byte string in internal byte order; its display form is reversed like any other hash (Chapter 2).

> **IMPLEMENTATION PITFALL**
> The hash of the empty filter is 32 zero bytes, not BLAKE-256 of the empty string (`716f6e86...`). Conversely, a filter that some peer encodes as the single byte `00` hashes to `0ce8d4ef...` and will not verify against a commitment to an empty filter. Block filters are not empty in practice, since coinbase transactions carry non-empty output scripts, but the rule matters to general-purpose implementations and to tests. DCP-0005 does not state this exception: read literally, its construction steps write `varint(N)` even for $N = 0$ and its commitment is BLAKE-256 of that serialization. dcrd, which produces the empty string and the all-zero hash, is authoritative.

16.3.3 Matching

Matching is performed only by light clients and is not part of consensus. It is specified so that clients agree with dcrd on the meaning of a filter.

To test a single item x against a serialized filter F built with key k: if F or x is empty, the result is *no match*. Otherwise read N from the front of F, compute $t = \lfloor \mathrm{siphash24}(k_0, k_1, x) \cdot NM / 2^{64} \rfloor$, and decode the code words in order, keeping a running sum s of the decoded differences. Each running sum is one of the set's values; report a match as soon as $s = t$, and no match as soon as $s > t$ or the bits are exhausted.

ALGORITHM 16.3 GCS membership test for one or several items (informative)

```
def gcs_match_any(B: int, M: uint64, key: bytes, F: bytes, xs: list[bytes]) -> bool:
    xs = [x for x in xs if len(x) > 0]
    if len(F) == 0 or len(xs) == 0:
        return False
    N, off = read_varint(F)            # canonical varint; else the filter is malformed
    NM = uint64(N * M)
    k0 = int_le(key[0:8]); k1 = int_le(key[8:16])
    targets = sorted((siphash24(k0, k1, x) * NM) >> 64 for x in xs)
    reader = BitReader(F[off:])        # reads bits from 0x80 down to 0x01
    s, t = 0, 0
    for _ in range(N):
        q = 0
        while reader.bit() == 1:       # end of data: return False
            q += 1
        r = reader.bits(B)             # B bits, MSB first; end of data: False
        s += (q << B) + r
        while t < len(targets) and targets[t] < s:
            t += 1                     # this target is not in the set
        if t == len(targets):
            return False
        if targets[t] == s:
            return True
    return False
```

dcrd: gcs/gcs.go · filter.Match | gcs/gcs.go · filter.MatchAny | gcs/bits.go · bitReader

A single-item test is the special case of a one-element list. The probability that a given item not in the set matches is close to $1/M$, about one in 785 000 for block filters; testing k items at once multiplies it by roughly k.

> **NOTE**
> Two behaviours of dcrd's matcher are worth knowing about, although neither affects consensus. Its single-item matcher does not stop after N values but reads until a value exceeds the target or the data ends; since the padding consists of zero bits, any extra values it decodes repeat the last value and cannot change the result. Its multi-item matcher, at the pinned commit, drops empty query items but bounds its walk through the remaining sorted query values by the length of the *unfiltered* list. When the list contains an empty item, no query value matches, and some value in the filter exceeds every query value, it reads past the end of its value list and panics with an index-out-of-range runtime error instead of returning *no match*. Clients should not pass empty items.

16.3.4 Maximum Size

The largest filter that N distinct items can produce occurs when every difference but one is zero and the remaining one is $NM - 1$. It occupies

$$\left\lceil \frac{N + NB + \lfloor (NM - 1)/2^B \rfloor}{8} \right\rceil + |\mathrm{varint}(N)| \tag{16.2}$$

bytes (`gcs.MaxFilterV2Size`). For block filters this works out to about $(N \cdot 20 + 1.5N)/8$ bytes in the worst case. dcrd's tests evaluate this bound for $N = \lfloor 1310720/14 \rfloor = 93622$, the number of outputs with a 3-byte script (11 bytes of output framing plus the script) that fit in the largest block any network's wire protocol accepts (`wire.MaxBlockPayload`), ignoring header and transaction overhead, and obtain 251 581 bytes. That is not the largest item count of a valid block, because an output may carry a one- or two-byte script. The count is nevertheless bounded: a regular-tree output script must parse as a version 0 script and contain no stake opcode (Chapter 7), which leaves 174 one-byte and 30 533 two-byte scripts (171 and 29 498 once the `treasury` agenda is active); those that contain signature opcodes are limited by the 5000 signature operations of a block; and a block of that size needs at least two transactions and 334 bytes outside its outputs. No block can therefore contribute more than 95 769 distinct items, for which the bound is 257 350 bytes; the bound first exceeds the wire protocol's limit of $256 \times 1024 = 262144$ bytes per filter (Section 16.6) at $N = 97554$. Every block filter thus fits in a filter message on every network; on the main network, whose blocks are at most 393 216 bytes, filters stay below 82 000 bytes. dcrd does not check the size of a filter when it validates or stores a block: a larger filter would not make the block invalid, but dcrd could not serve it, since encoding the `cfilterv2` or `cfiltersv2` message fails and the requesting peer is disconnected. An ordinary filter takes about $B + 2.05 \approx 21.05$ bits per distinct item.

16.3.5 Version 1 Filters

> **HISTORY**
> Before DCP-0005, dcrd built and served uncommitted **version 1** filters. They differ from version 2 in four ways: the false-positive parameter is tied to the bin size ($M = 2^P$ with a single parameter $P \leq 32$); values are reduced with $h \bmod NM$ instead of the multiply-and-shift method; empty items and duplicate hashes are **not** removed, so N is simply the number of items; and N is serialized as a 4-byte big-endian integer. The filter hash was BLAKE-256 of the serialization (zero for an empty filter), and clients chained filters with a *filter header* BLAKE256(FilterHash ‖ previous filter header). The `gcs` package still implements version 1 GCS construction and the filter header, but the pinned dcrd no longer builds version 1 block filters, no longer advertises the `SFNodeCF` service flag, and answers the version 1 requests `getcfilter`, `getcfheaders`

and `getcftypes` by disconnecting the peer (and banning it if its protocol version is at least 6 and banning is enabled). Version 1 filters have no current role and are not part of consensus.

16.4 Block Filter Contents

This section defines the list of items that enters a block's filter. It is the most intricate part of the construction and the part where a mistake splits the chain: an item missing or extra, a stake tag not removed, or a script included in the wrong form all change the filter hash.

The items are, almost exclusively, **public key scripts** (output scripts). For each output the block creates, the filter normally contains that output's script; for each input that spends an earlier output, it normally contains the script of the output being spent. A light client can therefore find both the payments it receives and the spends of its own coins by testing its own scripts. The filter contains no transaction hashes, outpoints or amounts.

16.4.1 Parameters and Key

RULE 16.1 Version 2 block filter
The version 2 filter of a block **MUST** be `gcs_build(19, 784931, key, items)` (Algorithm 16.2), where `items` is the list produced by Algorithm 16.4 for the block, and `key` is the first 16 bytes, in internal byte order, of the block header's `MerkleRoot` field (header bytes 36 through 51).

`dcrd: gcs/blockcf2/blockcf.go · Regular | gcs/blockcf2/blockcf.go · Key`

The key is taken from the `MerkleRoot` field *as it appears in the header*, whatever that field means for the block: the regular-tree Merkle root before DCP-0005 and the combined root of both trees afterwards (Section 4.5). After DCP-0005 the key therefore depends on every transaction in the block.

16.4.2 Predicates

Two predicates on scripts govern which scripts are admitted and in what form.

Excluded scripts. A script with script version v is *excluded* if $v \neq 0$, or it is empty, or it is longer than `MaxScriptSize` = 16384 bytes. The exclusion predicate is applied to regular-tree output scripts and to every spent script; it is not applied to stake-tree output scripts, whose forms are fixed by consensus (all of them have script version 0).

Stake-tagged scripts. Outputs of stake transactions carry a one-byte *stake tag* opcode in front of an otherwise ordinary pay-to-pubkey-hash or pay-to-script-hash script (Chapter 6). So that a client need test only its plain scripts, the tag is removed. A script s with version v is *stake-tagged* if and only if all of the following hold:

- $v = 0$ and $|s| \geq 1$;
- $s[0]$ is one of `OP_SSTX` (`0xba`), `OP_SSGEN` (`0xbb`), `OP_SSRTX` (`0xbc`), `OP_SSTXCHANGE` (`0xbd`) or `OP_TGEN` (`0xc3`);
- $s[1:]$ is a pay-to-pubkey-hash script, exactly the 25 bytes `76 a9 14` h_{20} `88 ac`, or a pay-to-script-hash script, exactly the 23 bytes `a9 14` h_{20} `87`, where h_{20} is any 20 bytes.

`OP_TADD` (`0xc1`), `OP_TSPEND` (`0xc2`) and all other opcodes are not stake tags for this purpose.

For a **spent** script the tag is removed only when the spending input's outpoint names the stake tree **and** the script is stake-tagged. For a **stake-tree output**, the first byte of the script is removed unconditionally, whatever it is (Algorithm 16.4, `add_stake_out`).

16.4.3 The Item List

The spent scripts depend on the state of the unspent transaction output set, which a full node has at hand when it connects the block. For an input i, let $\mathrm{prev}(i)$ be the script version and public key script of the output that i spends, that is, of the output named by i's previous outpoint (transaction hash, output index and tree), as

it is found when the input is validated. An output created by an earlier transaction of the same block qualifies, and so does, in the historical blocks of Section 7.10.1, output 0 of a regular transaction of the same block that a stake transaction spends, although the stake tree is processed before the regular tree. An output restored to the set because the block disapproves its parent's regular tree (Chapter 10) qualifies too. Being spent does not remove an output's script from consideration: inputs of the block that spend outputs created in the same block contribute those outputs' scripts. An implementation that builds the item list only after all of the block's transactions have been applied, as dcrd does, has every such output at hand.

ALGORITHM 16.4 **Items of the version 2 block filter**

```
STAKE_TAGS = {0xba, 0xbb, 0xbc, 0xbd, 0xc3}   # SSTX SSGEN SSRTX SSTXCHANGE TGEN
TREE_REGULAR, TREE_STAKE = 0, 1

def excluded(ver, s):
    return ver != 0 or len(s) == 0 or len(s) > 16384

def is_p2pkh(s):
    return len(s) == 25 and s[0:3] == b"\x76\xa9\x14" and s[23:25] == b"\x88\xac"

def is_p2sh(s):
    return len(s) == 23 and s[0:2] == b"\xa9\x14" and s[22] == 0x87

def is_stake_tagged(ver, s):
    return (ver == 0 and len(s) >= 1 and s[0] in STAKE_TAGS
            and (is_p2pkh(s[1:]) or is_p2sh(s[1:])))

def add_spent(items, txin):
    ver, s = prev(txin)                  # see Rule "Spent scripts"
    if excluded(ver, s):
        return
    if txin.prev_out.tree == TREE_STAKE and is_stake_tagged(ver, s):
        items.append(s[1:])
    else:
        items.append(s)

def add_stake_out(items, s):
    if len(s) > 0:
        items.append(s[1:])              # drop the first byte, whatever it is

def commitment_payment_script(c):        # c: a ticket commitment output script
    h = c[2:22]                          # 20-byte hash
    if c[29] & 0x80:                     # top bit of the 8-byte LE amount field
        return b"\xa9\x14" + h + b"\x87"                  # pay-to-script-hash
    return b"\x76\xa9\x14" + h + b"\x88\xac"              # pay-to-pubkey-hash

def block_filter_items(block) -> list[bytes]:
    items = []
    for i, tx in enumerate(block.regular_txs):
        for out in tx.outputs:
            if not excluded(out.version, out.script):
                items.append(out.script)          # unchanged, never stripped
        if i == 0:
            continue                              # coinbase: inputs skipped
        for txin in tx.inputs:
            add_spent(items, txin)

    for tx in block.stake_txs:
        t = stake_tx_type(tx)
        if t == TICKET:
            for txin in tx.inputs:
                add_spent(items, txin)
            for j, out in enumerate(tx.outputs):
                if j == 0:
```

```
                continue                        # voting rights: never included
            if j % 2 == 0:                      # change output
                if out.value != 0:
                    add_stake_out(items, out.script)
            elif len(out.script) > 0:           # commitment output
                items.append(commitment_payment_script(out.script))
    elif t == VOTE:
        for out in tx.outputs[2:]:
            add_stake_out(items, out.script)
    elif t == REVOCATION:
        for out in tx.outputs:
            add_stake_out(items, out.script)
    elif t == TREASURY_ADD:
        for txin in tx.inputs:
            add_spent(items, txin)
        if len(tx.outputs) == 2 and tx.outputs[1].value != 0:
            add_stake_out(items, tx.outputs[1].script)
    elif t == TREASURY_SPEND:
        for out in tx.outputs[1:]:
            add_stake_out(items, out.script)
    # TREASURYBASE: contributes nothing
return items
```

dcrd: gcs/blockcf2/blockcf.go · Regular, excludeFromFilter, isStakeOutput, commitmentConverter.paymentScript, Entries

Here `stake_tx_type` is the classification of stake transactions of Chapter 8 and Chapter 12, applied in dcrd's order: ticket, vote, revocation, and then, only for transactions whose version is at least 3, treasury add, treasury spend and treasurybase. In a valid block every transaction of the stake tree is of one of these six types, so the regular type never arises there.

RULE 16.2 Spent scripts

For every input from which Algorithm 16.4 takes a spent script (every input of every regular-tree transaction except the first, every input of a ticket, and every input of a treasury add), $\mathrm{prev}(i)$ is the script version and public key script of the output that input spends, in the unspent output set of the point of view in which the block's transactions are connected, including outputs created earlier in the same block, the outputs made spendable by Rule 7.40, and outputs restored by disapproval of the parent's regular tree. dcrd evaluates this after all of the block's transactions have been connected; an output stays available for this purpose after an input of the same block spends it. If that output does not exist, the filter cannot be built and the block **MUST** be rejected (dcrd reports `ErrMissingTxOut`). Inputs of votes, revocations, treasury spends and treasurybases, and the input of the coinbase, are never looked up.

dcrd: gcs/blockcf2/blockcf.go · Regular
internal/blockchain/validate.go · checkConnectBlock
internal/blockchain/utxoviewpoint.go · UtxoViewpoint.PrevScript

Because every input of a valid block spends an existing output, the failure case of Rule 16.2 is not reachable for a block that passes the input checks of Chapter 7, which dcrd applies first.

Table 16.3 summarizes Algorithm 16.4 by transaction class.

Table 16.3 Items contributed to the version 2 block filter by each transaction class. "Stripped" means that the first byte (the stake tag) is removed.

Transaction	Outputs	Spent scripts
Coinbase (regular tree, index 0)	Every output script not excluded, unchanged, including `OP_RETURN` scripts.	None.
Other regular-tree transaction	Every output script not excluded, unchanged, including `OP_RETURN` scripts.	Every input's spent script not excluded; stripped if the outpoint's tree is the stake tree and the script is stake-tagged.

Transaction	Outputs	Spent scripts
Ticket	Output 0 (voting rights): never. Odd outputs (commitments): converted to the P2PKH or P2SH script they commit to. Even outputs ≥ 2 (change): stripped, but only if the value is non-zero.	As for regular transactions.
Vote	Outputs 0 (block reference) and 1 (vote bits): never. All outputs from index 2 on, stripped. This includes the reward outputs and, when present, the trailing treasury-vote `OP_RETURN` output.	None (neither the stakebase nor the ticket).
Revocation	Every output, stripped.	None.
Treasury add	Output 0 (`OP_TADD`): never. Output 1 (change), if the transaction has exactly two outputs and its value is non-zero: stripped.	As for regular transactions.
Treasury spend	Output 0 (`OP_RETURN`): never. Every output from index 1 on (`OP_TGEN` payments), stripped.	None.
Treasurybase	None.	None.

The rules for a ticket's commitment outputs need a word of explanation. A ticket commitment output (Chapter 8) is an `OP_RETURN` script whose bytes 2 to 21 are a 20-byte hash and whose bytes 22 to 29 are an 8-byte little-endian amount; the most significant bit of that amount, bit 7 of byte 29, is set when the hash is a script hash. The filter does not contain the commitment script itself but the plain script that a vote or revocation will later pay: `OP_HASH160 OP_DATA_20` h `OP_EQUAL` if the flag is set, otherwise `OP_DUP OP_HASH160 OP_DATA_20` h `OP_EQUALVERIFY OP_CHECKSIG`. Only those fixed byte positions are read; consensus guarantees that a ticket's odd outputs are at least 32 bytes long.

RATIONALE
The voting-rights output of a ticket, and the references to it in votes and revocations, are left out on purpose. A light client does not have the full blocks and cannot judge their validity, so it should not vote; leaving the voting rights out of the filters makes "SPV voting" impractical. A client can still discover its tickets through their commitment and change outputs (DCP-0005).

IMPLEMENTATION PITFALL
The filter contents in dcrd differ from the text of DCP-0005 in three ways, and dcrd is authoritative:

- A ticket change output with value zero is **not** included. DCP-0005 lists change outputs without qualification.
- Treasury adds and treasury spends contribute items as listed above. They were introduced later by DCP-0006 and are not mentioned in DCP-0005.
- A vote that carries treasury-spend votes has a final output of the form `OP_RETURN` *push* `'T' 'V'` ... (Chapter 12). Because every vote output from index 2 on is passed to `add_stake_out`, this script enters the filter **with its first byte, `OP_RETURN`, removed**, so the item begins with the push opcode (for example `0x23` for one vote tuple pushed directly, or `0x4c` when `OP_PUSHDATA1` is used, as it must be for three or more tuples).

In addition, DCP-0005 states its two special cases more broadly than dcrd applies them: it applies the exclusion of non-version-0, empty and oversized scripts to every item of both trees, whereas dcrd applies it only to regular-tree outputs and spent scripts; and it removes the stake tag of stake-tree outputs also when they are spent, whereas dcrd removes the first byte of a spent script only under the exact conditions of Section 16.4.2. The conditions of this chapter are dcrd's.

IMPLEMENTATION PITFALL
Further traps in Algorithm 16.4:

- Regular-tree output scripts are never stripped, even if they begin with a stake tag opcode, and are filtered with the exclusion predicate. Stake-tree output scripts are always stripped and are not checked against the exclusion predicate.
- Whether a spent script is stripped depends on the **tree field of the spending input's outpoint** and on the exact tagged P2PKH/P2SH form. A stake-tree output whose script is not of that form is included unchanged.
- `OP_RETURN` outputs of regular transactions, including the coinbase's, **are** included (the comment in dcrd's own summary of the rules says "provably unspendable" scripts are excluded; the code includes them, except for zero-value ticket change).
- The first regular transaction's inputs are skipped by position, and its outputs are processed like any other regular transaction's.
- The filter of a block always describes that block's own regular tree, even if a later block disapproves it. Disapproval of the **parent** affects only which outputs the block's inputs may spend.
- Spent scripts are included even when the spent output was created in the same block; and an item that occurs several times (for example a script that is both created and spent in the block) counts once.

HISTORY
Filters exist for every block, including blocks before the activation of `headercommitments`, and are computed with the same rules for all of them. Treasury transactions cannot occur before the `treasury` agenda (DCP-0006) is active, so no special condition is needed for them. The genesis block's filter is computed by the same algorithm with no spent scripts (it has only a coinbase); it is stored and served but never checked, since the genesis block is valid by definition and predates the commitment.

16.5 Header Commitments

DCP-0005 introduced a general mechanism by which a block header commits to arbitrary per-block data structures, and used it first for the block filter. The header's `StakeRoot` field (header bytes 68 through 99), which until then held the Merkle root of the stake transaction tree, was repurposed to hold the **commitment root**: the Merkle root of a tree whose leaves are the hashes of the individual commitments. The stake tree's root was folded into the `MerkleRoot` field at the same time (Section 4.5). Future agendas can add leaves without changing the header format.

16.5.1 Version 1 Commitment Root

The leaves of the commitment tree are defined by a commitment *version*. The only version so far, version 1, has a single leaf, the hash of the version 2 block filter, at leaf index `HeaderCmtFilterIndex` = 0. The Merkle root of a one-leaf tree is the leaf itself (Chapter 3), so

$$\text{CommitmentRoot}_1 = \text{FilterHash}(\text{block filter}) \tag{16.3}$$

with FilterHash from Eq. (16.1).

RULE 16.3 Header commitment root
If the `headercommitments` agenda (DCP-0005) is active for a block, as determined from the block's parent by the rules of Chapter 13, the block header's `StakeRoot` field **MUST** equal, byte for byte, the version 1 commitment root of Eq. (16.3): the BLAKE-256 hash of the block's serialized version 2 filter (Rule 16.1), or 32 zero bytes if that serialization is empty. A block that violates this rule is invalid (dcrd reports `ErrBadCommitmentRoot`).

```
dcrd: internal/blockchain/validate.go · checkConnectBlock
      internal/blockchain/headercmt.go · CalcCommitmentRootV1
internal/blockchain/agendas.go · isHeaderCommitmentsAgendaActive
```

If the agenda is not active for the block, `StakeRoot` holds the stake-tree Merkle root instead and no filter commitment is enforced (Rule 4.8). The point in block validation at which dcrd evaluates Rule 16.3, after all transactions of the block have been connected to the unspent output view, is given in Chapter 10; the result

does not depend on it, except that the spent scripts must be those of Rule 16.2. dcrd builds the filter, and checks the commitment when the agenda is active, as part of the full validation of every block, including ancestors of the assumed-valid block for which it skips script validation; no network, height or flag exempts a block from Rule 16.3 once the agenda is active for it.

HISTORY
`headercommitments` was voted in on the main network with block 431488 as the first block for which it is active (its parent, block 431487, has hash `000000000000000225b8f795704d4ebc484fad4ab09eec615fb44b4a95c3c26`); on testnet3 the first such block is 323328. An implementation must determine activation from the chain as described in Chapter 13.

RATIONALE
DCP-0005 chose to repurpose `StakeRoot` rather than add a field so that the header keeps its size and the offsets of all mining-related fields, leaving mining hardware and software unaffected. Building a Merkle tree over the commitments, even with one leaf, means that a client can later be given a short proof for any single commitment without receiving the others.

16.5.2 Inclusion Proofs

An **inclusion proof** for a commitment consists of a 32-bit **proof index**, the leaf index of the commitment, and a list of **proof hashes**, the sibling hashes met on the path from that leaf to the root, ordered from the leaf upward. With a single leaf, as in version 1, the proof index is 0 and the proof hash list is empty, and verifying the proof reduces to comparing the filter hash with `StakeRoot`. The general algorithms below follow DCP-0005 and dcrd; the underlying Merkle construction (pairwise BLAKE-256 of the concatenation, with the last node of an odd level paired with itself) is that of Chapter 3. Proofs are not consensus data, but light clients and servers must agree on them.

ALGORITHM 16.5 Generating and verifying a commitment inclusion proof

```
def generate_proof(leaves: list[bytes], index: uint32) -> list[bytes]:
    if len(leaves) == 0 or index >= len(leaves):
        return []
    level, proof = list(leaves), []
    while len(level) > 1:
        if len(level) % 2 == 1:
            level.append(level[-1])          # duplicate the last node
        proof.append(level[index - 1] if index % 2 == 1 else level[index + 1])
        level = [blake256(level[j] + level[j + 1]) for j in range(0, len(level), 2)]
        index >>= 1
    return proof

def verify_proof(root: bytes, leaf: bytes, index: uint32, proof: list[bytes]) -> bool:
    if len(proof) > 32:
        return False
    if index > (1 << len(proof)) - 1:        # index must fit in len(proof) bits
        return False
    cur = leaf
    for p in proof:
        cur = blake256(p + cur) if index % 2 == 1 else blake256(cur + p)
        index >>= 1
    return cur == root
```

dcrd: blockchain/standalone/inclusionproof.go · GenerateInclusionProof
blockchain/standalone/inclusionproof.go · VerifyInclusionProof

> **NOTE**
> dcrd stores the commitment leaves of a block only if `headercommitments` is active for it. For earlier blocks it still serves the filter, with proof index 0 and an empty proof, but such a "proof" proves nothing: the header's `StakeRoot` is then the stake-tree root, and the filter cannot be checked against the header.

16.6 Light Clients

This section is informative. It describes how a light client, often called an SPV (simplified payment verification) client, uses headers, filters and proofs, and what it can and cannot verify. dcrd implements the serving side; the client side described here is an outline of how a light client works, not a consensus requirement.

16.6.1 Synchronizing Headers

The client first downloads the chain of block headers from its peers (`getheaders` and `headers`, Chapter 15). For each header it can check everything that depends only on headers: the link to the previous header, the proof of work against the header's `Bits` and the difficulty schedule (Chapter 9), the timestamps, and the stake difficulty `SBits`, whose calculation (Chapter 9) uses only fields of earlier headers. It follows the valid header chain with the most cumulative work.

Decred's hybrid design puts several facts in the header that a client without the blocks cannot check: that the `Voters` count matches the votes actually included and that those votes were cast by the tickets the lottery selected, the `FreshStake`, `Revocations` and `PoolSize` counts, the lottery state `FinalState`, and the treasury and agenda consequences of the votes. The header-only checks above rest on these fields, which are verified only by full nodes. A client therefore trusts, in addition to the proof of work, that the stakeholders' votes that miners include are genuine; an attacker must control both a majority of the hash power and enough tickets to vote on its blocks to deceive it.

The `VoteBits` field of a block's header tells whether the block's votes approved the regular transaction tree of its **parent** (bit 0 set means approved; Chapter 10). A client that finds a relevant transaction in the regular tree of block h should treat it as confirmed only if the header of block $h+1$ approves the parent, since a disapproved regular tree has no effect on the ledger.

16.6.2 Fetching and Verifying Filters

Filters are requested with the messages of Table 16.4, whose encodings are given in Chapter 15. dcrd ignores, without replying, a request for a block it does not have, a batched request whose start block is not an ancestor of its end block, and a batch of more than 100 blocks.

Table 16.4 Peer-to-peer messages for version 2 filters

Message	Since protocol version	Contents and use
`getcfilterv2`	7	Block hash. Requests the filter of one block.
`cfilterv2`	7	Block hash, serialized filter (at most 262144 bytes), proof index (`uint32`), proof hashes (at most 32). Reply to `getcfilterv2`.
`getcfsv2`	11	Start and end block hashes. Requests the filters of a chain segment, both ends included, at most 100 blocks.
`cfiltersv2`	11	A list of at most 100 `cfilterv2` bodies, in chain order from the start block. Reply to `getcfsv2`.

The same data is available from dcrd's JSON-RPC interface (`getcfilterv2`), with the filter in hexadecimal.

For each received filter the client verifies the commitment against a header it has already accepted:

ALGORITHM 16.6 Verifying a received filter against its header (informative)

```
def verify_received_filter(header, msg) -> bool:
    if msg.block_hash != block_hash(header):
        return False
    if not headercommitments_active_for(header):
        return False                    # nothing to verify against
    if msg.proof_index != 0:            # HeaderCmtFilterIndex
        return False
    leaf = bytes(32) if len(msg.data) == 0 else blake256(msg.data)
    return verify_proof(header.stake_root, leaf, msg.proof_index, msg.proof_hashes)
```

dcrd: blockchain/standalone/inclusionproof.go · VerifyInclusionProof
internal/blockchain/headercmt.go · FilterByBlockHash

A client must check the proof index itself: the proof only shows that the hash sits at the stated index, and only index 0 holds the filter.

16.6.3 Matching and Downloading Blocks

The client tests, against each block's filter, the set of scripts it is watching, using `gcs_match_any` (Algorithm 16.3) with $B = 19$, $M = 784931$ and the key formed from the first 16 bytes of the header's `MerkleRoot`. The watched set contains the plain P2PKH and P2SH scripts of its addresses, which match both payments to it and, because spent scripts are included, spends of its outputs; stake outputs are matched by the same plain scripts because their tags are removed. On a match it downloads the block (`getdata`, Chapter 15), checks that the block's transactions hash to the header's `MerkleRoot` (Section 4.5), and scans the transactions. Since each filter is committed, a peer cannot hide a relevant transaction by serving a filter that omits it: the client detects the mismatch and can ask another peer.

16.6.4 Security Model

A light client using committed filters relies on: the validity of the header chain with the most work, and through it on the honesty of the majority of miners and of the voting stakeholders, since it cannot validate transactions or votes itself and Decred has no fraud proofs; connection to at least one honest peer, so that it learns of the best chain; and the collision resistance of BLAKE-256, which binds filters to headers. Filters reveal nothing about the client's interests to the peers that serve them, but downloading particular blocks does leak some information, mitigated by false positives and by fetching blocks from several peers. Blocks from before the activation of `headercommitments` have uncommitted filters, which a client can only trust or replace by downloading those blocks.

16.7 Test Vectors

All vectors in this section were produced by running dcrd's `gcs`, `blockcf2` and `blockchain/standalone` packages at the pinned commit. Hashes are shown in internal byte order unless labelled "display".

16.7.1 Reduction

TEST VECTOR Multiply-and-shift reduction with $NM = 3 \cdot 784931 = 2354793$

```
x (uint64)              high 64 bits of x * 2354793
0x0000000000000000      0
0x0123456789abcdef      10465
0x8000000000000000      1177396
0xffffffffffffffff      2354792
```

16.7.2 A Small Filter, Step by Step

TEST VECTOR GCS with $B = 19$, $M = 784931$, key `000102...0f`

```
items:  "Alex" 416c6578 | "Bob" 426f62 | "" (empty) | "Charlie" 436861726c6965
        | "Bob" 426f62

1. hash (k0 = 0x0706050403020100, k1 = 0x0f0e0d0c0b0a0908)
   "Alex"     0x78c2a5d04dd7e946
   "Bob"      0xb207e659424ba11f
   ""         skipped (empty)
   "Charlie"  0x9a67bfa9c1907c07
   "Bob"      0xb207e659424ba11f   duplicate hash: dropped
   N = 3, N*M = 2354793

2. reduce (high 64 bits of h * 2354793)
   "Alex" -> 1110803    "Bob" -> 1637600    "Charlie" -> 1420282

3. sort, take differences, split at 2^19 = 524288
   value     difference   q   r        code word (unary q, 0, r in 19 bits)
   1110803   1110803      2   62227    110 0001111001100010011
   1420282   309479       0   309479   0 1001011100011100111
   1637600   217318       0   217318   0 0110101000011100110
   total 62 bits, padded with 2 zero bits:
   11000011 11001100 01001101 00101110 00111001 11001101 01000011 10011000

4. serialize: varint(3) || bit string
   filter        03c3cc4d2e39cd4398
   filter hash   b9a61f8ba0d6ce4d68a65684405d66aed7aee53a3f2989bc375854d5b1df3236
     (display)   3632dfb1d5545837bc89293f3ae5aed7ae665d408456a6684dced6a08b1fa6b9

matching: "Alex" yes, "Bob" yes, "Charlie" yes, "Dave" no, "" no
          any of {"Dave", "Charlie"} yes; any of {"Dave", "Eve"} no
```

TEST VECTOR Empty filter

```
items          none, or only empty items (e.g. a single empty item)
filter         (zero bytes)
filter hash    0000000000000000000000000000000000000000000000000000000000000000
for contrast:
BLAKE-256("")  716f6e863f744b9ac22c97ec7b76ea5f5908bc5b2f67c61510bfc4751384ea7a
BLAKE-256(00)  0ce8d4ef4dd7cd8d62dfded9d4edb0a774ae6a41929a74da23109e8f11139c87
```

16.7.3 Genesis Block Filters

The genesis block of each network contains only a coinbase with a single output, so its filter has exactly one item.

TEST VECTOR Main network genesis block filter

```
block (display)  298e5cc3d985bfe7f81dc135f360abe089edd4396b86d2de66b0cef42b21d980
MerkleRoot       0dc101dfc3c6a2eb10ca0c5374e10d28feb53f7eabcc850511ceadb99174aa66
key              0dc101dfc3c6a2eb10ca0c5374e10d28
item             coinbase output 0, version 0, 32 bytes:
                 801679e98561ada96caec2949a5d41c4cab3851eb740d951c10ecbcf265c1fd9
SipHash          0xdcc0d8901bf41c47
N = 1, N*M = 784931, reduced 676859 = 1 * 2^19 + 152571 (q = 1, r = 152571)
code word        10 0100101001111111011 + 3 padding bits
filter           01929fd8
filter hash      47a29cecacd4456da3eafb80e7d8b018a758efcbfe2ae8d4be2aa026c44fa569
  (display)      69a54fc426a02abed4e82afecbef58a718b0d8e780fbeaa36d45d4acec9ca247
```

The genesis header's `StakeRoot` is 32 zero bytes (the root of its empty stake tree), not this hash: the genesis block predates the commitment.

Table 16.5 Filters of the other networks' genesis blocks

Network	Filter	Filter hash (internal order)
testnet3	`01922e18`	`14316678bb53f2d2eb21a1169c5d96712d18a7bc808544ab139ec55f4a63bb35`
simnet	`01927598`	`4047ebc912b70262c4bb991ba452327109fb9575667f9ce37d24cb77e19079b6`
regnet	`01929fd8`	`47a29cecacd4456da3eafb80e7d8b018a758efcbfe2ae8d4be2aa026c44fa569` (same `MerkleRoot` and output as the main network)

16.7.4 A Block With Every Transaction Class

The following synthetic block exercises every rule of Algorithm 16.4. It was built with dcrd's `wire` package; each stake transaction was confirmed to be classified as stated by dcrd's `stake.DetermineTxType`. The notation `P2PKH(aa)` is the pay-to-pubkey-hash script whose 20-byte hash consists of the byte `aa` repeated, `P2SH(aa)` likewise, and `[tag]` a stake tag prefix.

TEST VECTOR Synthetic block: transactions

```
header MerkleRoot = 11 repeated 32 times; key = 11 repeated 16 times
regular tree
  0 coinbase   out0 v0 6a0c40420f000000000000000000      out1 v0 P2PKH(a1)
  1 regular    in0 spends regular-tree P2PKH(b1)
               in1 spends stake-tree [OP_SSGEN]P2PKH(b2)
               in2 spends regular-tree P2PKH(b3) with script version 1
               out0 P2SH(c1)   out1 6a (bare OP_RETURN)   out2 empty script
               out3 P2PKH(c2) with script version 2      out4 P2PKH(a1)
stake tree
  0 treasurybase  out0 c1 (OP_TADD)   out1 6a0c40420f000102030405060708
  1 ticket        in0 spends regular-tree P2PKH(b4)
                  in1 spends stake-tree [OP_SSTXCHANGE]P2SH(b5)
                  out0 [OP_SSTX]P2PKH(d0)
                  out1 commitment hash d1..d1, amount 600, P2PKH flag clear
                  out2 [OP_SSTXCHANGE]P2PKH(d2), value 0
                  out3 commitment hash d3..d3, amount 400, P2SH flag set
                  out4 [OP_SSTXCHANGE]P2PKH(d4), value 5
  2 vote (tx version 3)
                  out0 6a24 00..00   out1 6a020100   out2 [OP_SSGEN]P2PKH(e1)
                  out3 6a 23 5456 (e2 x 32) 01        (one treasury vote)
  3 revocation    out0 [OP_SSRTX]P2SH(e3)
  4 treasury add  in0 spends regular-tree P2PKH(b8)
                  out0 c1 (OP_TADD)   out1 [OP_SSTXCHANGE]P2PKH(f1), value 7
  5 treasury spend
                  out0 6a20 00..00    out1 [OP_TGEN]P2PKH(f2)
```

TEST VECTOR Synthetic block: items and filter

```
items (18; "P2PKH(a1)" occurs twice, so N = 17)
  6a0c40420f000000000000000000    coinbase out0 (OP_RETURN, included)
  P2PKH(a1)                       coinbase out1
  P2SH(c1)                        regular tx out0
  6a                              regular tx out1 (OP_RETURN, included)
  P2PKH(a1)                       regular tx out4 (duplicate)
  P2PKH(b1)                       spent by regular tx in0
  P2PKH(b2)                       spent by regular tx in1, tag stripped
  P2PKH(b4)                       spent by ticket in0
  P2SH(b5)                        spent by ticket in1, tag stripped
  P2PKH(d1)                       ticket out1, commitment converted
```

```
  P2SH(d3)                        ticket out3, commitment converted
  P2PKH(d4)                       ticket out4, change, stripped
  P2PKH(e1)                       vote out2, stripped
  235456e2e2...e2e201             vote out3, OP_RETURN byte stripped
  P2SH(e3)                        revocation out0, stripped
  P2PKH(b8)                       spent by treasury add in0
  P2PKH(f1)                       treasury add out1, stripped
  P2PKH(f2)                       treasury spend out1, stripped
not included: regular tx out2 (empty), out3 (version 2), in2 (version 1);
  ticket out0 and out2 (zero change) and the raw commitment scripts;
  vote out0, out1 and the vote and revocation inputs; treasurybase outputs;
  treasury add out0; treasury spend out0

filter       11 9032506d092db48491849d6975300e60449351e14fe0574bb476c8a0
                7c0124116756998ff2720ae47d8c12bac0
filter hash  d3f85d41a099da29efbc3e6d6162dc01092fa574e8dc8f964b544312561692e6
  (display)  e69216561243544b968fdce874a52f0901dc62616d3ebcef29da99a0415df8d3
commitment root (if headercommitments were active) = filter hash;
inclusion proof: index 0, no proof hashes
```

Rebuilding the filter from the 18 listed items alone with `gcs_build(19, 784931, key, items)` gives the same bytes, and every listed item matches while none of the excluded scripts does.

16.7.5 Inclusion Proofs

The version 1 commitment has one leaf, so its proofs are empty. The following vector illustrates Algorithm 16.5 on a three-leaf tree, the shape a future commitment version with three leaves would have.

TEST VECTOR Inclusion proofs over three leaves

```
leaf i = BLAKE-256 of the single byte i
leaf 0    0ce8d4ef4dd7cd8d62dfded9d4edb0a774ae6a41929a74da23109e8f11139c87
leaf 1    4a6c419a1e25c85327115c4ace586decddfe2990ed8f3d4d801871158338501d
leaf 2    49af37ab5270015fe25276ea5a3bb159d852943df23919522a202205fb7d175c
root      0135cd60725091a8795118c7ed50b8bf8a272707938a9fa7b24f0399bbe4a048
proof for index 0:
          4a6c419a1e25c85327115c4ace586decddfe2990ed8f3d4d801871158338501d
          6eda2f28b5a79a4c793914418c1b337cf36f806cefb3a10163db703400fc4951
proof for index 1:
          0ce8d4ef4dd7cd8d62dfded9d4edb0a774ae6a41929a74da23109e8f11139c87
          6eda2f28b5a79a4c793914418c1b337cf36f806cefb3a10163db703400fc4951
proof for index 2:
          49af37ab5270015fe25276ea5a3bb159d852943df23919522a202205fb7d175c
          ad204769bc6f9a916299fe1af8d54828b7403e2dbe65c8888e16e9fa1deae0c5
proof for index 3 (out of range): empty
each of the three proofs verifies against the root
```

Leaf 2 is paired with itself at the first level, so its first proof hash is the leaf itself.

NOTE
The test vectors of DCP-0005 for combined Merkle roots, commitment roots and inclusion proof generation and verification (in display byte order there) all pass against dcrd's `blockchain/standalone` package at the pinned commit, including the case of a 33-hash proof, which is rejected.

CHAPTER 17

Relay Policy and Mining

This chapter describes the *local policy* of the reference node: which unconfirmed transactions it accepts into its memory pool and relays to peers, how it maintains that pool as the chain changes, how it assembles candidate blocks (block templates) for miners, how it hands work to mining hardware, and how it estimates fees.

> NOTE
> **Nothing in this chapter is a consensus rule.** Every value and procedure described here may differ between nodes without affecting which blocks are valid, and none of it appears in a rule box. A node that deviates from it remains in consensus, but it may find that its transactions are not relayed by reference nodes, that it relays transactions other nodes discard, or that the blocks it produces collect fewer votes, fewer fees or less propagation than those of other miners. The values are nevertheless given exactly, with their dcrd names, so that an implementation can match dcrd's behaviour where interoperability matters. The consensus rules that the policy builds on (transaction validity, block validity, stake rules) are specified in Chapter 7, Chapter 10 and Chapter 8, and are only summarized here.

Throughout, the *tip* is the current best-chain tip, the *next block* is the block that would extend it, and next $=$ tip height $+ 1$. Where a check uses "the median time" it means the past median time of the tip as defined in Chapter 10. Amounts are in atoms and fee rates in atoms per 1000 bytes (written atoms/kB; Decred's kilobyte is 1000 bytes). Default values are those of the main network unless stated otherwise.

17.1 Pools and Transaction Classes

The memory pool ("mempool") of the reference node consists of several separate structures:

Main pool Fully validated transactions that may be mined in the next block. Only this pool feeds block templates.

Stage pool Ticket purchases that are otherwise acceptable but spend an output of a transaction that is itself still unconfirmed. Consensus does not allow a ticket to spend an output created in the regular tree of the same block (the stake tree is connected first, and a regular tree only becomes spendable by stake transactions once the next block's votes approve it, see Chapter 10), so such tickets cannot be mined in the next block and are parked (Section 17.3.1).

Orphan pool Transactions that reference at least one output that is neither in the UTXO set nor created by a pool transaction (Section 17.2.8).

Vote tracker An index of recently accepted votes keyed by the block they vote on. It is deliberately not kept in sync with the main pool: entries survive the removal of the vote from the pool until they age out (Section 17.3.2).

Treasury-spend set The treasury spends currently in the main pool, counted against a limit (Section 17.3.4).

The pool has **no overall size limit**, no fee-based eviction, and no limit on chains of unconfirmed transactions. Transactions leave the pool only when mined, when they conflict with a mined transaction, when they expire, when a transaction they depend on fails to re-enter the pool after a reorganization, when a staged ticket fails re-validation, or through the stake-specific pruning rules of Section 17.4. Except when a transaction is mined, a removal takes the transaction's dependents with it.

Transactions are classified by type as in Chapter 8 and Chapter 12. The class determines which policy checks apply and how the block template orders the transaction (Table 17.1).

Table 17.1 Policy treatment by transaction class. "Pool double spend" refers to the check of Section 17.2.7; "Priority" is the stake priority used when ordering block template candidates (Section 17.5.3).

Class	Min. relay fee	Dust check	Sequence locks	Pool double spend	Priority
Regular	yes	yes	yes	all inputs	0
Ticket purchase	yes	no	yes	all inputs	1
Vote	no	no	no	per-ticket rules instead	3
Revocation	no	no	yes	one revocation per ticket instead	0 (automatic: 2)
Treasury add	yes	no	yes	all inputs	0
Treasury spend	yes	no	no	none (single treasury input)	0
Coinbase, treasurybase	Never accepted as standalone transactions.				

17.1.1 Entry Points

A transaction reaches the acceptance procedure of Section 17.2 through one of four paths, which differ in three flags: whether an orphan result is stored in the orphan pool, whether the high-fee limit of Section 17.2.4 is enforced, and whether a transaction already present in the orphan pool counts as a duplicate.

Table 17.2 Acceptance entry points

Path	Orphans kept	High-fee limit	Orphan dup.	Notes
`tx` message from a peer	if `MaxOrphanTxs` > 0	not enforced	rejected	Orphans are tagged with the peer's identifier. Ignored entirely when the node runs with `--blocksonly`. A hash rejected once is ignored until a block received from a peer is added to the main chain (Section 17.4).
`sendrawtransaction` RPC	no (rejected with `ErrOrphan`)	enforced unless `allowhighfees` is true	rejected	The flag defaults to false.
Orphan reprocessing	n/a	not enforced	allowed	Run when a parent becomes available (Section 17.2.8).
Block disconnect / disapproval	no (silently dropped)	not enforced	rejected	Re-adds transactions of blocks leaving the chain (Section 17.4). Marks the transactions as "not new".

The agenda-dependent behaviour of the acceptance checks is selected from the state of the agendas for the block after the tip. The flag for explicit version upgrades (DCP-0008, `explicitverupgrades`) is **always** applied by policy, regardless of the agenda's state; the flags for the decentralized treasury (DCP-0006, `treasury`), automatic ticket revocations (DCP-0009, `autorevocations`) and the two subsidy-split changes (DCP-0010 `changesubsidysplit`, DCP-0012 `changesubsidysplitr2`) are applied when the respective agenda is active. In consequence the pool never accepts a transaction with version greater than 3, and never accepts a regular transaction with an output script version other than 0.

17.2 Mempool Acceptance

Algorithm 17.1 and Algorithm 17.2 give the complete acceptance procedure in the order dcrd applies it (the second continues the first, with the same variables in scope). The order is significant only for which error is reported, except where noted: the orphan determination happens after all context-free policy checks and

before any check that needs the spent outputs, so an orphan has passed every check up to that point and nothing after it.

ALGORITHM 17.1 **Mempool acceptance (policy), part 1: checks before the inputs are known**

```
def maybe_accept(tx, is_new, allow_high_fees, reject_dup_orphans):
    if tx in main_pool or tx in stage_pool:            fail(ErrDuplicate)
    if reject_dup_orphans and tx in orphans:           fail(ErrDuplicate)
    check_transaction(tx, flags)         # consensus sanity (transactions chapter)
    t = determine_tx_type(tx)            # also sets the tree: regular or stake
    if treasury_active and t == TREASURYBASE:          fail(ErrTreasurybase)
    if is_coinbase(tx, treasury_active):               fail(ErrCoinbase)
    next = tip_height + 1
    if tx.expiry != 0 and next >= tx.expiry:           fail(ErrExpired)
    is_vote = t == VOTE
    is_tspend = treasury_active and t == TSPEND
    if (is_vote or is_tspend) and next < StakeValidationHeight:
        fail(ErrInvalid)
    if is_vote:
        check_vote_height(tx)            # ErrOldVote, ErrFutureVote
        check_vote_block(tx)             # ErrVoteBlockUnknown, ErrVoteBlockHeight
    if t == REVOCATION and next < StakeValidationHeight + 1:
        fail(ErrInvalid)
    if not AcceptNonStd:
        check_transaction_standard(tx, t, next, median_time)
                                         # ErrNonStandard, ErrDustOutput
    if t == TICKET and tx.out[0].value < next_stake_difficulty:
        fail(ErrInsufficientFee)
    if non_mix_tx_spends_pair_request(tx):             fail(ErrMixpoolDoubleSpend)
    if spends_misbehaving_mix_output(tx):              fail(ErrMixingMisbehavior)
    if not is_vote and t != REVOCATION:
        check_pool_double_spend(tx)      # ErrMempoolDoubleSpend
    elif is_vote:
        check_vote_double_spend(tx)      # ErrAlreadyVoted, ErrTooManyVotes
    else:
        check_duplicate_revocation(tx)   # ErrDuplicateRevocation
    view = fetch_input_utxos(tx)         # chain + main, stage, transient pools
    for i in range(len(tx.out)):
        if view.has_unspent(outpoint(tx, i)):          fail(ErrAlreadyExists)
    missing = [inp.prev for k, inp in enumerate(tx.inputs)
               if not ((k == 0 and is_vote) or is_tspend)
               and not view.has_unspent(inp.prev)]
    if missing:
        return missing                   # orphan: the caller decides
    return accept_with_inputs(tx, t, view, is_new, allow_high_fees)
```

dcrd: internal/mempool/mempool.go · maybeAcceptTransaction

ALGORITHM 17.2 **Mempool acceptance (policy), part 2: checks that use the spent outputs**

```
def accept_with_inputs(tx, t, view, is_new, allow_high_fees):
    rewrite_fraud_proofs(tx, view)
    if not is_vote and not is_tspend:
        if not sequence_lock_active(tx, view, next, median_time):
            fail(ErrSeqLockUnmet)
    fee = check_transaction_inputs(tx, next, view, prev_header=tip_header)
    if not AcceptNonStd:
        check_inputs_standard(tx, t, view)                  # ErrNonStandard
    if count_total_sigops(tx, view) > MaxSigOpsPerTx:       fail(ErrNonStandard)
    size = serialize_size(tx)            # full serialization: prefix + witness
    if fee < min_relay_fee(size) and t in (REGULAR, TICKET, TADD, TSPEND):
        fail(ErrInsufficientFee)         # TADD, TSPEND only with treasury active
    if not allow_high_fees and fee > min_relay_fee(size * 10000):
```

```
        fail(ErrFeeTooHigh)
    validate_scripts(tx, view, standard_verify_flags())
    if is_vote:
        check_vote_ticket(tx)            # ErrIneligibleTicketVote
    if is_tspend:
        check_tspend(tx)                 # ErrTSpendInvalidExpiry, ErrTooManyTSpends,
                                         # ErrInvalid, ErrTSpendMinedOnAncestor
    if t == TICKET and spends_main_pool_output(tx):
        stage_pool.add(tx)
        return []
    main_pool.add(tx)
    if not is_new and t == REGULAR:
        move every main-pool ticket that spends an output of tx to stage_pool
    if is_vote:
        vote_tracker.add(tx)
    if is_tspend:
        tspend_set.add(tx)
    return []
```

dcrd: internal/mempool/mempool.go · maybeAcceptTransaction

Here `fail(kind)` means the transaction is rejected by policy with the given error kind from Table 17.3; consensus failures inside `check_transaction`, `check_transaction_inputs` and `validate_scripts` are reported with the consensus error kinds of Chapter 7.

Two details of the input lookup matter. First, `fetch_input_utxos` loads the spent outputs from the chain as of the tip, but **excludes the outputs created by the tip's regular tree**, and makes the outputs that tree spent available again, when the pool's own votes already show that tree to be disapproved (Section 17.3.2); outputs of transactions in the main, stage or transient pools are then added, with height `UnminedHeight` (0x7fffffff) and block index `0xffffffff`. Second, the fraud-proof fields of each input (`ValueIn`, `BlockHeight`, `BlockIndex`, which are part of the witness and not of the transaction hash) are overwritten with the values from the view whenever any of them differs, so that relayed copies carry correct values (for an input that spends a pool output these are the placeholders just given, not final values). Consensus input checks are then run with fraud-proof checking enabled, against the tip's header as the previous header.

Because a pool output carries the height `UnminedHeight`, the consensus maturity checks see it as far from mature. Two consequences follow: a transaction that spends an output of an unconfirmed transaction with a nonzero `Expiry` is rejected outright (`ErrExpiryTxSpentEarly`), not held as an orphan, and an unconfirmed output tagged `OP_SSTXCHANGE`, `OP_SSGEN`, `OP_SSRTX` or, with the `treasury` agenda active, `OP_TGEN` cannot be spent in the pool. The first matters to a wallet that sets an expiry on the transaction funding a ticket.

17.2.1 Standard Transactions

When the node does not accept non-standard transactions (the default on the main network, see Table 17.9), every transaction must pass the following checks before its inputs are fetched. `AcceptNonStd` is controlled by the network parameter `AcceptNonStdTxs` (false on the main network, true on the test, simulation and regression networks) and can be overridden by `--acceptnonstd` or `--rejectnonstd`.

1. The transaction is serialized in the full format (prefix and witness).
2. The transaction is final with respect to height `next` and the median time, using the consensus finality function of Chapter 7: a lock time of 0 is final; otherwise the lock time is compared with `next` if it is below 500000000 and with the median time otherwise, and is final if strictly less; otherwise the transaction is final only if every input has sequence number `0xffffffff`.
3. The full serialized size is at most `MaxStandardTxSize` = 100000 bytes.
4. Unless the transaction is a treasury spend, every input's signature script is at most 1650 bytes (`maxStandardSigScriptSize`, enough for a 15-of-15 multisignature P2SH redemption with compressed keys) and is *push only*: every opcode is at most `OP_16` (0x60) and the script parses.

5. Every output has script version 0, and its script is of one of the standard types of Chapter 6 (anything other than `nonstandard`). A bare multisignature output must have between 1 and 3 (`maxStandardMultiSigKeys`) public keys inclusive and between 1 and the number of keys required signatures. A null-data output is `OP_RETURN` alone or `OP_RETURN` followed by a single canonical push of at most 256 bytes (`MaxDataCarrierSizeV0`); a longer push makes the output non-standard.
6. For a **regular** transaction only: no output other than a null-data output is dust (Section 17.2.2), and at most 4 outputs (`maxNullDataOutputs`) are null data. Stake transactions are exempt from both checks.

After the inputs are known, two further checks apply to each input other than the first input of a vote and (with the treasury agenda active) the input of a treasury spend:

1. The script of the spent output, interpreted with its script version, must be of a standard type. In particular any spent output with a script version other than 0 is non-standard.
2. If the spent output is a plain pay-to-script-hash output (type `scripthash`, not the stake-tagged variants), the precise signature operation count of the redeem script (Chapter 6) must not exceed 15 (`maxStandardP2SHSigOps`).

IMPLEMENTATION PITFALL
With `AcceptNonStd` set, the finality check of item 2 is skipped as well, so a node with that setting (the default on the test network) admits transactions that are not yet final into its pool. The block template generator nevertheless excludes non-final transactions (Section 17.5.2).

NOTE
The source comment at the null-data check in `checkTransactionStandard`, and the error message it produces, say a regular transaction may carry only one null-data output, but the code compares the count with `maxNullDataOutputs` $= 4$ using "greater than", so up to four are standard.

17.2.2 Dust

An output is *dust* if spending it would cost more than a third of its value at the minimum relay fee rate r (`MinRelayTxFee`). Because the spending input does not yet exist, a typical pay-to-pubkey-hash input of 165 bytes is assumed. For an output with value v and serialized size s (8 bytes of value, 2 of script version, the varint script length and the script):

$$\text{dust} \iff \text{IsUnspendable}(v, \text{script}) \vee \left\lfloor \frac{v \cdot 1000}{3 \cdot (s + 165)} \right\rfloor < r \tag{17.1}$$

with truncating `int64` division. `IsUnspendable` (Chapter 6) is true for zero-valued outputs, scripts starting with `OP_RETURN`, scripts larger than the maximum script size and scripts that fail to parse; all such outputs are dust, which is why null-data outputs are exempted explicitly.

TEST VECTOR Dust thresholds at the default minimum relay fee (10000 atoms/kB)

```
output type   serialized size   smallest non-dust value
P2PKH         36 bytes          6030 atoms
P2SH          34 bytes          5970 atoms
```

17.2.3 Minimum Relay Fee

The minimum fee for a transaction of n serialized bytes (full serialization) at relay fee rate r atoms/kB is computed by Algorithm 17.3. The default rate is `DefaultMinRelayTxFee` $= 10000$ atoms/kB (0.0001 DCR/kB), configurable with `--minrelaytxfee` (given in DCR/kB).

ALGORITHM 17.3 Minimum relay fee (policy)

```
MAX_AMOUNT = 21_000_000 * 100_000_000

def min_relay_fee(n: int64, r: int64) -> int64:
    fee = tdiv(int64(n * r), 1000)      # int64 product; wraps on overflow
    if fee == 0 and r > 0:
        fee = r                          # tiny transactions pay the full rate
    if fee < 0 or fee > MAX_AMOUNT:
        fee = MAX_AMOUNT
    return fee
```

dcrd: internal/mempool/policy.go · calcMinRequiredTxRelayFee

A regular transaction, ticket purchase, treasury add or treasury spend is rejected with `ErrInsufficientFee` if its fee (inputs minus outputs, as computed by the consensus input checks) is **less than** `min_relay_fee(n, r)`; paying exactly the minimum is acceptable. Votes and revocations are exempt: votes must be feeless by construction, and revocations under DCP-0009 must pay no fee. Coinbase and treasurybase transactions never enter the pool.

There is no other fee exemption. The reference node has no "free transaction" or high-priority area in its blocks and no rate limiter for low-fee transactions; the former `--blockminsize` and `--blockprioritysize` options are accepted only to print a deprecation warning.

TEST VECTOR Minimum and maximum fees at r = 10000 atoms/kB

size (bytes)	minimum fee	maximum fee without allowhighfees
1	10	100000
250	2500	25000000
999	9990	99900000
1000	10000	100000000
1500	15000	150000000
100000	1000000	10000000000

17.2.4 High-Fee Limit

To protect users from fat-finger mistakes, a transaction submitted through `sendrawtransaction` without `allowhighfees` is rejected with `ErrFeeTooHigh` if its fee exceeds `min_relay_fee(n * 10000, r)` (`maxRelayFeeMultiplier` $= 10^4$), i.e. 1 DCR/kB at the default rate. Transactions received from peers and orphans being reprocessed are not subject to this limit.

17.2.5 Signature Operations and Script Flags

The total signature-operation count of a transaction, counted exactly as for the consensus block limit (Chapter 6, including the redeem scripts of P2SH inputs), must not exceed `MaxSigOpsPerTx` $= \left\lfloor \frac{5000}{5} \right\rfloor = 1000$, one fifth of the consensus per-block limit `MaxSigOpsPerBlock` $= 5000$, so that the transaction can always fit in a block next to the coinbase. Violations are reported as `ErrNonStandard` even when `AcceptNonStd` is set.

Scripts are verified with a stricter set of flags than consensus requires. The standard flags are `BaseStandardVerifyFlags`:

- `ScriptDiscourageUpgradableNops`,
- `ScriptVerifyCleanStack`,
- `ScriptVerifyCheckLockTimeVerify`,
- `ScriptVerifyCheckSequenceVerify`,

plus `ScriptVerifySHA256` when the `lnfeatures` agenda is active and `ScriptVerifyTreasury` when the `treasury` agenda is active, both evaluated for the block after the tip. The meaning of each flag is specified in Chapter 6.

These flags are used whether or not `AcceptNonStd` is set, and the block template generator verifies scripts with the same flags.

17.2.6 Expiry, Lock Time and Sequence Locks

All three are evaluated as if the transaction were to be mined in the next block, with the tip's median time standing in for the block time:

- A transaction with a non-zero expiry is rejected with `ErrExpired` if next $\geq$ expiry.
- Finality is checked as part of standardness (Section 17.2.1).
- Except for votes and treasury spends, which spend no ordinary outputs, the relative lock times of Chapter 7 are computed from the input view and the transaction is rejected with `ErrSeqLockUnmet` unless next $>$ MinHeight and median time $>$ MinTime.

Pool transactions that become expired as later blocks arrive are removed (Section 17.4).

17.2.7 Double Spends and Replacement

The pool is **first-seen**: there is no replace-by-fee or any other replacement mechanism. For all classes other than votes and revocations, a transaction is rejected with `ErrMempoolDoubleSpend` if any of its inputs spends an outpoint already spent by a transaction in the main pool or the stage pool. The treasury input of a treasury spend is not checked. Votes and revocations skip this check entirely and are handled by their own rules (Section 17.3.2, Section 17.3.3), which in particular allow a vote and a revocation spending the same ticket to coexist in the pool.

Two further conflict checks protect the StakeShuffle mixing protocol (Chapter 18): a transaction that is not itself a completed mix but spends an output named in a currently accepted pair request is rejected with `ErrMixpoolDoubleSpend`, and a transaction spending an output that the mixing code has flagged as belonging to a misbehaving participant is rejected with `ErrMixingMisbehavior`.

A transaction is also rejected with `ErrAlreadyExists` if any of its own outputs already exists unspent in the chain, which detects resubmission of a confirmed transaction.

17.2.8 Orphans

When Algorithm 17.1 returns a non-empty list of missing outpoints, the transaction is an orphan. On the peer path it is offered to the orphan pool; on the RPC path it is rejected with `ErrOrphan`.

- If `MaxOrphanTxs` is 0, the peer path does not keep orphans either: the transaction is rejected with `ErrOrphan` as on the RPC path. The default is 100 (`--maxorphantx`; negative values are refused at startup).
- An orphan whose full serialized size exceeds `MaxOrphanTxSize` $= 100000$ bytes (equal to `MaxStandardTxSize`) is rejected with `ErrOrphanPolicyViolation`.
- Before an orphan is stored, if the current time is after the next scheduled expiry scan, every orphan older than its expiry time is removed together with all orphans that spend its outputs (recursively), and the next scan is scheduled `orphanExpireScanInterval` $= 5$ minutes later. The first scan is scheduled 5 minutes after the pool is created. Scans happen only when an orphan is added, never on a timer.
- If the pool still holds `MaxOrphanTxs` or more orphans (that is, if adding one would exceed the limit), one arbitrary orphan is evicted (Go map iteration order), without its dependents.
- The orphan is stored with an expiry of `orphanTTL` $= 15$ minutes from now and tagged with the identifier of the peer that sent it. When that peer disconnects, all orphans with its tag are removed together with their dependents.

Whenever a transaction is accepted through the peer or RPC path, and for every transaction of a newly connected block (and, on disconnect, of a parent block whose regular tree becomes valid again), the orphans that spend its outputs are reprocessed breadth-first. Transactions re-added from disconnected blocks (Section 17.4) and staged tickets moved to the main pool do not trigger this. For each output of the transaction, the orphans spending it are tried in turn, in unspecified order; the first one that is accepted stops the scan for that output, is removed from the orphan pool and is appended to the work list so that its own dependents are tried; an

orphan that fails with an error is removed together with its dependents and also stops the scan for that output; one that is still an orphan stays and the scan continues. Afterwards every orphan that spends any outpoint also spent by one of the newly accepted transactions is removed with its dependents, since it is now a definite double spend. Orphans accepted this way are relayed after their parent, except on the disconnect path, which does not relay them.

17.2.9 Error Kinds

Table 17.3 Mempool policy error kinds (`internal/mempool/error.go`)

Kind	Condition
`ErrDuplicate`	Already in the main or stage pool (or orphan pool when duplicates of orphans are rejected).
`ErrInvalid`	Vote or treasury spend before `StakeValidationHeight`; revocation before `StakeValidationHeight` +1; treasury spend failing its structure, Pi key or signature checks.
`ErrCoinbase` , `ErrTreasurybase`	Standalone coinbase or (treasury active) treasurybase.
`ErrExpired`	Expired as of the next block.
`ErrNonStandard`	Any standardness failure, including the per-transaction sigop limit.
`ErrDustOutput`	A non-null-data output of a regular transaction is dust.
`ErrInsufficientFee`	Fee below the minimum relay fee; or ticket price below the next stake difficulty.
`ErrFeeTooHigh`	Fee above the high-fee limit on the RPC path.
`ErrMempoolDoubleSpend`	Input already spent by a main- or stage-pool transaction.
`ErrMixpoolDoubleSpend` , `ErrMixingMisbehavior`	Mixing-protocol conflicts (Section 17.2.7).
`ErrAlreadyExists`	An output of the transaction already exists unspent in the chain.
`ErrOrphan`	Missing inputs on a path that does not keep orphans.
`ErrOrphanPolicyViolation`	Orphan larger than `MaxOrphanTxSize` .
`ErrSeqLockUnmet`	Relative lock times not satisfied for the next block.
`ErrOldVote` , `ErrFutureVote`	Vote outside the accepted height window.
`ErrVoteBlockUnknown` , `ErrVoteBlockHeight`	Vote on an unknown block, or with a height that does not match that block.
`ErrIneligibleTicketVote`	Recent vote using a ticket that did not win for the block it votes on.
`ErrAlreadyVoted`	A vote spending the same ticket on the same block is already in the pool.
`ErrTooManyVotes`	Five votes spending the same ticket are already in the pool.
`ErrDuplicateRevocation`	A revocation of the same ticket is already in the pool.
`ErrTSpendInvalidExpiry`	Treasury spend expiry malformed or voting window too far in the future.
`ErrTooManyTSpends`	Seven treasury spends already in the pool.
`ErrTSpendMinedOnAncestor`	Treasury spend already mined in an ancestor of the tip.

17.3 Stake Transaction Policy

17.3.1 Tickets

A ticket purchase is accepted only if its first output (the stake submission, Chapter 8) has a value of at least the stake difficulty for the next block, `SBits` of that block as computed in Chapter 9; otherwise it is rejected with `ErrInsufficientFee` . Tickets pay the minimum relay fee like regular transactions. There is no limit on the number of tickets in the pool.

A ticket that passes every check but spends an output of a transaction in the main pool is placed in the stage pool instead of the main pool. A staged ticket is relayed like any accepted transaction but is not a template candidate. When a block containing the funding transaction is connected, every staged ticket spending one of that transaction's outputs, and having no remaining inputs in the main pool, is moved to the main pool

without being validated again (it is dropped only if looking up its inputs fails with an error). Conversely, when a regular transaction re-enters the main pool from a disconnected block, tickets in the main pool that spend it are moved back to the stage pool.

Each time a block becomes the new tip, tickets are pruned from both the main and the stage pool if

- their first output value is below the stake difficulty of the block after the new tip (so tickets priced for a window that has ended are dropped when the price rises), or
- they were accepted when the tip height was h_a and $h_a + 288 <$ new tip height (`heightDiffToPruneTicket`).

Pruning from the main pool also removes, recursively, every pool transaction that spends the pruned ticket.

17.3.2 Votes

Votes are the most time-critical transactions in Decred: a block at or above `StakeValidationHeight` cannot be built without a majority of the votes on its parent (Chapter 10). The pool therefore accepts votes on any known block, including blocks on side chains, so that the miner can choose among competing tips. Let h_v and H_v be the height and hash a vote commits to (Chapter 8), and next as before.

- **Height window.** The vote is rejected with `ErrOldVote` if $h_v <$ next $-$ MaxVoteAge, unless the `--allowoldvotes` option is set (default off). `MaxVoteAge` is `CoinbaseMaturity` (256 on the main network, 16 on the simulation and regression networks) except on the test network, where it is 1440. The vote is rejected with `ErrFutureVote` if $h_v >$ tip height $+ 5$ (`maxFutureVoteAge`), regardless of `--allowoldvotes`.
- **Known block.** The block H_v must be known to the node, on any chain (`ErrVoteBlockUnknown`), and its header height must equal h_v (`ErrVoteBlockHeight`).
- **Same ticket, same block.** If a vote spending the same ticket and voting on the same block H_v is recorded in the vote tracker **and** is still in the main pool, the vote is rejected with `ErrAlreadyVoted`.
- **Same ticket, different blocks.** A ticket may be selected by the lottery of several competing blocks at the same height, so votes spending one ticket on different blocks are allowed, but the vote is rejected with `ErrTooManyVotes` if the main pool already contains 5 (`maxVoteDoubleSpends`) votes spending the same ticket.
- **Winning ticket.** After script validation, if $h_v \geq$ next $-$ MaxVoteAge, the ticket spent by the vote must be one of the winning tickets of block H_v (Chapter 8), else `ErrIneligibleTicketVote`. Older votes (admissible only with `--allowoldvotes`) skip this check to avoid an expensive lottery recomputation; consensus rejects them later if they are mined.

Votes skip the pool double-spend check, the sequence-lock check and the minimum fee. An accepted vote is recorded in the vote tracker under H_v (with h_v and whether it approves the parent's regular tree, bit 0 of its vote bits), unless a vote with the same ticket is already recorded for H_v. Accepted votes also notify the background template generator (Section 17.6).

The pool uses the tracker to recognise a **known-disapproved** tip: with n tracked votes on the tip, of which y approve, the tip's regular tree is considered known disapproved if $n > \left\lfloor \frac{\text{TicketsPerBlock}}{2} \right\rfloor$ and $y \leq n - y$. While that holds, new transactions cannot spend outputs of the tip's regular tree, and may again spend the outputs that tree spent (Section 17.2); the template generator treats them likewise.

Votes and revocations in the main pool are pruned when a block becomes the new tip if they were accepted at tip height h_a with $h_a + 10 <$ new tip height (`heightDiffToPruneVotes`). Tracker entries are deleted, whether or not the votes are still in the pool, when the recorded height h_v of the voted block satisfies $h_v + 10 <$ new tip height.

Vote Solicitation

Votes are produced by wallets in response to the node's `winningtickets` notification (Chapter 19). The reference node sends it when a block is accepted (on any chain) only if the node is synced (or unsynced mining is allowed), the block height is at least StakeValidationHeight $- 1$, the depth of the reorganization that the block would cause is less than 6 (`maxReorgDepthNotify`), the block does not spend outputs flagged as belonging

to misbehaving mixing participants, and, on the main network, the block is not a pre-version-11 block at height 1035288 or above. Each block is notified at most once.

17.3.3 Revocations

A revocation is rejected with `ErrInvalid` if next $<$ StakeValidationHeight $+ 1$, since no ticket can have been missed before then. Instead of the pool double-spend check, it is rejected with `ErrDuplicateRevocation` if the main pool already contains a revocation spending the same ticket. It is exempt from the minimum fee. Its validity, including the version and fee rules of DCP-0009, is checked by the consensus input checks against the tip.

Once the `autorevocations` agenda (DCP-0009) is active, every block must itself revoke all tickets that become missed or expired in it, using revocations whose content depends on the previous block's header (Chapter 8). A revocation left in the pool after the block it was built for is therefore useless, and with the agenda active **every** revocation is removed from the pool each time a block becomes the new tip. The template generator creates the required revocations itself (Section 17.5.4).

17.3.4 Treasury Spends and Adds

Treasury adds (Chapter 12) are treated like regular transactions: they pay the minimum relay fee and are subject to the pool double-spend check. Treasury spends, with the treasury agenda active, are subject to the following additional rules, applied after script validation:

1. Not before `StakeValidationHeight` (`ErrInvalid`, checked early).
2. The expiry must define a valid voting window: with T = `TreasuryVoteInterval` and M = `TreasuryVoteIntervalMultiplier`, the expiry e must satisfy $e \geq TM + 2$ and $e - 2 \equiv 0 \pmod{T}$, $e - 2 \neq 0$, giving a voting window starting at $s = e - TM - 2$ (Chapter 12); otherwise `ErrTSpendInvalidExpiry`.
3. The window must not start too far in the future: the spend is rejected with `ErrTSpendInvalidExpiry` if $s >$ next and $s -$ next $\geq 2TM$ (6912 blocks on the main network, where $T = 288$ and $M = 12$).
4. The pool may hold at most 7 treasury spends (`MempoolMaxConcurrentTSpends`, which equals the number of treasury-spend votes a vote can carry); an eighth is rejected with `ErrTooManyTSpends`.
5. The structure and signature are checked as in Chapter 12: the spend must be well formed, its public key must be one of the network's Pi keys, and the signature must verify; failures give `ErrInvalid`.
6. The spend must not already have been mined in an ancestor of the tip (`ErrTSpendMinedOnAncestor`).

Treasury spends skip standardness checks of their signature script, the pool double-spend check of their (only) input and sequence locks, but pay the minimum relay fee computed from their declared input value. They are not considered by the fee estimator (Section 17.8). An accepted spend is announced to RPC clients subscribed to treasury-spend notifications.

17.4 Mempool Maintenance

The pool is updated whenever the best chain changes.

Block connected. For each transaction of the connected block, in order — the regular tree without its coinbase, then the stake tree (without the treasurybase when the treasury agenda is active) — the node:

1. removes the transaction from the main pool (without its dependents, which remain valid);
2. moves staged tickets that spend it and have no other main-pool inputs into the main pool (Section 17.3.1);
3. removes every main-pool transaction that spends an outpoint the block transaction spends, together with its dependents, and every stage-pool transaction that does so;
4. removes the transaction from the orphan pool;
5. reprocesses the orphans that spend it (Section 17.2.8) and relays any accepted.

The fee estimator records the block before these removals (Section 17.8). If the new block disapproves its parent's regular tree (vote bit 0 of its header clear), the parent's regular transactions other than the coinbase are re-added as described below, since disapproval makes it as if they had never been mined.

Block disconnected. If the disconnected block had disapproved its parent, the parent's regular transactions (other than the coinbase) are valid again and are treated as in steps 1–5 above, except that accepted orphans are not relayed. Then the disconnected block's regular transactions other than the coinbase, and afterwards its stake transactions (other than the treasurybase when the treasury agenda is active), are re-added.

Re-adding. A batch of transactions in block order is re-added by processing it in **reverse** order, with the not-yet-processed earlier transactions of the batch made available as a source of inputs (the "transient" pool). Each goes through Algorithm 17.1 marked not new. If it fails for any reason other than being a duplicate, already existing, or spending a missing or spent output, it is removed from the pool together with its dependents; orphan results are discarded.

After a block becomes the tip (by extension or as the result of a reorganization it caused when processed; a forced reorganization by the template generator, Section 17.5.1, does not trigger this), the node prunes stake transactions as in Section 17.3.1, Section 17.3.2 and Section 17.3.3, removes from the main pool (with dependents) and from the stage pool every transaction for which tip height $+ 1 \geq$ expiry $\neq 0$, and, when the block was received from a peer (not when it was submitted locally through `submitblock`, `getwork` or the CPU miner), clears its filter of recently rejected transaction hashes (an approximate filter of up to 62500 entries with a false-positive rate of 10^{-7}, used to avoid re-requesting and re-validating transactions that peers keep announcing).

17.5 Block Template Construction

A *block template* is a complete block, valid in every respect except the proof of work, that the node hands to miners. Templates are built by `NewBlockTemplate` and maintained in the background by the generator of Section 17.6. The generator runs only when at least one mining address is configured (`--miningaddr`); each template pays to one of them chosen uniformly at random.

17.5.1 Choosing the Parent

Before `StakeValidationHeight` the template always extends the tip. From then on a block must contain at least $\left\lfloor \frac{\text{TicketsPerBlock}}{2} \right\rfloor + 1$ votes (3 on every network) on its parent, so the parent is chosen by the votes available:

1. Let the *tip generation* be the set of all chain tips (blocks with no known children) at the tip's height, the tip included.
2. Keep the blocks with at least 3 votes recorded in the vote tracker and sort them by vote count, descending. If the first block has the same count as the current tip but is a different block, swap the current tip into the first position, so that the node does not reorganize between equally supported siblings.
3. If no block qualifies, build the replacement template of Section 17.5.8.
4. Otherwise walk the sorted list. Stop at the current tip. For any other block, ask the chain to *force a reorganization* to it, which succeeds only if it is a sibling of the tip (same parent) whose data is available and that is not known to be invalid; on failure, try the next block. After a successful reorganization, if any vote recorded for the new tip is no longer in the pool, try the next block; otherwise use the new tip as parent.

The forced reorganization is the one place where the reference node deliberately selects between equal-work siblings by policy; chain selection proper is specified in Chapter 10.

> **IMPLEMENTATION PITFALL**
> `NewBlockTemplate` reads the next-block state (winning tickets, expiring tickets, final state, pool size, stake difficulty, missed tickets, tip header, the median time used for the finality filter, agenda states, and the block

passed to the ticket-exhaustion check) from the best-chain snapshot taken **before** any forced reorganization in this step, while the vote filter, the treasury budget, `Bits`, `StakeVersion` and `PrevBlock` use the new parent. The forced reorganization emits the usual reorganization events, on which the background generator clears its current template and afterwards handles the new tip as newly connected (Section 17.6).

17.5.2 Candidate Collection

Every main-pool transaction is examined; it becomes a candidate unless

- it is a coinbase;
- it is not final for height next and the tip's median time;
- it is a vote whose committed block is not exactly (parent hash, next − 1);
- one of its inputs (other than the stakebase input of a vote and the input of a treasury spend) is neither unspent in the chain view (excluding the parent's regular tree when known disapproved) nor created by a transaction whose hash is in the main, stage or orphan pool.

For each candidate the node computes its *fee rate* and *priority*:

$$\text{feePerKB} = \frac{(f + F_a) \cdot 1000}{s + S_a} \tag{17.2}$$

as a floating-point number, where f and s are the transaction's fee and full serialized size and F_a, S_a the total fee and size of its unconfirmed ancestors still in the pool (both zero when ancestor statistics are unavailable). The *priority* is the legacy Bitcoin coin-age priority, used only as a tie-breaker:

$$\text{priority} = \frac{\sum_i v_i \cdot (\text{next} - h_i)}{s - \sum_i (58 + \min(110, |\text{sigScript}_i|))} \tag{17.3}$$

summed over inputs found in the view, where v_i is the input value and h_i the height of the spent output (age 0 when the output is unconfirmed), and the priority is 0 if the denominator is not positive.

A candidate is pushed onto the priority queue immediately if it has no parents in the pool, or if ancestor statistics are available for it (see below). A candidate with pool parents but no statistics waits until all its parents have been included in the template.

Ancestor statistics. When the node has a mining address, the pool tracks for each transaction the total fee, size and signature operations of its unconfirmed ancestors, allowing a low-fee parent to be mined because of a high-fee child ("child pays for parent"). Tracking is abandoned for a transaction (its statistics become unavailable) if it would have more than 25 (`ancestorTrackingLimit`) ancestors, if any of its ancestors has no statistics itself, or if any ancestor already has 25 ancestors or 25 tracked descendants. This limit affects only template construction, not pool admission.

17.5.3 Ordering

The queue pops the candidate that is greatest under the following order:

1. higher **stake priority** first: votes (3), then automatic revocations created by the generator (2), then ticket purchases (1), then everything else (0) — regular transactions, revocations from the pool, treasury adds and treasury spends;
2. within equal stake priority, higher `feePerKB`;
3. for equal `feePerKB`, higher `priority`.

Ties beyond that are broken by the heap's internal order, which is not specified.

17.5.4 Selection

Algorithm 17.4 gives the selection loop. `BlockMaxSize` is the `--blockmaxsize` option (default 375000 bytes, permitted range 1000 to `MaximumBlockSizes`[0] −1000, i.e. 392216 on the main network); `TxMinFreeFee`

equals the minimum relay fee rate. `mark_rejected(x)` removes x and all its pool descendants from consideration for this template.

ALGORITHM 17.4 Block template transaction selection (policy)

```
size = 180 + 9                    # header + maximum varint (blockHeaderOverhead)
sigops = 0
revocations_added = False
while queue:
    x = queue.pop()
    if x in template: continue
    if AUTOREVOCATIONS and not revocations_added and x.type != VOTE:
        # all votes have been considered: revoke every winning ticket that
        # has no vote in the template and every ticket expiring in this block
        for ticket in unvoted_winners() + expiring_tickets():
            queue.push(auto_revocation(ticket))   # version 2, zero fee
        revocations_added = True
        queue.push(x); continue
    if x.type == TSPEND:
        if not is_tvi(next) or not inside_tspend_window(next, x.expiry): continue
        if not tspend_has_enough_votes(parent, x): continue
        if budget - x.value_in < 0: continue
        budget -= x.value_in       # budget = max treasury expenditure (TVI only)
    if x.type == TADD and n_tadds >= 20: continue          # MaxTAddsPerBlock
    if x.type == TICKET and n_tickets >= MaxFreshStakePerBlock: continue
    if x.type == TICKET and x.out[0].value < next_stake_difficulty: continue
    if x.type == REVOCATION and not (
            (AUTOREVOCATIONS and ticket_missed_or_expiring_this_block(x))
            or ticket_in_missed_set(x)):
        continue
    if is_rejected(x): continue
    anc = ancestors_not_in_template(x)
    rate = fee_per_kb(x, anc)
    if rate < x.fee_per_kb:          # some ancestors were already included
        x.fee_per_kb = rate
        if not anc: queue.push(x)    # re-queue with the corrected rate
        continue                     # else wait until all parents are included
    if uint32(size + x.size + total_size(anc)) overflows or \
            size + x.size + total_size(anc) >= BlockMaxSize:
        mark_rejected(x); continue
    if sigops + x.sigops + total_sigops(anc) > 5000:     # MaxSigOpsPerBlock
        mark_rejected(x); continue
    if x.type == VOTE and (ticket_already_voted_in_template(x)
                           or not ticket_is_winner(x)):
        continue
    if x.tree == REGULAR and x.fee_per_kb < TxMinFreeFee:
        mark_rejected(x); continue
    for y in anc + [x]:              # ancestors in topological order
        if not inputs_valid(y) or not scripts_valid(y, standard_flags):
            mark_rejected(y); continue outer loop
    for y in anc + [x]:
        add y to template; spend its inputs; size += y.size; sigops += y.sigops
        count tickets, votes, treasury adds
        queue.push(c) for each child c of y now without parents in the pool
if AUTOREVOCATIONS and not revocations_added:
    add revocations as above and resume the loop
```

dcrd: internal/mining/mining.go · NewBlockTemplate

Notes on Algorithm 17.4:

- `inputs_valid` is the consensus input check of Chapter 7 for height next against the tip header, **without** the fraud-proof check (the fraud proofs are filled in afterwards).

- Automatic revocations are created only after all votes have been considered, so that tickets whose votes failed any check are revoked rather than left unvoted; their format is specified in Chapter 8. A pool revocation identical to an automatic one has the same hash and is skipped as already included.
- The treasury budget is the maximum treasury expenditure for a block extending the parent (Chapter 12), computed only when the next block is at or above `StakeValidationHeight` and on a treasury vote interval. Treasury spends compete for it in queue order.
- The size budget counts every selected transaction but not the coinbase or treasurybase, and the final header `Size` is the actual serialized size; the 1000-byte margin below the consensus limit absorbs the difference.

IMPLEMENTATION PITFALL
The template's fee-rate floor uses the exact floating-point rate of Eq. (17.2), whereas pool admission uses the truncated integer minimum of Algorithm 17.3. A transaction can therefore be admitted but never selected on its own by the reference node (it can still be included as an ancestor of a child whose combined rate passes the floor, since the floor is applied only to the popped candidate): at a relay rate of 2550 atoms/kB a 782-byte transaction needs a fee of 1994 atoms to enter the pool, but its rate of 1994000/782 ≈ 2549.87 atoms/kB is below the floor. At the default rate of 10000 atoms/kB the minimum fee is always exactly $10n$ and the effect does not occur.

17.5.5 Assembling the Trees

After selection, the two transaction trees are assembled from the selected transactions in selection order. Every stake transaction placed in the stake tree is first copied and has its inputs re-resolved against the chain UTXO view (excluding the parent's regular tree when known disapproved), with its fraud-proof fields set from that view; a stake transaction with an input not found there is dropped. The stake tree is, in order:

1. the treasurybase, if the treasury agenda is active (Section 17.5.6);
2. if next $\geq$ StakeValidationHeight, the selected votes (at most `TicketsPerBlock`, since a vote is only selected for a distinct winning ticket);
3. the selected ticket purchases that spend no output of any selected transaction and still pay at least the next stake difficulty, up to `MaxFreshStakePerBlock` (20);
4. if next $\geq$ StakeValidationHeight, the selected revocations, up to 255;
5. if the treasury agenda is active, the selected treasury adds and treasury spends, interleaved in selection order.

Selected transactions dropped here are simply absent from the block. The node then checks that adding the block with this number of ticket purchases cannot lead to ticket exhaustion (Chapter 8) and abandons the template otherwise.

The regular tree is the coinbase followed by the selected regular transactions in selection order, which places every parent before its children. The fraud-proof fields of each regular input (from height 2 on) are set from the chain view; an input whose output is not in the chain must be created by an earlier transaction of the same tree, and gets `ValueIn` = that output's value, `BlockHeight` = next and `BlockIndex` = the creating transaction's index in the regular tree.

If next $\geq$ StakeValidationHeight and fewer than 3 votes ended up in the stake tree, the template is abandoned in favour of the replacement template of Section 17.5.8.

17.5.6 Coinbase and Treasurybase

The amounts are those of Chapter 11 and Chapter 12, computed for height next with the number of votes V in the template and the subsidy-split variant selected by the active agendas.

Fees. Let F be the sum of the fees of all transactions in both trees. From `StakeValidationHeight` on, $F \leftarrow \left\lfloor F \cdot \frac{V}{\text{TicketsPerBlock}} \right\rfloor$ (the same reduction consensus applies, Chapter 10). For heights above 1, F is added to the value of the proof-of-work output.

Table 17.4 Coinbase transaction built by the template generator

Element	Value
Version	1 before the treasury agenda; 3 (`TxVersionTreasury`) once it is active.
Input	Previous outpoint: zero hash, index `0xffffffff` , regular tree; sequence `0xffffffff` ; `BlockHeight` 0; `BlockIndex` `0xffffffff` ; `ValueIn` = work subsidy + treasury subsidy (the latter only before the treasury agenda), **excluding** fees.
Signature script	`00 00` followed by the ASCII bytes of `/dcrd/` (8 bytes).
Output 0 (before the treasury agenda only)	If `BlockTaxProportion` > 0: the treasury subsidy paid to `OrganizationPkScript` (script version 0); otherwise value 0 to the script `OP_TRUE` .
Next output	Value 0, script `6a 0c` $\|\| \mathrm{LE}_{32}(\text{next}) \|\| \mathrm{LE}_{64}(r)$: `OP_RETURN` with a 12-byte push of the height and a fresh random 64-bit extra nonce r, which makes every template's merkle root unique.
Last output	Work subsidy plus F, paid to the mining address's payment script (or `OP_TRUE` , version 0, when no address is given).
Height 1	When `BlockOneLedger` is non-empty, the coinbase instead has version 1, `ValueIn` = the block-one subsidy, and one output per ledger entry (Chapter 11).

The treasurybase (treasury agenda active) has version 3; one input with the same null outpoint, sequence, `BlockHeight` , `BlockIndex` as the coinbase, an empty signature script and `ValueIn` equal to the treasury subsidy; output 0 paying the treasury subsidy to the one-byte script `OP_TADD` (`c1`), script version 0; and output 1 of value 0 with the same 12-byte `OP_RETURN` height-and-random-nonce script as the coinbase (with its own random nonce).

17.5.7 Header Fields

Table 17.5 Header of a block template

Field	Value
`Version`	11 on the main network (`generatedBlockVersion`), 12 on all other networks (`generatedBlockVersionTest`).
`PrevBlock`	The chosen parent (Section 17.5.1).
`MerkleRoot`	With the `headercommitments` agenda (DCP-0005) active for the block: the combined root of both trees; otherwise the regular tree's merkle root (Chapter 4).
`StakeRoot`	With `headercommitments` active: the version 1 commitment root over the block's version 2 compact filter (Chapter 16); otherwise the stake tree's merkle root.
`VoteBits`	Below `StakeValidationHeight` : `0x0001` . Otherwise `0x0001` if strictly more than half of the included votes set bit 0 of their vote bits (approve the parent's regular tree), else `0x0000` . No other bits are set.
`FinalState` , `PoolSize` , `SBits`	The next-block values from the chain state (Chapter 8, Chapter 9).
`Voters` , `FreshStake` , `Revocations`	The numbers of votes, tickets and revocations in the stake tree.
`Bits`	The required proof-of-work difficulty for a block extending the parent with this timestamp (Chapter 9).
`Height`	next.
`Size`	The serialized size of the complete block.
`Timestamp`	See below.
`Nonce` , `ExtraData`	Zero; varied by the miner.
`StakeVersion`	The stake version expected for the block after the parent (Chapter 13).

Timestamp. The generator takes the network-adjusted time (local clock corrected by the median offset reported by peers), raises it to one second after the tip's median time if it is earlier, and then subtracts `--miningtimeoffset` seconds (default 0; positive values move the timestamp into the past). The header stores

whole seconds. Miners refresh the timestamp the same way while they work (Section 17.7.1, Section 17.7.5); `Bits` is not recomputed when they do.

Finally the complete block is validated against every consensus rule except the proof of work (Chapter 10); a template that fails is discarded and an error recorded.

17.5.8 Replacement Template When Votes Are Missing

If the next block is at or above `StakeValidationHeight` and no block of the tip generation has 3 votes, or too few votes survive into the stake tree, the generator by default ("aggressive mining") builds a **replacement for the tip**: a sibling of the tip, at the tip's height and extending the tip's parent, which already has votes. With `--nonaggressive` it produces no template instead. The replacement:

- starts from a copy of the tip's header, so every field not listed here (including `Version`, `FinalState`, `Voters`, `FreshStake`, `Revocations`, `PoolSize`, `SBits`, `StakeVersion`, `Nonce` and `ExtraData`) keeps the tip's value;
- has a new coinbase built for the tip's height and the tip's `Voters`, with a fresh extra nonce and **no fees added** to the proof-of-work output; its version, treasury output and subsidy split are selected by the agenda states for the block after the tip (not after the tip's parent);
- if the treasury agenda is active for the block after the tip, has a new treasurybase for the tip's height and `Voters`;
- contains all of the tip's other regular transactions and all of its other stake transactions, in the tip's order (the tip's first stake transaction is skipped as its treasurybase whenever the treasury agenda is active for the block after the tip);
- has a fresh `Timestamp` (as above), recomputed `Size`, `MerkleRoot` and `StakeRoot` (using the agenda state for the block after the tip's parent), and `Bits` recomputed for a block extending the tip's parent.

It is fully validated as a block connecting to the tip's parent. Because the coinbase differs, the replacement has a different hash from the tip and competes with it; it can win only if the tip never collects enough votes.

17.6 Background Template Generation

The background generator keeps one current template and regenerates it in response to chain events, votes and new transactions. Its aim is to start work on a new tip as soon as it can be mined, to include as many votes as possible without waiting long, and to prefer the first block seen at a height so that a miner cannot gain an advantage by withholding the votes on a competitor's block. Generation runs asynchronously; starting a new generation cancels any in progress. While a generation triggered by a new parent or new votes is running, requests for the current template block until it finishes.

No templates are generated while the node is not synced unless `--allowunsyncedmining` is given (refused on the main network, always on for the simulation and regression networks), nor during a chain reorganization: when a reorganization starts the current template is cleared and requests block; when it ends the new tip is handled as a newly connected block. At startup (after waiting for sync if required) the tip is likewise handled as newly connected.

Table 17.6 Template regeneration triggers. $V(B)$ is the number of votes on block B in the vote tracker; $V_{\max}$ = `TicketsPerBlock` (5); $V_{\min} = \left\lfloor \frac{V_{\max}}{2} \right\rfloor + 1$ (3).

Event	Reaction
New tip B connected, $\text{height}(B) + 1 <$ StakeValidationHeight	Generate immediately on B.
New tip B connected, $V(B) \geq V_{\max}$	Generate immediately on B.
New tip B connected, $V_{\min} \leq V(B) < V_{\max}$	Lock B in as the base; start a 2.5 s timer (`maxVoteTimeoutDuration`). Generate when it fires, or earlier when $V(B)$ reaches $V_{\max}$.
New tip B connected, $V(B) < V_{\min}$	Keep the current template (normally built on B's parent, so that it competes with B); stop the regeneration timer; start a 3 s timer (`minVotesTimeoutDuration`).

Event	Reaction
Vote on such a waiting tip B	When $V(B) \geq V_{\min}$: lock B in, cancel side-chain tracking, then as for a newly connected tip with that many votes.
3 s timer fires, B still short of votes	Consider B's siblings in the tip generation in descending order of votes: force a reorganization to the first with $V \geq V_{\min}$ for which it succeeds, and build on it once the reorganization completes. Otherwise start watching the siblings with fewer than $V_{\min}$ votes; if no base block is set for the next template (at startup and after a reorganization), generate on B's parent (which yields the replacement template of Section 17.5.8); else restart the 30 s timer.
Block accepted that is a sibling of a tip still short of votes (after the 3 s timer)	Watch it for votes; when it reaches $V_{\min}$, force a reorganization to it.
Vote on the base block of the current template	If the template already has $V_{\max}$ votes, nothing. Otherwise regenerate immediately with the extra vote, except while the 2.5 s timer is pending, when only reaching $V_{\max}$ triggers.
Template generated	Record the time; (re)start a 30 s timer (`templateRegenSecs`).
30 s timer fires	If the pool changed (to the second) since the template was generated, regenerate to include new transactions; otherwise check again every second.
Generation failed	Retry after 1 s.
Tip disconnected	Generate immediately on the new tip (its votes were in the disconnected block).
`regentemplate` RPC	Regenerate immediately, unless the 2.5 s timer is pending.

Subscribers (the CPU miner and websocket `notifywork` clients) receive each new template with a reason: *new parent* (also used for the first template on a parent even if triggered by a vote), *new votes* or *new transactions*. A new subscriber immediately receives the current template. Notifications are dropped for subscribers that fall more than $2(V_{\max} - V_{\min} + 1)$ notifications behind.

17.7 Mining Interfaces

dcrd provides work to external miners through the `getwork` RPC and its websocket counterpart `notifywork`; it has no `getblocktemplate` RPC. Solved blocks can also be submitted whole with `submitblock`.

17.7.1 getwork Data

`getwork` without arguments returns an object with two hex strings, `data` and `target`. `data` is the 180-byte serialized block header (Chapter 4), **exactly as serialized**, with every integer little-endian and no per-word byte swapping, followed by 12 bytes of padding, for a total of 192 bytes (both `getworkDataLenBlake256` and `getworkDataLenBlake3` are 192). Table 17.7 gives the offsets.

Table 17.7 Layout of getwork `data` (offsets in bytes)

Offset	Size	Field	Notes
0	4	`Version`	
4	32	`PrevBlock`	
36	32	`MerkleRoot`	With `StakeRoot`, identifies the template on submission.
68	32	`StakeRoot`	
100	2	`VoteBits`	
102	6	`FinalState`	
108	2	`Voters`	
110	1	`FreshStake`	
111	1	`Revocations`	
112	4	`PoolSize`	
116	4	`Bits`	

Offset	Size	Field	Notes
120	8	`SBits`	
128	4	`Height`	Start of the third 64-byte block.
132	4	`Size`	
136	4	`Timestamp`	May be updated by the miner.
140	4	`Nonce`	Varied by the miner.
144	32	`ExtraData`	Extra nonce space; dcrd's own miner uses the first 8 bytes.
176	4	`StakeVersion`	
180	12	padding	See below; not part of the hashed message.

The padding makes the last 64 bytes of `data` exactly the final message block of the hash function in use, so a miner can compute the state after the first two blocks (bytes 0–127) once and iterate only on the final block, which contains `Timestamp`, `Nonce` and `ExtraData`. Which hash is used is decided by whether the `blake3pow` agenda (DCP-0011) is active for the block after `PrevBlock`:

- **BLAKE-256** (agenda inactive): the padding is the BLAKE-256 message padding for a 1440-bit message, `800000010000000000005a0`: the byte `80`, two zero bytes, the byte `01` (the final 1 bit of BLAKE padding), and the bit length 1440 as a big-endian 64-bit integer. The final block is processed with counter 1440.
- **BLAKE3** (agenda active): the padding is 12 zero bytes. BLAKE3 hashes the 180-byte header as a single chunk of three blocks; the final block must be compressed with a block length of 52 (not 64), counter 0 and the chunk-end and root flags, the zero padding being the zero fill BLAKE3 applies to a short block.

In both cases the proof-of-work hash is computed over the 180 header bytes only, as specified in Chapter 9; the padding is a convenience and is not hashed as message data.

`target` is the 256-bit target encoded by `Bits` (Chapter 9), written as 32 bytes **little-endian**. A hash solves the work when, read as a little-endian 256-bit integer, it is less than or equal to the target.

IMPLEMENTATION PITFALL
Source comments in dcrd's `serializeGetWorkData` list `Bits`, `Timestamp`, `Nonce` and the extra nonce at offsets one byte lower than the ones the serialization actually produces. Table 17.7 gives the offsets of the serialized data, which agree with the offsets the CPU miner writes to (136, 140 and 144).

TEST VECTOR getwork data for the main-network genesis header (BLAKE-256 padding)

```
  0  01000000000000000000000000000000000000000000000000000000000000000000000000
 32  000000000dc101dfc3c6a2eb10ca0c5374e10d28feb53f7eabcc850511ceadb9
 64  9174aa66000000000000000000000000000000000000000000000000000000000000000000
 96  00000000000000000000000000000000000000000000ffff011b00c2eb0b00000000
128  0000000000000000a0d7b856000000000000000000000000000000000000000000000000
160  00000000000000000000000000000000000000000000800000010000000000005a0

With BLAKE3 padding the last 12 bytes are 000000000000000000000000.
target (Bits 1b01ffff):
     00000000000000000000000000000000000000000000ffff010000000000
```

Serving work. `getwork` fails if the CPU miner is running, if no mining address is configured, if the node has no peers or is not synced (both unless unsynced mining is allowed; the sync check is also skipped while both the best header and the best block are at height 0; these first four checks apply to submissions as well), or, for a request, if the current background template is in an error state. Requests and submissions are serialized. When the tip has changed since the previous request, templates whose height is below tip height − 3 (`getworkExpirationDiff`) are discarded and the request waits for a fresh template: it takes the next template notification (the current template is delivered at once), and if that template has already been handed out, waits up to a further 5.5 s for a newer one before falling back to the current template. The header of the chosen

template is copied, its timestamp refreshed as in Section 17.5.7, serialized as above, and the template is stored in a table keyed by the 64 bytes `MerkleRoot` ‖ `StakeRoot`. Because the coinbase extra nonce makes the merkle root unique per template, refreshing the timestamp does not create new entries.

17.7.2 Work Submission

`getwork` with a `data` argument submits a solution:

1. The hex string, after prefixing a `0` if its length is odd, must decode to exactly 192 bytes; otherwise an invalid-parameter error is returned.
2. The first 180 bytes are deserialized as a header; the padding is ignored.
3. If `PrevBlock` is unknown the result is `false`.
4. The proof-of-work hash (BLAKE-256 or BLAKE3 as above) must satisfy the proof-of-work check against the header's own `Bits` and the network's proof-of-work limit; otherwise `false`.
5. The template is looked up by `MerkleRoot` ‖ `StakeRoot`; if absent, `false`.
6. The block is reconstructed from the template's transactions and the **submitted** header in its entirety (so every header field is taken from the miner), and processed exactly like a block received from a peer. The result is `true` if it is accepted and `false` if consensus rejects it.

17.7.3 notifywork

A websocket client that has called `notifywork` receives a `work` notification for every new background template, with the same `data` and `target` encodings and a `reason` of `newparent`, `newvotes`, `newtxns` or `unknown`. The data carries the template's own timestamp (not refreshed). Each notified template is stored in the same submission table, which is pruned as above whenever the reason is `newparent`. `stopnotifywork` ends the subscription.

17.7.4 Other Mining RPCs

Table 17.8 Mining-related RPC methods

Method	Behaviour
`submitblock`	Accepts a fully serialized block in hex (a leading `0` is added to odd-length input) and processes it like a block from a peer; returns null on success or the string `rejected:` followed by the reason.
`regentemplate`	Asks the background generator to regenerate (Table 17.6).
`generate`	Mines the requested number of blocks with the CPU miner and returns their hashes. Refused on networks with `GenerateSupported` false (the main network).
`setgenerate`, `getgenerate`	Start or stop the CPU miner with a given number of workers (0 stops it); report whether it runs.
`gethashespersec`, `getmininginfo`, `getnetworkhashps`	Informational.
`estimatefee`	Returns the configured minimum relay fee rate in DCR/kB, irrespective of the argument.
`estimatesmartfee`	Returns the fee estimator's result (Section 17.8) for a confirmation target; only the `conservative` mode is supported.

17.7.5 CPU Miner

The built-in CPU miner, intended for test networks, subscribes to the background generator and works on each new template, abandoning the previous one. A worker serializes the header once, picks a random 64-bit offset, and for each extra nonce value (offset plus a counter, wrapping) writes it little-endian into `ExtraData` [0:8] (offset 144) and tries every nonce from 0 to $2^{32} - 1$ (offset 140), hashing with BLAKE-256 or BLAKE3 as the `blake3pow` agenda dictates and accepting a hash less than or equal to the target. Every 65535 nonces it checks for cancellation and refreshes the timestamp (offset 136). Without peers it waits, except on the simulation and regression networks, where it also stops after 4 solved blocks on the same parent fail to be

accepted, so that a stalled simulated chain is not mined pointlessly. It runs one worker by default, at most twice the number of CPU cores.

17.8 Fee Estimation

The fee estimator (`internal/fees`) answers `estimatesmartfee`: the fee rate that, judged by recent history, gets a transaction confirmed within a target number of blocks with high probability. It is purely advisory and uses floating-point arithmetic throughout.

Buckets. Fee rates are divided into buckets whose upper bounds start at the minimum relay fee rate r and grow by a factor of 1.1 (`DefaultFeeRateStep`) while below $100r$ (`DefaultMaxBucketFeeMultiplier`). An extra bound of 100000 atoms/kB (the default minimum relay fee of dcrd releases up to 1.1.0) is inserted where the sequence crosses it, and a last bound of $+\infty$ catches everything else. With the default $r = 10000$ this gives 51 buckets: 10000, 11000, 12100, ..., 98497.33, 100000, 108347.06, ..., 970172.34, $+\infty$. A rate falls in the first bucket whose bound is at least the rate. Each bucket keeps, for each of 32 confirmation ranges (`DefaultMaxConfirmations`; range c means "confirmed within $c + 1$ blocks", the last range meaning "32 or more"), a decayed count of confirmed transactions and the sum of their rates, plus the same data for transactions still in the pool.

Tracking. The estimator starts when the first block is accepted after startup while the node is synced (or unsynced mining is allowed). A transaction entering the pool is recorded if it has not been recorded before, is not a treasury spend, and its rate $\left\lfloor \frac{f}{s} \right\rfloor \cdot 1000$, computed with integer division **before** the multiplication, is at least r; so recorded rates are multiples of 1000 atoms/kB, and votes (fee 0) are never recorded. A recorded transaction is placed in range 0 of its bucket's pool data, together with the estimator's current height (the height of the last block it processed, or of the block that started it).

Per block. When a block is connected at a height above the last processed one (reorganizations are not tracked): every confirmed statistic is multiplied by 0.998 (`defaultDecay`); the pool data of every bucket shifts up one range, the next-to-last range being added into the last; and each transaction of the block that was recorded is moved from the pool data to the confirmed data, counted in every range from $d - 1$ (capped at 31) upward, where d is the block height minus the height at which it entered the pool. Unrecorded transactions are ignored, so a miner cannot skew the estimate with transactions it never broadcast. Transactions leaving the pool unmined are removed from the pool data.

Estimate. For a target of k blocks ($1 \le k \le 32$), with range index $c = k - 1$ and required success ratio 0.95: walk the buckets from the highest down, accumulating the total (all confirmed transactions of the bucket plus those still pending in range c) and the number confirmed within range c. Whenever the accumulated total exceeds 1, close a group: if its success ratio is below 0.95, stop (failing if no group has passed yet); otherwise remember it as the best group and start a new one. If no group is ever closed, the best group is the highest ($+\infty$) bucket alone. The estimate is the average rate (rate sum divided by count) of the bucket containing the median confirmed transaction of the best — that is, the lowest-fee passing — group (failing if that group has no confirmed transactions), rounded to the nearest atom and raised to r if lower. The estimator's state, but not its pool data, persists across restarts in a `feesdb` database in the data directory.

17.9 Policy Constants

Table 17.9 Policy constants of the reference node. Per-network values are main-network values; options are command-line or configuration settings.

Name	Value	Meaning
Mempool (`internal/mempool`)		
`MaxStandardTxSize`	100000 bytes	Largest standard transaction; also `MaxOrphanTxSize`.
`maxStandardSigScriptSize`	1650 bytes	Largest standard signature script.
`maxStandardP2SHSigOps`	15	Sigops allowed in a standard P2SH redemption.
`maxStandardMultiSigKeys`	3	Keys allowed in a standard bare multisig output.

Name	Value	Meaning
`maxNullDataOutputs`	4	Null-data outputs allowed in a standard regular transaction.
`MaxDataCarrierSizeV0`	256 bytes	Largest standard null-data push (`txscript/stdscript`).
`DefaultMinRelayTxFee`	10000 atoms/kB	Default `--minrelaytxfee` (0.0001 DCR/kB); also the template fee floor and dust basis.
dust input size, divisor	165 bytes, 3	Constants of Eq. (17.1).
`maxRelayFeeMultiplier`	10000	High-fee limit factor (1 DCR/kB by default).
`MaxSigOpsPerTx`	1000	`MaxSigOpsPerBlock` / 5.
`BaseStandardVerifyFlags`	—	Discourage upgradable NOPs, clean stack, CLTV, CSV.
`AcceptNonStdTxs`	false	Per network; true on testnet, simnet, regnet.
`defaultMaxOrphanTransactions`	100	Default `--maxorphantx` .
`orphanTTL`	15 min	Orphan lifetime.
`orphanExpireScanInterval`	5 min	Minimum time between orphan expiry scans.
`maxFutureVoteAge`	5	Blocks beyond the tip a vote may reference.
`MaxVoteAge`	`CoinbaseMaturity` (256)	Vote age window; 1440 on testnet (`defaultMaximumVoteAge`).
`defaultAllowOldVotes`	false	Default of `--allowoldvotes` .
`maxVoteDoubleSpends`	5	Votes per ticket in the pool.
`heightDiffToPruneTicket`	288	Ticket age (blocks) before pruning.
`heightDiffToPruneVotes`	10	Vote and revocation age before pruning; also tracker pruning.
`MempoolMaxConcurrentTSpends`	7	Treasury spends in the pool.
tspend start threshold	$2TM$ (6912)	Maximum distance to a treasury spend's voting window.
`maxRejectedTxns`	62500	Capacity of the rejected-hash filter (`internal/netsync`), false-positive rate 10^{-7}.
`maxReorgDepthNotify`	6	Reorganization depth limit for winning-ticket notifications.
Block templates (`internal/mining` , `config.go`)		
`generatedBlockVersion`	11	Template block version, main network.
`generatedBlockVersionTest`	12	Template block version, other networks.
`defaultBlockMaxSize`	375000 bytes	Default `--blockmaxsize` .
`blockMaxSizeMin`	1000 bytes	Smallest permitted `--blockmaxsize` .
`blockMaxSizeMax`	`MaximumBlockSizes`[0] −1000	Largest permitted `--blockmaxsize` (392216).
`blockHeaderOverhead`	189 bytes	`MaxBlockHeaderPayload` (180) + `MaxVarIntPayload` (9).
`coinbaseFlags`	`/dcrd/`	Coinbase signature script tag (after two zero bytes).
`ancestorTrackingLimit`	25	Ancestor/descendant limit for ancestor statistics.
`UnminedHeight`	`0x7fffffff`	Height assigned to outputs of unconfirmed transactions.
`MaxTAddsPerBlock`	20	Consensus limit, enforced during selection.
`MaxSigOpsPerBlock`	5000	Consensus limit, enforced during selection.
minimum votes	$\left\lfloor \frac{\text{TicketsPerBlock}}{2} \right\rfloor + 1$ (3)	Votes needed to build on a block.
`MiningTimeOffset`	0 s	Default `--miningtimeoffset` .
aggressive mining	on	Replacement templates unless `--nonaggressive` .
`defaultAllowUnsyncedMining`	false	Forced on for simnet and regnet; refused on mainnet.

Name	Value	Meaning
Background generator		
`minVotesTimeoutDuration`	3 s	Wait for the minimum votes before considering siblings.
`maxVoteTimeoutDuration`	2.5 s	Wait for further votes after the minimum is reached.
`templateRegenSecs`	30 s	Regeneration interval for new transactions.
retry and recheck delay	1 s	After a failed generation; between pool-change checks.
Mining interfaces (`internal/rpcserver`, `internal/mining/cpuminer`)		
`getworkDataLenBlake256`, `getworkDataLenBlake3`	192 bytes	Length of getwork `data`.
`getworkExpirationDiff`	3	Templates below tip height − 3 are discarded.
`maxTemplateTimeoutDuration`	5.5 s	Wait for a fresh template after a tip change.
`merkleRootPairSize`	64 bytes	Submission table key.
`maxNonce`	$2^{32} - 1$	CPU miner nonce range.
`maxSimnetToMine`	4	Failed CPU-mined blocks per parent before stopping (simnet, regnet).
`defaultNumWorkers`	1	Default CPU miner workers; maximum 2 × CPU cores.
Fee estimation (`internal/fees`, `server.go`)		
`DefaultMaxConfirmations`	32	Confirmation ranges.
`DefaultFeeRateStep`	1.1	Ratio between bucket bounds.
`DefaultMaxBucketFeeMultiplier`	100	Finite bucket bounds stay below $100r$.
`ExtraBucketFee`	100000 atoms/kB	Additional tracked bound.
`defaultDecay`	0.998	Per-block decay of confirmed statistics.
success ratio	0.95	Required by `EstimateFee`.
`maxAllowedBucketFees`, `maxAllowedConfirms`	2000, 788	Sanity limits on the estimator database.

CHAPTER 18

StakeShuffle Mixing

> **NOTE**
> **Status: not consensus.** Nothing in this chapter affects the validity of a block or transaction. A completed mix is an ordinary regular-tree transaction validated by the rules of Chapter 7. StakeShuffle is nevertheless an *interoperability* protocol: a client that deviates from any byte layout, hash preimage, ordering or key-derivation step described here cannot complete a mix with dcrd and dcrwallet peers, and may be blamed and excluded as a disruptor. The chapter therefore uses **MUST** and **SHOULD** in their RFC 2119 sense for interoperability, and labels relay-node behavior as *policy*. No rule boxes appear in it.

StakeShuffle is Decred's peer-to-peer coin-mixing protocol. Wallets that wish to break the on-chain link between their coins and their future spending cooperate to build a single *CoinJoin* transaction in which every participant contributes inputs and receives one or more outputs of an identical, agreed value. The protocol makes sure that no participant, relay node or outside observer learns which participant owns which of the equal-valued outputs. It is an implementation of CoinShuffle++ [49], more precisely of DiceMix Light [50], a simpler protocol inspired by the DiceMix of that paper, in which a separate slot-reservation phase precedes an XOR DC-net. It is carried as eight new message types over the ordinary Decred peer-to-peer network (Chapter 15), with no coordinating server. Full nodes relay the messages after checking them against a *mixpool* policy that resists denial of service.

The reference implementation is the `mixing` module of dcrd: the packages `mixing` (cryptographic primitives), `mixing/mixclient` (the participant state machine used by dcrwallet [22]), `mixing/mixpool` (the message pool used by both relays and wallets) and `mixing/utxoproof`, together with the message encodings in `wire/msgmix*.go`.

18.1 Overview

18.1.1 What is mixed

Each participant, called a *peer* in this chapter, contributes one or more unspent outputs it controls, asks for $m \geq 1$ fresh outputs of the common *mix amount* (its *message count*), and may ask for one change output. The final transaction contains every peer's inputs, every peer's change output, and $\sum m$ outputs of exactly the mix amount, each paying a pay-to-pubkey-hash (P2PKH) script to a key freshly generated by its owner. Only the mixed outputs are anonymized: inputs and change are publicly linked to the peer that contributed them. Amounts are public; equality of the mixed outputs is what makes them indistinguishable.

dcrwallet uses mixing in two ways (informative): to mix ordinary funds into a "mixed" account in standard denominations, and in its ticket buyer, which mixes the outputs of the *split transaction* that funds ticket purchases (Chapter 8) so that a ticket cannot be traced to the coins that paid for it. Both produce the same kind of CoinJoin; the denominations a wallet chooses are wallet policy and are not specified by dcrd, but peers can only mix together when their mix amounts are exactly equal (Section 18.5).

18.1.2 Threat model and guarantees

The guarantees are those of CoinShuffle++ [49]:

- **Unlinkability.** An adversary who controls the network and all but two of the peers of a successful run cannot tell which honest peer owns which honest mixed output with probability better than guessing. The anonymity set of an output is the set of honest peers in the final run. The adversary does learn the full set of inputs, change outputs and mixed outputs.
- **Termination.** A malicious peer can force a run to fail, but every failure either identifies the misbehaving peers to all honest peers, publicly and verifiably, or is attributable to a timeout. The honest peers then rerun without them. The papers assume an honest bulletin board or reliable broadcast channel and at least two honest peers for termination; dcrd floods messages through the peer-to-peer network and gives up below `MinPeers` peers. With f disruptors, at most about $f + 1$ runs are needed, since every failed run excludes at least one of them; DiceMix Light states its cost as $4 + 3f$ communication rounds [50].
- **No custody.** Each peer signs only the final transaction, after checking that its own outputs are present. Nothing can be stolen.

Two departures from those designs strengthen it. Pairwise keys combine elliptic-curve Diffie-Hellman with the post-quantum key-encapsulation mechanism Streamlined NTRU Prime 4591^{761} [51], so that a future adversary able to break secp256k1 from a recorded transcript still faces the lattice scheme ("If sntrup4591761 is discovered to be broken in the future, the security only reduces to that of ECDH", in the words of the source). And every peer commits, in its first session message, to all secrets it will ever need to reveal, so that blame assignment cannot be gamed by choosing secrets after the fact. DiceMix Light commits only to the messages to be mixed, through an additional DC-net over a group sent with the slot reservation; its other secrets follow deterministically from the key-exchange secret.

The protocol does not hide network-level metadata. Messages are flooded through the peer-to-peer network, so a node directly connected to a participant may infer that a message originated there; random delays ("jitter") on every message reduce, but do not remove, timing correlation. Reusing an address, or spending change together with mixed outputs, undoes the mix; that is wallet policy.

18.1.3 Protocol at a glance

Time is divided into *epochs* of fixed length (ten minutes on the main network). Between epoch boundaries, peers broadcast *pair requests* (PR) describing what they want to mix and proving that they own the inputs. At each epoch boundary every waiting peer independently collects the compatible pair requests it knows of, derives a *session identifier* from them, and runs the protocol of Table 18.1. A *run* is one attempt; if a run fails, the survivors start a new run, which in this implementation is always a new session with its own identifier.

Table 18.1 Messages of a StakeShuffle session, in protocol order

Abbr.	Command	Sent when	Purpose
PR	`mixpairreq`	before epoch	Describe the requested mix, list inputs with ownership proofs; no session yet.
KE	`mixkeyxchg`	run start	Fix the session (PR list and order), publish ECDH and KEM public keys and a commitment to all secrets.
CT	`mixcphrtxt`	after all KEs	One KEM ciphertext to every peer (including self).
SR	`mixslotres`	after all CTs	Slot reservation: padded power sums over the field $\mathbb{F}_F$.
FP	`mixfactpoly`	with DC	Roots of the slot-reservation polynomial, published by peers able to factor it.
DC	`mixdcnet`	after roots	XOR DC-net vectors carrying the 20-byte output hashes in their reserved slots.
CM	`mixconfirm`	after DC	The CoinJoin with the sender's input signatures.
RS	`mixsecrets`	on failure	Reveal the committed secrets for blame assignment.

18.2 Primitives and Notation

The following primitives are used. Byte strings, `||`, `le32`, `be64` and friends follow Chapter 2; `varint`, `varbytes` and the transaction serialization follow Chapter 4.

BLAKE-256 The 32-byte hash of Chapter 3, written $H(\cdot)$. Every hash in this chapter is a single BLAKE-256, never the double hash.

HASH160 RIPEMD-160 of BLAKE-256, as defined in Chapter 3.

secp256k1 and EC-Schnorr-DCRv0 The curve and the 64-byte Schnorr signature scheme of Chapter 3. Public keys in mixing messages are 33-byte compressed SEC encodings.

Field All slot-reservation arithmetic is in the prime field $\mathbb{F}_F$, where

$$F = 2^{127} - 1 = 170141183460469231731687303715884105727 \tag{18.1}$$

(`F`, hexadecimal `7fff...ff`, 16 bytes). "x mod F" always denotes the non-negative residue. Field elements are serialized as *minimal big-endian* byte strings (no leading zero bytes; zero is the empty string) inside `varbytes`, at most `MaxMixFieldValLen` $= 32$ bytes. A received value is parsed as an unsigned big-endian integer regardless of leading zeros.

ChaCha20 PRNG PRNG(k, n) is the keystream of the IETF ChaCha20 stream cipher [52] in the 96-bit-nonce, 32-bit-counter variant of RFC 8439 [53], with 32-byte key k, 12-byte nonce `le32(n) || 00 00 00 00 00 00 00 00` and initial block counter 0. Reading ℓ bytes returns the next ℓ keystream bytes; reads are consecutive. `mixing/internal/chacha20prng/prng.go · New`

sntrup4591761 The key-encapsulation mechanism of Section 18.2.1.

18.2.1 The sntrup4591761 KEM

The KEM is *Streamlined NTRU Prime* 4591^{761} as specified in the NTRU Prime paper [51], in the parameters of its August 2017 revision. dcrd uses a Go port of the reference C code of that paper, the package

`github.com/companyzero/sntrup4591761 v0.0.0-20220309191932-9e0f3af2f07a`

and an implementation **MUST** be bit-compatible with it. Its parameters are $p = 761$, $q = 4591$ and $t = 143$ (the scheme's own names; this p is unrelated to the peer p of later sections), so a *t-small* polynomial has exactly $2t = 286$ non-zero coefficients in $\{-1, 0, 1\}$. Sizes are fixed:

Table 18.2 sntrup4591761 object sizes

Object	Bytes	Content
Public key	1218	Encoding of $h = g/(3f)$ in $\mathcal{R}/q$.
Private key	1600	`Encode(f)` (191) `||` `Encode(1/g mod 3)` (191) `||` public key (1218).
Ciphertext	1047	First 32 bytes of SHA-512(Encode(r)) (confirmation) `||` rounded encoding of hr (1015).
Shared key	32	Last 32 bytes of SHA-512(Encode(r)).

Decapsulation reports failure unless the recovered r has weight 286, its re-encryption equals the ciphertext and the confirmation matches.

> **IMPLEMENTATION PITFALL**
> This is the 2017 "sntrup4591761" scheme, **not** the later NIST-submission "sntrup761" (whose keys and ciphertexts have different sizes and hashing). Implementations from current cryptographic libraries are not compatible.

Because blame assignment (Section 18.8) recomputes another peer's keys and ciphertexts from its revealed seed, the *consumption of randomness* must be reproduced exactly, not only the scheme:

- Each random 32-bit word is 4 bytes read from the source, interpreted as a little-endian `int32`.
- *Key generation*: repeat { read 761 words r_i and set $g_i = (((r_i \text{ AND } 0\text{x}3\text{fffffff}) \times 3) \gg 30) - 1$ } until g is invertible modulo 3; then sample f as a t-small element (below).
- *Encapsulation*: sample r as a t-small element.
- *t-small sampling*: read 761 words w_i; for $i < 286$ set $w_i \leftarrow w_i \text{ AND } (-2)$, otherwise $w_i \leftarrow (w_i \text{ AND } (-3)) \text{ OR } 1$; reorder w with the library's fixed sorting network (Algorithm 18.1, a port of the reference implementation's); set coefficient i to $(w_i \text{ AND } 3) - 1$.

ALGORITHM 18.1 sntrup4591761 sorting network

```
def sort761(w, le):              # w: 761 int32 words; le: see pitfall
    n, top = 761, 1
    while top < n - top:
        top += top               # top = 512
    def minmax(i, j):            # exchange so that the "smaller" word is at i
        if le(w[j], w[i]):
            w[i], w[j] = w[j], w[i]
    p = top
    while p > 0:
        for i in range(n - p):
            if i & p == 0: minmax(i, i + p)
        q = top
        while q > p:
            for i in range(n - q):
                if i & p == 0: minmax(i + p, i + q)
            q >>= 1
        p >>= 1
```

dcrd: github.com/companyzero/sntrup4591761/zx/random.go · sort, minmax

IMPLEMENTATION PITFALL

`minmax` evaluates Go's `crypto/subtle.ConstantTimeLessOrEq(int(uint32(a)), int(uint32(b)))`, whose result for words of 2^{31} or more depends on the Go release and the target's `int` width:

- Go 1.25 and earlier: `le(a, b)` is the sign bit of the wrapping 32-bit difference `int32(a) - int32(b) - 1`. This is not an order relation, and the result depends on the exact network above.
- Go 1.26 and later, 64-bit targets: `le(a, b)` is `uint32(a) <= uint32(b)`, so the network sorts the words into ascending unsigned order (any correct sort gives the same result).
- Go 1.26 and later, 32-bit targets: `int(uint32(a))` wraps, `le(a, b)` is `int32(a) <= int32(b)`, and the words end up in ascending signed order.

The same seed therefore yields different sntrup4591761 public keys and ciphertexts in the three cases. This was observed with Go 1.24.13, 1.25.0 and 1.25.14 on amd64 (1.24.13 and 1.25.14 also on 386), Go 1.26.0, 1.26.11 and 1.27.4 on amd64, and Go 1.26.0, 1.26.11 and 1.27.4 on 386 (Section 18.11); for 1000 random inputs the library's t-small output matched Algorithm 18.1 with the comparison listed for each case every time. Normal mixing is unaffected, since any valid key pair works. In blame assignment, however, each peer recomputes the others' public keys from their revealed seeds with its own build (Section 18.8.4, step 1); when the two builds fall in different cases above, the recomputed key differs from the KE's and the verifier blames a peer that followed the protocol. A new implementation **SHOULD** use the unsigned comparison (ascending unsigned sort) of current 64-bit builds.

18.3 Identities, Hashes and Signatures

18.3.1 Ephemeral identities

Every mixing message carries the sender's *identity*: a 33-byte compressed secp256k1 public key generated freshly for each pair request and discarded after the mix. A peer **MUST** use a new identity for every pair request; the mixpool rejects a pair request from an identity that has already sent any message (Section 18.9). All later messages of the peer in every session formed from that pair request are signed by the same identity.

18.3.2 Message hash

Each mixing message is identified by its *message hash*, the BLAKE-256 of its canonical serialization (the full payload of Section 18.4, signature first). Message hashes are the inventory identifiers used for relay and the references in "seen" lists. They are always used in internal byte order, not reversed for display.

18.3.3 Signatures

Every message, including the pair request, is signed with EC-Schnorr-DCRv0 by the identity key. The signed digest binds the message contents, its type, its session and its run, using an ASCII domain-separation string.

ALGORITHM 18.2 Mixing message signature digest

```
def signed_data_hash(msg) -> bytes:
    # varstring of the command with "-sig" appended (e.g. "mixkeyxchg-sig"),
    # then every field of the message except the leading 64-byte signature,
    # serialized exactly as on the wire
    label = (msg.command + "-sig").encode("ascii")
    return BLAKE256(varint(len(label)) + label + serialize_without_signature(msg))

def signature_digest(msg) -> bytes:
    sid, run = msg.session_id, msg.run
    if msg.command == "mixpairreq":          # has no session
        sid, run = bytes(32), 0
    s = (b"decred-mix-signature" + b"," + msg.command.encode("ascii") + b"," +
         hex_lower(sid) + b"," + str(run).encode("ascii") + b"," +
         hex_lower(signed_data_hash(msg)))
    return BLAKE256(s)

def sign(msg, priv):
    msg.signature = ec_schnorr_dcrv0_sign(priv, signature_digest(msg))   # 64 bytes

def verify(msg) -> bool:
    return (len(msg.identity) == 33 and parse_pubkey(msg.identity) is not None
            and parse_schnorr_sig(msg.signature) is not None
            and ec_schnorr_dcrv0_verify(msg.identity, signature_digest(msg),
                                        msg.signature))
```

dcrd: mixing/signatures.go · sign, verify, schnorrHash | wire/msgmix*.go · WriteSignedData

`hex_lower` is lowercase hexadecimal of the raw bytes (64 characters for 32 bytes), and the run is written in decimal without leading zeros. Because the session identifier and run are inside the digest, a signature cannot be replayed into another session even if the rest of a message were identical.

> **NOTE**
> dcrd signs with the deterministic nonce of its EC-Schnorr-DCRv0 implementation, which makes the signature vector of Section 18.11 reproducible. Verifiers do not depend on how the nonce was chosen.

18.3.4 Transcript references

Every message after the pair request lists the hashes of the messages of the previous stage that the sender used: the KE lists the session's PRs (`SeenPRs`), the CT lists KEs, the SR lists CTs, the FP and DC list SRs, the CM lists DCs, and an RS lists RSs. Because the lists are inside the signed data, each message commits to the transcript its author saw. The reference implementation reads only two of them: the `SeenPRs` of a KE, which define the session (Algorithm 18.4, Algorithm 18.5), and the `SeenSecrets` of an RS, which are used for blame (Section 18.8.4). For the CT, SR, FP, DC, CM and RS lists the mixpool checks only the number of entries (Table 18.16), and the client never inspects the hashes of the CT, SR, FP, DC and CM lists. The client fills these lists in unmixed order, except `SeenSecrets`, whose order is unspecified. An implementation **SHOULD** fill them exactly as described in Section 18.7. `mixing/mixpool/limits.go · checkCTLimits` | `mixing/mixclient/blame.go · blame`

18.4 Wire Messages

The eight messages are valid only on connections that negotiated protocol version `MixVersion` = 10 or later (Chapter 15); decoding them at a lower version is an error. They are announced and requested with inventory

vectors of type `MSG_MIX` = 4 whose hash is the message hash. All integers are little-endian; "hash" fields are 32 raw bytes. Each table gives the payload of the message, which follows the common message header.

The messages share a prefix: the 64-byte Schnorr signature, the 33-byte identity, and (except in the PR) the 32-byte session ID and the 4-byte run number. Every message produced by the reference client carries run number 0, and the mixpool rejects any other value before all other checks (Section 18.9.2). `mixing/mixpool/mixpool.go · AcceptMessage`

Table 18.3 `mixpairreq` (PR) payload

Field	Type	Bytes	Description
`Signature`	`[64]byte`	64	Signature (Algorithm 18.2).
`Identity`	`[33]byte`	33	Compressed public key of the ephemeral identity.
`Expiry`	`uint32`	4	Block height at which the PR expires (Section 18.5.3).
`MixAmount`	`int64`	8	Value in atoms of each mixed output. Decoding fails if negative.
`ScriptClass`	`varstring`	1+n	Script class of mixed outputs; strict ASCII (bytes `0x20` – `0x7e`), $n \leq$ 32 (`MaxMixPairReqScriptClassLen`).
`TxVersion`	`uint16`	2	Version of the CoinJoin transaction.
`LockTime`	`uint32`	4	Lock time of the CoinJoin transaction.
`MessageCount`	`uint32`	4	Number m of mixed outputs requested.
`InputValue`	`int64`	8	Sum of the values of the contributed UTXOs. Decoding fails if negative.
`UTXOCount`	`varint`	1–3	Number of contributed UTXOs, at most `MaxMixPairReqUTXOs` = 512.
`UTXOs`	`MixPairReqUTXO[]`	var	UTXOCount entries, Table 18.4.
`HasChange`	`uint8`	1	0: no change output; 1: a change output follows. Any other value fails decoding.
`Change`	`TxOut`	var	Present only if HasChange is 1: value (`int64` , 8), script version (`uint16` , 2), public key script (`varbytes`), exactly as a transaction output (Chapter 4).
`Flags`	`byte`	1	Capability flags (Table 18.13).
`PairingFlags`	`byte`	1	Pairing flags; part of the pairing description (Section 18.5.2).

Table 18.4 `MixPairReqUTXO` (one contributed output)

Field	Type	Bytes	Description
`Hash`	`[32]byte`	32	Hash of the transaction containing the output.
`Index`	`uint32`	4	Output index.
`Tree`	`int8`	1	Transaction tree (0 regular, 1 stake). Decoding fails if negative.
`Script`	`varbytes`	var	Redeem script for P2SH outputs, at most 16384 bytes (`MaxMixPairReqUTXOScriptLen`). Relays reject a non-empty script (P2SH inputs are unsupported).
`PubKey`	`varbytes`	var	Public key controlling the output, at most 33 bytes.
`Signature`	`varbytes`	var	UTXO ownership proof (Section 18.5.4), at most 64 bytes.
`Opcode`	`byte`	1	0 for a plain P2PKH output, or the stake opcode tagging the output script: `OP_SSGEN` (`0xbb`), `OP_SSRTX` (`0xbc`) or `OP_TGEN` (`0xc3`).

Table 18.5 `mixkeyxchg` (KE) payload

Field	Type	Bytes	Description
`Signature`	`[64]byte`	64	Signature.
`Identity`	`[33]byte`	33	Sender identity.
`SessionID`	`[32]byte`	32	Session identifier (Algorithm 18.4).
`Epoch`	`uint64`	8	Epoch of the session as Unix time in seconds.
`Run`	`uint32`	4	Run number; 0.

Field	Type	Bytes	Description
Pos	uint32	4	Index of the sender's own PR in SeenPRs (its *unmixed position*).
ECDH	[33]byte	33	Compressed secp256k1 public key for Diffie-Hellman.
PQPK	[1218]byte	1218	sntrup4591761 public key.
Commitment	[32]byte	32	Commitment to the sender's secrets (Section 18.7.3).
SeenPRCount	varint	1–3	Number of PR hashes, at most MaxMixPeers = 512.
SeenPRs	[32]byte[]	32 each	Hashes of the session's PRs in unmixed order (Section 18.6.3).

Table 18.6 `mixcphrtxt` (CT) payload

Field	Type	Bytes	Description
Signature	[64]byte	64	Signature.
Identity	[33]byte	33	Sender identity.
SessionID	[32]byte	32	Session identifier.
Run	uint32	4	Run number; 0.
Count	varint	1–3	Number n of ciphertexts **and** of seen KE hashes (one count for both lists), at most 512. An encoder **MUST NOT** produce lists of different lengths.
Ciphertexts	[1047]byte[]	1047 each	n sntrup4591761 ciphertexts; entry k is encapsulated to the PQPK of the peer at unmixed position k.
SeenKeyExchanges	[32]byte[]	32 each	n KE message hashes in unmixed order.

Table 18.7 `mixslotres` (SR) payload

Field	Type	Bytes	Description
Signature	[64]byte	64	Signature.
Identity	[33]byte	33	Sender identity.
SessionID	[32]byte	32	Session identifier.
Run	uint32	4	Run number; 0.
MCount	varint	1–3	Outer dimension: the sender's message count m; $1 \le \text{MCount} \le 1024$ (MaxMixMcount).
Width	varint	1–3	Inner dimension: the session's total message count M; $1 \le \text{Width} \le 512$ (MaxMixPeers, see the pitfall below).
DCMix	varbytes[][]	var	MCount rows of Width field elements (minimal big-endian, each at most 32 bytes), row-major. Row j is the padded power-sum vector of the sender's j-th message.
SeenCount	varint	1–3	At most 512.
SeenCiphertexts	[32]byte[]	32 each	CT message hashes in unmixed order.

IMPLEMENTATION PITFALL

The SR decoder bounds the inner dimension by MaxMixPeers = 512, and the encoder refuses (with the error "too many mixing peers") to serialize an SR whose first row is longer than 512, although that dimension is the session's total message count M, which the client allows up to MaxMtot = 1024 (Section 18.6.6) and the mixpool accepts up to 1024. In a session with $M > 512$, an SR is therefore accepted into the sender's own mixpool but cannot be written to any connection: a dcrd node that tries to send it closes that connection after the encoding error. An SR with such a width built by other software fails to decode at a dcrd receiver, which bans the sender for a malformed wire message. The reference client does not keep M at or below 512 when forming sessions. Clients **SHOULD NOT** form sessions with $M > 512$. `wire/msgmixslotreserve.go · BtcDecode, writeMessageNoSignature` | `peer/peer.go · outHandler` | `server.go · OnRead`

Table 18.8 `mixfactpoly` (FP) payload

Field	Type	Bytes	Description
`Signature`	`[64]byte`	64	Signature.
`Identity`	`[33]byte`	33	Sender identity.
`SessionID`	`[32]byte`	32	Session identifier.
`Run`	`uint32`	4	Run number; 0.
`RootCount`	`varint`	1–3	At most 1024 (`MaxMixMcount`).
`Roots`	`varbytes[]`	var	Roots of the slot-reservation polynomial in increasing numeric order, minimal big-endian, each at most 32 bytes.
`SeenCount`	`varint`	1–3	At most 512.
`SeenSlotReserves`	`[32]byte[]`	32 each	SR message hashes in unmixed order.

Table 18.9 `mixdcnet` (DC) payload

Field	Type	Bytes	Description
`Signature`	`[64]byte`	64	Signature.
`Identity`	`[33]byte`	33	Sender identity.
`SessionID`	`[32]byte`	32	Session identifier.
`Run`	`uint32`	4	Run number; 0.
`X`	`varint`	1–3	Number of vectors (the sender's m). If 0, the next two fields and the data are absent. Encoders **MUST NOT** send 0.
`Y`	`varint`	1–3	Elements per vector (the session's M). $X \leq 1024$ and $Y \leq 1024$.
`MsgSize`	`varint`	1	Element size; **MUST** be 20 (`MixMsgSize`).
`DCNet`	`[20]byte[X][Y]`	20XY	Vector j, element k, row-major.
`SeenCount`	`varint`	1–3	At most 512.
`SeenSlotReserves`	`[32]byte[]`	32 each	SR message hashes in unmixed order.

Table 18.10 `mixconfirm` (CM) payload

Field	Type	Bytes	Description
`Signature`	`[64]byte`	64	Signature.
`Identity`	`[33]byte`	33	Sender identity.
`SessionID`	`[32]byte`	32	Session identifier.
`Run`	`uint32`	4	Run number; 0.
`Mix`	`MsgTx`	var	The CoinJoin transaction in full serialization (prefix and witness, Chapter 4), with signature scripts present for the sender's own inputs.
`SeenCount`	`varint`	1–3	At most 512.
`SeenDCNets`	`[32]byte[]`	32 each	DC message hashes in unmixed order.

Table 18.11 `mixsecrets` (RS) payload

Field	Type	Bytes	Description
`Signature`	`[64]byte`	64	Signature.
`Identity`	`[33]byte`	33	Sender identity.
`SessionID`	`[32]byte`	32	Session identifier.
`Run`	`uint32`	4	Run number; 0.
`Seed`	`[32]byte`	32	The run's PRNG seed (Section 18.7.2).
`SRCount`	`varint`	1–3	At most 1024.
`SlotReserveMsgs`	`varbytes[]`	var	The unpadded slot-reservation messages, minimal big-endian, each at most 32 bytes.

Field	Type	Bytes	Description
N	varint	1–3	Number of DC messages, at most 1024. If 0, MsgSize and data are absent.
MsgSize	varint	1	Present if $N > 0$; **MUST** be 20.
DCNetMsgs	[20]byte[N]	20N	The unpadded DC-net messages (output key hashes).
SeenCount	varint	1–3	At most 512.
SeenSecrets	[32]byte[]	32 each	Hashes of RS messages of the session already seen (Section 18.8.4).

All `varint`s **MUST** be canonical (Chapter 4). The decode-time bounds above come from `wire`; the mixpool applies tighter limits from the `mixing` package (Table 18.16), and the maximum payload lengths that the message framing enforces are listed in Table 18.12.

Table 18.12 Maximum payload lengths (protocol version 10 and later)

Message	Max payload	Message	Max payload
`mixpairreq`	8476848	`mixfactpoly`	49291
`mixkeyxchg`	17815	`mixdcnet`	20988047
`mixcphrtxt`	552584	`mixconfirm`	1016520
`mixslotres`	17318030	`mixsecrets`	70831

Table 18.13 Pair request capability flags

Bit	Name	Meaning
`0x01`	`PRFlagCanSolveRoots`	The peer can factor the slot-reservation polynomial and will publish FP messages (Section 18.7.8).
others		Reserved; ignored.

18.5 Pair Requests

A pair request announces a peer's wish to mix, binds its identity to the inputs it will spend and proves it can spend them. It is broadcast before the epoch in which the peer wants to mix and remains valid, possibly through several epochs, until it expires or its inputs are spent.

18.5.1 Field semantics

- `MixAmount` is the value of every mixed output. `MessageCount` $= m$ is the number of mixed outputs the peer receives.
- `ScriptClass` **MUST** be `P2PKH-secp256k1-v0` (`ScriptClassP2PKHv0`), the only class currently accepted: mixed outputs are version-0 P2PKH scripts `76 a9 14 <20-byte hash> 88 ac`.
- `TxVersion` and `LockTime` are copied into the CoinJoin. The reference client creates the CoinJoin with transaction version 1 and lock time 0.
- `InputValue` is the total value of the UTXOs; `Change` is the optional change output, taken verbatim into the CoinJoin. The peer's fee contribution is InputValue $-\ m \cdot$ MixAmount $-$ change value.
- `Expiry` is a block height; see Section 18.5.3.
- `Flags` advertise capabilities (Table 18.13). A client sets `PRFlagCanSolveRoots` only if a root solver is available to it.
- `PairingFlags` select the protocol variant: the reference client uses `pairingVersion` $= 3$ on every network except the test network, where it uses `pairingVersionTestnet` $= 4$. A client **MUST** use the same value to pair with reference peers.

18.5.2 Pairing description

Two pair requests are *compatible*, and can be mixed together, exactly when their *pairing descriptions* are byte-identical. The description is

$$\begin{aligned}\text{pairing} = {} & \text{le64}(\text{MixAmount}) \parallel \text{varstring}(\text{ScriptClass}) \\ & \parallel \text{le16}(\text{TxVersion}) \parallel \text{le32}(\text{LockTime}) \parallel \text{PairingFlags}\end{aligned} \tag{18.2}$$

(one byte for the flags). `wire/msgmixpairreq.go · Pairing`

18.5.3 Expiry

With h the height of the current chain tip, a pair request is acceptable only if $h < \text{Expiry} \le \text{MaxExpiry}(h)$, where

$$\text{MaxExpiry}(h) = h + \left\lfloor \frac{60 \text{ min}}{\texttt{TargetTimePerBlock}} \right\rfloor + 1, \tag{18.3}$$

that is $h + 13$ on the main network (5-minute blocks), $h + 31$ on the test network and $h + 3601$ on the simulation and regression networks. `mixing/expiry.go · MaxExpiry` The reference client signs ownership proofs for, and sets, an expiry chosen by the wallet within this window; dcrwallet uses $\text{MaxExpiry}(h) - 2$. Every session formed from a PR inherits the earliest expiry of its known PRs (Section 18.9.4).

18.5.4 UTXO ownership proofs

Each contributed UTXO carries a proof that the peer holds the private key controlling it. The proof is an EC-Schnorr-DCRv0 signature over a digest that commits to the PR's expiry and to the public key. The source states its purpose as preventing the proof's inclusion in other pair requests signed by an unrelated identity. Together with the mixpool rule that a PR may claim an outpoint already claimed by an accepted PR only with a strictly later expiry (Section 18.9.2), a copied proof, valid only for the original expiry, cannot displace the owner's PR at a node that already holds it.

ALGORITHM 18.3 UTXO ownership proof

```
PREAMBLE = b"mixpr-utxoproof,P2PKH(EC-Schnorr-DCRv0),"   # tag "," scheme ","

def utxo_proof_digest(pubkey: bytes, expiry: uint32) -> bytes:
    return BLAKE256(PREAMBLE + be32(expiry) + pubkey)     # no separator after expiry

def sign_utxo_proof(priv, expiry):
    pub = compressed_pubkey(priv)                         # 33 bytes
    return pub, ec_schnorr_dcrv0_sign(priv, utxo_proof_digest(pub, expiry))

def check_utxo(utxo, prev_pk_script, expiry) -> bool:
    # prev_pk_script: the script of the referenced output, script version 0
    if utxo.opcode == 0:          h = extract_p2pkh_v0(prev_pk_script)
    elif utxo.opcode == OP_SSGEN: h = extract_stakegen_p2pkh_v0(prev_pk_script)
    elif utxo.opcode == OP_SSRTX: h = extract_stakerevoke_p2pkh_v0(prev_pk_script)
    elif utxo.opcode == OP_TGEN:  h = extract_treasurygen_p2pkh_v0(prev_pk_script)
    else: return False
    return (h is not None and HASH160(utxo.pubkey) == h
            and ec_schnorr_dcrv0_verify(utxo.pubkey,
                                        utxo_proof_digest(utxo.pubkey, expiry),
                                        utxo.signature))
```

dcrd: `mixing/utxoproof/utxoproof.go · SignUtxoProof, ValidateSecp256k1P2PKH`

The expiry is written **big-endian**, unlike the wire fields. The supported output types are version-0 P2PKH outputs and the version-0 P2PKH forms of vote, revocation and treasury-generation outputs tagged with `OP_SSGEN`, `OP_SSRTX` and `OP_TGEN` (Chapter 6). P2SH outputs are not supported: the `Script` field **MUST** be empty. A client sets `Opcode` to the first byte of the previous output script when that byte is one of the three stake opcodes, and to 0 otherwise.

18.5.5 Fee and dust policy

Relays enforce a minimum and maximum fee contribution per PR so that the CoinJoin passes default mempool policy (Chapter 17). The fee rate is `feeRate` $= 10000$ atoms per kilobyte. The size of the peer's share of the transaction is estimated as if every input redeemed a P2PKH output with a 108-byte signature script and every output were a P2PKH output:

$$\text{size}(n_{\text{in}}, n_{\text{out}}, c) = 12 + 2 \cdot \text{varintlen}(n_{\text{in}}) + \text{varintlen}(n_{\text{out}} + c) + 166 n_{\text{in}} + 36(n_{\text{out}} + c) \quad (18.4)$$

where n_{in} is the UTXO count, $n_{\text{out}} = m, c \in \{0, 1\}$ indicates a change output, and 166 and 36 are the estimated sizes of one input and one P2PKH output. With $\text{fee}(s) = \lfloor 10000 \cdot s/1000 \rfloor$ (but at least 10000 if that is 0, and clamped to 21×10^{14}), the contributed fee must satisfy $\text{fee}(\text{size}) \leq \text{contributed} \leq \text{fee}(10000 \cdot \text{size})$. `mixing/mixpool/mixpool.go · checkFee, estimateP2PKHv0SerializeSize`

An output is *dust* if $\lfloor v \cdot 1000/(3 \cdot 201) \rfloor < 10000$, where 201 is the estimated cost in bytes of creating and later spending a P2PKH output $(8 + 2 + 1 + 25 + 165)$. The smallest non-dust value is therefore 6030 atoms. The mix amount and any change value are checked with this formula (the change with the P2PKH size, whatever its actual script). The product $v \cdot 1000$ is computed in signed 64-bit arithmetic and wraps for $v > 9223372036854775$, far above the money supply. `mixing/mixpool/mixpool.go · isDustAmount`

18.6 Epochs and Session Formation

18.6.1 Epochs

An *epoch* boundary is a Unix time that is a multiple of the epoch duration E: 600 seconds on the main, simulation and regression networks, and 180 seconds on the test network (test network 3). The duration is hard-coded in the mixpool (10 minutes, replaced by 3 minutes when the network is test network 3), not a chain parameter; the client in turn selects the test-network stage timeout and pairing version by checking whether the mixpool's epoch is 3 minutes. The value of an epoch is that boundary in seconds. `mixing/mixpool/mixpool.go · NewPool` | `mixing/mixclient/client.go · NewClient`

Clients **SHOULD** publish their pair request at a random time well away from the boundaries, so that publication time does not link a PR to its owner and so that all peers know all PRs before the tick. The reference client, with the next boundary at T, stage timeout τ (Table 18.14) and maximum jitter $j_{\max} = \tau/10$, publishes at

$$t = \text{now} + d_p + U[0, 300\ \text{ms}) + U[0, T - \tau - j_{\max} - \text{now}) \quad (18.5)$$

if $\text{now} < T - \tau - j_{\max}$, and otherwise waits until $T + \tau$ and then adds $d_p + U[0, 300\ \text{ms}) + U[0, T + E - \tau - j_{\max} - \text{now})$. Here d_p is a per-identity delay drawn once from $U[0, j_{\max} - 300\ \text{ms})$ and added to every message the peer sends. `mixing/mixclient/client.go · prDelay, msgJitter`

18.6.2 Choosing the participants

At each epoch boundary T, a client with waiting local pair requests, for each distinct pairing description, takes the set of PRs in its mixpool with that description, as follows.

1. Before anything else, prune the mixpool: remove every session for which a CM from each of its PRs has been seen, together with those PRs, and remove every PR and every session whose expiry is at or below the height most recently reported to the client by its wallet when a block was attached to the main chain (0, so no height expiry, until the first such block after the client started). This is immediate, unlike the delayed expiry dcrd performs (Section 18.9.4), and it runs at every epoch boundary after the first, whether or not local pair requests are waiting.
2. Take all PRs with the pairing description.
3. Sort them by decreasing expiry and drop any PR that spends an outpoint also spent by a PR with a strictly greater expiry (a peer replaces a PR by broadcasting one with a later expiry).
4. Drop PRs that spend outputs with two or more misbehavior strikes (Section 18.9.5).
5. If more than 512 remain, keep the local peers' PRs and then the others in increasing hash order, up to 512.

`mixing/mixclient/client.go · epochTicker, expireMessages | mixing/mixpool/mixpool.go · RemoveConfirmedSessions, ExpireMessages, CompatiblePRs, Observer.ExcludePRs`

Different peers may see different PR sets; the session-agreement procedure of Section 18.6.5 resolves that. A client that omits the pruning of step 1 keeps PRs that reference clients have already dropped, derives a different session ID from them in its first attempt, and reaches a common session only through that procedure. The reference peers' first attempt fails as well, because they wait for a KE carrying their own session ID from every PR, so one such client costs every peer of the pairing a KE timeout and an alternate-session round.

18.6.3 Session identifier and unmixed order

The session identifier commits to the exact set of PRs and to the epoch. The *unmixed order* of the peers, which fixes each peer's index in every vector of the run, is the order of the PR hashes after XOR with the session ID, which is pseudorandom and not under any peer's control.

ALGORITHM 18.4 Session ID and unmixed order

```
def session_id(pr_hashes: list[bytes], epoch: uint64) -> bytes:
    h = sorted(pr_hashes)                       # bytewise lexicographic, ascending
    return BLAKE256(b"decred-mix-session" + be64(epoch) + b"".join(h))

def form_session(prs, epoch):
    sid = session_id([pr.hash for pr in prs], epoch)
    ordered = sorted(prs, key=lambda pr: xor(pr.hash, sid))   # ascending
    return sid, ordered       # ordered[k] is the peer at unmixed position k

def validate_session(ke) -> bool:                # checked for every KE
    x = [xor(h, ke.session_id) for h in ke.seen_prs]
    if any(x[k] < x[k - 1] for k in range(1, len(x))):
        return False                             # not in unmixed order
    if ke.run == 0 and session_id(ke.seen_prs, ke.epoch) != ke.session_id:
        return False
    return True
```

dcrd: `mixing/sid.go · deriveSessionID, SortPRsForSession, ValidateSession`

The epoch is serialized **big-endian** in the session-ID preimage. For a peer, `Pos` is its index in the ordered list, m_k the message count of the peer at position k, $M = \sum_k m_k$ the session's total message count, and $s_k = \sum_{i<k} m_i$ the *global index* of the first message of peer k. Messages are numbered $0..M-1$ in this order everywhere below.

18.6.4 Minimum participation

A client always broadcasts its KE for the session it formed, even when the session cannot proceed, so that other peers can fetch PRs they are missing. It does not proceed past the KE unless the session has at least `MinPeers` $= 4$ PRs and at least one PR has `PRFlagCanSolveRoots` set. `mixing/mixclient/client.go · run`

The KE also shows that the peer is still present. When its pairing attempt for the epoch ends, the reference client removes from its own mixpool the PR of every identity that an alternate-session computation of that epoch (Algorithm 18.5) found unresponsive, that is, from which the mixpool holds no KE for that epoch. A computation contributes to this list only if it found an alternate session that differs from the one attempted, and the removal happens only if at least `MinPeers` identities of that pairing description have a most recent KE of that epoch. The client applies a similar test once, shortly after the KE deadline of the first epoch boundary following its start, to the PRs it held at start: every such PR with no KE of that epoch is a candidate, without the alternate-session condition, and the client stops at the first pairing description with fewer than `MinPeers` such KEs, in unspecified order, so silent PRs of other pairings may or may not be removed. A removed PR is not paired again unless it is received anew. A peer with a pending PR therefore **SHOULD** send a KE at every epoch boundary, and an implementation that never removes silent PRs keeps forming initial

sessions that reference clients do not form. `mixing/mixclient/client.go · removeUnresponsiveDuringEpoch, pairSession, epochTicker | mixing/mixpool/mixpool.go · RemoveUnresponsiveDuringEpoch`

18.6.5 Session agreement

After sending its KE, a peer waits until the KE deadline (Table 18.14) for a KE from every PR of the session, where "the session" is any of the session attempts made so far in this epoch since the last point of agreement; if one of them completes it is used. If none completes and the peers have not yet reached agreement in this epoch, the peer computes an *alternate session* from the KEs it has received, which are evidence of what PR sets the other peers saw. If none completes after agreement was reached, the pairing is abandoned for the epoch.

ALGORITHM 18.5 Alternate session

```
def alternate_session(prs, latest_ke, epoch):
    # prs: PRs of the failed attempt; latest_ke[id]: the KE most recently
    # accepted from identity id, present only if that KE has this epoch
    # (an older KE of the epoch is not used)
    by_hash = {pr.hash: pr for pr in prs}
    resp = {pr.identity: (pr, latest_ke[pr.identity])
            for pr in prs if pr.identity in latest_ke}
    silent = {pr.hash for pr in prs if pr.identity not in latest_ke}
    count = {}
    for pr, ke in resp.values():
        for h in ke.seen_prs:
            if h != pr.hash and h not in silent:        # self-references excluded
                count[h] = count.get(h, 0) + 1
    need = len(count) // 2
    rem = [h for h in count if count[h] >= need]
    while True:
        if len(rem) < MIN_PEERS:
            fail("too few peers")
        rem.sort(key=lambda h: (count[h], h))          # fewest sightings first
        if count[rem[0]] == len(rem) - 1:              # everyone saw everyone
            if any(h not in by_hash for h in rem):
                fail("agreement reached on PRs unknown locally")
            return form_session([by_hash[h] for h in rem], epoch)
        rem = rem[1:]                                   # drop least-seen PR
        count = {h: 0 for h in rem}
        for h in rem:
            if h in by_hash and by_hash[h].identity in resp:
                for s in resp[by_hash[h].identity][1].seen_prs:
                    if s != h and s in count:
                        count[s] += 1
```

`dcrd: mixing/mixclient/client.go · alternateSession`

If the alternate session differs from the current attempt, the peer starts a new attempt with it (fresh KE, same keys and secrets), restarting the stage deadlines from the current time; if it is identical, agreement is assumed to have been reached and the attempt is waited for again with restarted deadlines. If the computation fails (too few peers, or agreement on PRs unknown locally), the peer restarts the deadlines and keeps waiting for KEs of the attempts made so far. Each outcome is counted per resulting session identifier, all failures counting as one outcome; the third occurrence of the same outcome in an epoch abandons the pairing for that epoch. All of this happens within the epoch: the client abandons the pairing attempt when the next epoch begins. `mixing/mixclient/client.go · pairSession`

Figure 18.1 summarizes the procedure.

18.6.6 Size limits

When all KEs of a session have arrived and agreement has not yet been recorded in the epoch (agreement recorded because an alternate session reproduced the current attempt, Section 18.6.5, skips this check), each peer checks the session against the mixing limits of Table 18.16, walking the PRs in unmixed order

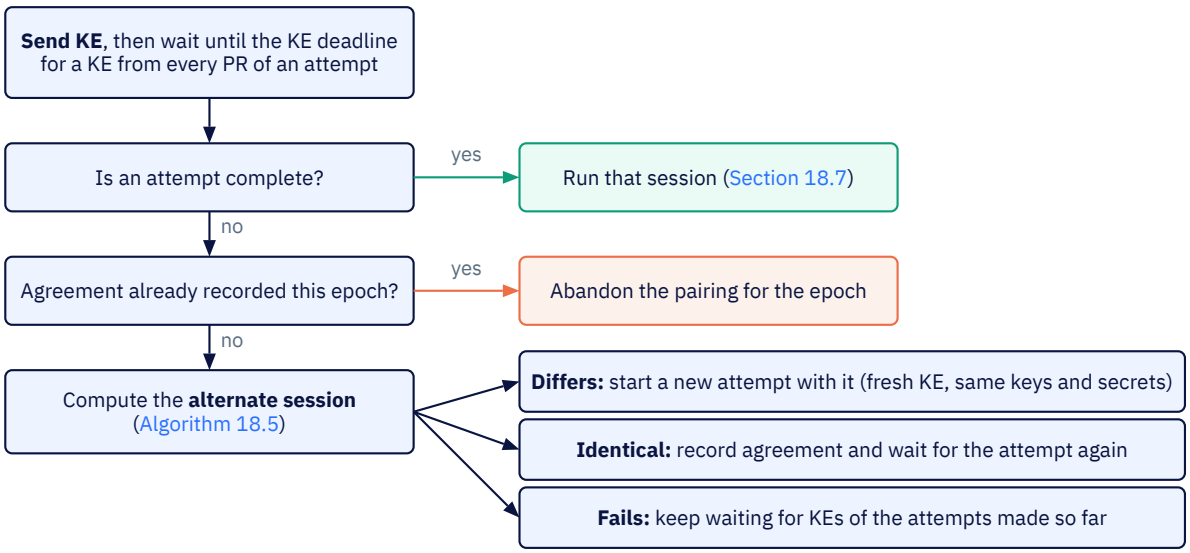


Figure 18.1 Session agreement within one epoch. Every outcome of the alternate session restarts the stage deadlines and returns to the wait; the third occurrence of the same outcome in an epoch abandons the pairing, as does the start of the next epoch.

and excluding a PR if adding it would exceed 64 peers, a total of 1024 messages, or an estimated CoinJoin size of 100000 bytes. The size estimate is $12 + 2 \cdot \mathrm{varintlen}(n_{\mathrm{in}}) + \mathrm{varintlen}(n_{\mathrm{out}}) + 166 n_{\mathrm{in}} + \sum_{\mathrm{outputs}} (10 + \mathrm{varintlen}(\ell) + \ell)$ over all inputs and outputs so far, with $\ell = 25$ for each mixed output and the actual script length for each change output; a PR is excluded when the estimate is ≥ 100000. If any PR is excluded, the remaining PRs form a new session (Algorithm 18.4) and the run restarts with it, provided at least four remain.
`mixing/mixclient/client.go · run, coinjoinSize.join`

18.6.7 Runs and reruns

CoinShuffle++ numbers successive runs of a session. In dcrd every run is instead a fresh session: whenever peers are excluded (blame, timeout, size limits, disagreement), the remaining PRs are re-sorted with Algorithm 18.4 using the **same epoch value**, giving a new session ID, and every message of the new attempt carries run number 0. Relays reject any message whose run number is not 0. The `Epoch` field of every KE in the epoch is the original epoch value, and relays accept KEs only up to 20 minutes after it (Section 18.9.2), which bounds the time available for reruns.

18.7 The Protocol Run

This section describes one run of a session with peers $0..n-1$ in unmixed order. "Peer k" refers to unmixed positions, "message g" to global message indices.

18.7.1 Deadlines

Each run has a start time T_0: the epoch boundary for the first attempt of an epoch, and the current time when a rerun is started. All deadlines are relative to it:

Table 18.14 Stage deadlines of a run (τ = 30 s on the main, simulation and regression networks, 10 s on the test network)

Deadline	Time	By then
KE	$T_0 + \tau$	All KEs of the session received.
CT	$T_0 + 2\tau$	All CTs received.
SR	$T_0 + 3\tau$	All SRs received. (A peer that cannot solve roots starts waiting for a valid FP as soon as it has all SRs; Section 18.7.8.)
DC	$T_0 + 4\tau$	All DCs (and FPs, sent together with them) received.

Deadline	Time	By then
DC-RS	$T_0 + 5\tau$	Wait for secrets revealed after the DC stage; CMs are not sent before this time.
CM	$T_0 + 6\tau$	All CMs received.

`mixing/mixclient/client.go · deadlines.start, timeoutDuration`

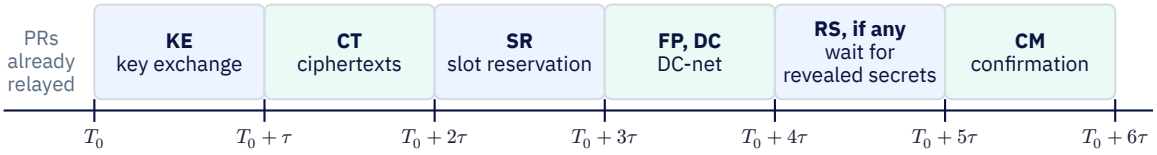


Figure 18.2 Stages of one run. Each band names the messages that are due by the deadline at its right edge (Table 18.14).

Each message is sent after the sender's per-identity delay d_p plus a fresh $U[0, 300\text{ ms})$, and is due by the deadline of its stage. A peer that has not received a message of the current stage from some peer by the deadline treats that peer as having timed out (Section 18.8.2).

18.7.2 Key generation and commitment

For every run in which it must generate fresh secrets (the first run of a PR, and every run after secrets were revealed), a peer:

1. Draws a uniformly random 32-byte *seed* and sets $P = \text{PRNG}(\text{seed}, 0)$.
2. Derives its secp256k1 Diffie-Hellman key pair from P: read 32 bytes, interpret them as a big-endian integer, and repeat until the value is in $[1, N-1]$ (N the group order).
3. Derives its sntrup4591761 key pair from the continuation of P, consuming randomness as specified in Section 18.2.1.
4. Draws m slot-reservation messages $r_0..r_{m-1}$ uniformly from $[0, F)$ with a system random source (not from P).
5. Generates m fresh secp256k1 keys in its wallet and takes their HASH160s $d_0..d_{m-1}$ as its DC-net messages; these are the 20-byte hashes of the mixed outputs it will receive.

The state of P after step 3 is kept: the encapsulations of the CT stage continue reading from it. `mixing/mixclient/client.go · genDicemixKeys` | `mixing/keyagreement.go · NewKX`

IMPLEMENTATION PITFALL
Blame verification (Section 18.8.4, step 2) recomputes a peer's CT ciphertexts by encapsulating from the state of P immediately after key generation. The reference client keeps a single P per secret generation and continues it, without rewinding, whenever it reuses secrets: in a rerun after a timeout or an observable-misbehavior exclusion, and in a session of a later epoch for a pair request whose earlier session ended without a reveal (the peer state is cloned with the same PRNG object). A peer that already encapsulated in an earlier run therefore encapsulates the new CT from an advanced state, and if secrets are revealed in that run it fails step 2 and is blamed although it followed the protocol. Driving dcrd's client with six local peers, one withholding its CT (CT timeout, rerun of five peers reusing secrets) and then one flipping a bit of its SR in the rerun, blame assignment blamed all five peers for ciphertexts that differ from the recomputation, and the session was abandoned. A new implementation **SHOULD** generate fresh secrets for every run, which protects its own messages; it cannot prevent dcrd peers of the same run from being blamed this way. `mixing/mixclient/client.go · cloneLocalPeer, excludeBlamed, epochTicker`

18.7.3 The commitment

The KE's `Commitment` is the message hash of the RS message the peer would publish for this session, computed with the signature set to 64 zero bytes and an empty `SeenSecrets` list:

$$\begin{aligned}\text{Commitment} = H(0^{64} \parallel \text{Identity} \parallel \text{SessionID} \parallel \text{le32}(0) \\ \parallel \text{Seed} \parallel \text{SR-list} \parallel \text{DC-list} \parallel \text{varint}(0))\end{aligned} \tag{18.6}$$

where the SR list and DC list are serialized as in Table 18.11 (the SR messages minimal big-endian). `wire/msgmixsecrets.go · Commitment`

18.7.4 Key exchange (KE)

The peer sends a KE with the session ID, the epoch, run 0, `Pos` set to its unmixed position, its ECDH public key (compressed), its sntrup4591761 public key, the commitment, and `SeenPRs` set to all PR hashes of the session in unmixed order. The session ID, epoch and `SeenPRs` must satisfy Algorithm 18.4. When all n KEs have arrived (Section 18.6.5), every peer parses every ECDH key; a KE whose key does not parse is observable misbehavior (Table 18.15).

18.7.5 Ciphertexts (CT)

Peer p encapsulates once to the sntrup4591761 public key of every peer $k = 0..n-1$ **including itself**, in increasing k, using the continuation of its PRNG P as the randomness source, obtaining ciphertext $c_{p,k}$ and cleartext key $e_{p,k}$. It sends a CT whose `Ciphertexts` list is $c_{p,0}..c_{p,n-1}$ and whose `SeenKeyExchanges` lists the KE hashes in unmixed order. `mixing/keyagreement.go · KX.Encapsulate`

18.7.6 Shared keys

On receiving all CTs, peer p checks that each lists exactly n ciphertexts (otherwise the sender is excluded, Table 18.15) and decapsulates the ciphertext $c_{k,p}$ addressed to it by each peer k. A pairwise 32-byte key between peers p and k is

$$K_{p,k} = H\left(\text{ECDH}_{x(p,k)}\right) \oplus e_{k,p} \oplus e_{p,k} \quad (k \neq p), \qquad K_{p,p} = H\left(\text{ECDH}_{x(p,p)}\right) \oplus e_{p,p} \tag{18.7}$$

where $\text{ECDH}_{x(p,k)}$ is the 32-byte big-endian x coordinate of $a_p \cdot A_k$ for p's ECDH private key a_p and k's ECDH public key A_k. $K_{p,k} = K_{k,p}$. From the pairwise keys, a separate PRNG is derived for every *pair of messages*, giving a 32-byte slot-reservation secret and an M-element XOR pad.

ALGORITHM 18.6 Pairwise message secrets

```
def shared_secrets(p, mcounts, sid, kx, ecdh_pub, ct_to_me):
    # p: own unmixed position; ct_to_me[k]: CT_k.ciphertexts[p]
    M = sum(mcounts)
    sr = [[None] * M for _ in range(mcounts[p])]   # sr[i][g]: 32 bytes
    dc = [[None] * M for _ in range(mcounts[p])]   # dc[i][g]: M x 20 bytes
    for i in range(mcounts[p]):                     # own message index
        g = 0                                       # global index of the other message
        for k in range(len(mcounts)):
            if k == p and mcounts[p] == 1:
                g += 1
                continue
            key = BLAKE256(ecdh_x(kx.ecdh_priv, ecdh_pub[k]))
            e_in = sntrup_decapsulate(ct_to_me[k], kx.pq_priv)
            if e_in is None:
                raise DecapsulateError(k)           # triggers blame
            key = xor(key, e_in)
            if k != p:
                key = xor(key, kx.cleartext[k])     # the key p encapsulated to k
            for j in range(mcounts[k]):
                if k == p and j == i:
                    g += 1
                    continue
                if p < k or (p == k and j < i):
                    counter, nonce = i, j           # lower peer's index is the counter
                else:
```

```
                counter, nonce = j, i
            seed = BLAKE256(sid + key + le32(counter))
            rng = PRNG(seed, nonce)
            sr[i][g] = rng.read(32)
            dc[i][g] = [rng.read(20) for _ in range(M)]
            g += 1
    return sr, dc
```

dcrd: mixing/keyagreement.go · KX.SharedSecrets

For own message i (global index $s_p + i$), entry g is the secret shared with message g; the entry for the message's own index is unused. The two holders of a message pair derive the same seed and nonce: the counter is always the message index of the lower-positioned peer (for two messages of the same peer, the larger index), and the nonce the other index.

18.7.7 Slot reservation (SR)

Slot reservation lets the peers agree on a random permutation of the M messages without revealing who owns which position. Each message g is represented by its random field element r, and the peers jointly compute, with a DC-net [54] over $\mathbb{F}_F$, the *power sums* $S_t = \sum_g r_g^t \bmod F$ for $t = 1..M$, from which the set $\{r_g\}$ (but not the owners) can be recovered.

ALGORITHM 18.7 Slot-reservation pads and mix

```
def sr_pads(kp: list[bytes], my: int, M: int) -> list[int]:
    # kp[g]: 32-byte SR secret shared with message g; my: own global index
    pads = []
    for t in range(M):
        acc = 0
        for g in range(M):
            if g == my:
                continue
            d = int.from_bytes(BLAKE256(kp[g] + le64(t + 1)), "big")   # 256-bit
            acc = acc + d if my > g else acc - d
        pads.append(acc % F)          # non-negative residue
    return pads

def sr_mix(r: int, pads: list[int]) -> list[int]:
    return [(pow(r, t + 1) + pads[t]) % F for t in range(len(pads))]
```

dcrd: mixing/dcnet.go · SRMixPads, SRMix

For its i-th message, peer p computes `sr_mix(r_i, sr_pads(sr[i], s_p + i, M))` and sends the m resulting rows as the SR's `DCMix` matrix (m rows of width M), with `SeenCiphertexts` listing the CT hashes in unmixed order. Every pad term appears once with each sign across the session, so pads cancel in the sum.

On receiving all SRs, each peer checks that the SR of peer k has exactly m_k rows of width M (otherwise the sender is excluded), concatenates all rows in unmixed order and adds them element-wise modulo F, giving $S_1..S_M$. The polynomial whose roots are the messages is built with Newton's identities:

ALGORITHM 18.8 Slot-reservation polynomial coefficients

```
def coefficients(S: list[int]) -> list[int]:
    # S[t] = sum of r^(t+1); returns a[0..M] of phi(x) = sum a[i] x^i (mod F)
    M = len(S)
    a = [0] * (M + 1)
    a[M] = F - 1                                  # -1
    for i in range(M):
```

```
        acc = sum(a[M - i + j] * S[j] for j in range(i + 1))
        a[M - 1 - i] = (acc * (F - pow(i + 1, -1, F))) % F
    return a

def is_root(x: int, a: list[int]) -> bool:
    return sum(a[i] * pow(x, i, F) for i in range(len(a))) % F == 0
```

dcrd: mixing/dcnet.go · Coefficients, IsRoot

$\varphi(x) = -\prod_g (x - r_g)$; its M roots are exactly the messages.

18.7.8 Root finding and the FP message

Factoring a degree-M polynomial over a 127-bit prime field needs a dedicated solver; dcrd delegates it to an external program (`csppsolver` from the `decred.org/cspp/v2` module) and sets `PRFlagCanSolveRoots` only if that program starts. Any correct root-finding algorithm is acceptable.

- A peer that can solve computes the roots. If the solver returns an error, a number of roots other than M, or reports a repeated root (two peers chose the same r), the peer triggers blame (Section 18.8). Otherwise it sorts the roots in increasing numeric order and sends an FP carrying them (minimal big-endian) and `SeenSlotReserves` listing the SR hashes in unmixed order; the FP is sent together with its DC message. If the solver process has exited, the peer instead clears `PRFlagCanSolveRoots` for its future PRs and waits for an FP as below.
- A peer that cannot solve waits for any FP of the session that is *valid*: exactly M roots, each in $[0, F)$ and a root of φ by Algorithm 18.8, pairwise distinct, and in strictly increasing order. It uses the first valid FP it finds (each sender's FP is examined once) and ignores invalid ones. The SR deadline ends only the first blocking wait: the reference client keeps checking for a valid FP after it, and uses one that arrives later, until an RS of the session appears or the session leaves its mixpool. `mixing/mixclient/client.go · roots`

The *slot* of message r_i is its index in the sorted root list. A peer that does not find one of its own r_i among the roots triggers blame. The search **SHOULD** be constant-time.

18.7.9 XOR DC-net (DC)

The second DC-net transmits the 20-byte output hashes in the reserved slots. For its i-th message, with slot σ_i and the pad vectors $\mathrm{dc}[i][g]$ of Algorithm 18.6, peer p computes

$$V_i = \left(\bigoplus_{g \neq s_p + i} \mathrm{dc}[i][g] \right) \oplus (d_i \text{ placed at element } \sigma_i), \tag{18.8}$$

a vector of M 20-byte elements (`mixing/dcnet.go · DCMixPads, DCMix`), and sends the m vectors $V_0..V_{m-1}$ in a DC, with `SeenSlotReserves` listing the SR hashes in unmixed order. On receiving all DCs, each peer checks that peer k's DC has m_k vectors of M elements (otherwise the sender is excluded) and XORs all M vectors element-wise; element σ of the result is the message in slot σ.

18.7.10 Building the CoinJoin

Every peer builds the same unsigned transaction:

1. Version `TxVersion` and lock time `LockTime` (common to all PRs by compatibility), expiry 0, full serialization type.
2. For each PR in unmixed order, one input per UTXO in PR order, with the given outpoint, sequence `0xffffffff`, empty signature script and the null witness values (`ValueIn` $= -1$, `BlockHeight` $= 0$, `BlockIndex` $=$ `0xffffffff`); and the PR's change output, if any, verbatim.
3. For each slot $\sigma = 0..M-1$, an output of value `MixAmount`, script version 0 and script `76 a9 14 || D[σ] || 88 ac`, where D is the XOR DC-net result.

4. Sort the transaction in place: inputs by previous-outpoint tree (ascending), then by previous transaction hash compared bytewise in **display** (byte-reversed) order, then by output index; outputs by value, then script version, then script bytes (lexicographic, a proper prefix first).

`mixing/mixclient/coinjoin.go · resetUnmixed, addMixedMessage, sort | dcrutil/txsort/txsort.go · InPlaceSort`

The witness values of the inputs are not covered by the transaction hash, which is computed over the prefix only (Chapter 4); dcrd's mempool fills in the fraud-proof witness from the UTXO set when the transaction is accepted.

18.7.11 Confirmation and publication (CM)

Each peer checks that for each of its own messages d_i the sorted transaction has an output with value `MixAmount`, script version 0 and the P2PKH script paying to d_i (the reference implementation scans in constant time over outputs of equal value, version and script length). If any is missing, the peer triggers blame. Otherwise it signs each of its own inputs and waits until the DC-RS deadline; if any RS of the session has appeared by then, it joins blame assignment instead of confirming. Otherwise it sends a CM containing the transaction with its own signature scripts and `SeenDCNets` listing the DC hashes in unmixed order.

On receiving all CMs, a peer checks that each CM's transaction has the same transaction hash as its own; a CM that differs is observable misbehavior. It copies, from each CM, the signature scripts of exactly the inputs that the CM's sender contributed, then executes every input script against the previous output script reconstructed from the UTXO's public key and opcode (P2PKH, or the stake-tagged P2PKH form) with script version 0 and the flags `DiscourageUpgradableNops`, `VerifyCleanStack`, `VerifyCheckLockTimeVerify`, `VerifyCheckSequenceVerify` and `VerifyTreasury` (Chapter 6). The owners of any failing input are excluded. If all succeed, every peer publishes the transaction, after a random delay, as an ordinary transaction. The session is then complete. `mixing/mixclient/client.go · run, validateMergedCoinjoin`

18.8 Blame and Failure Handling

18.8.1 Observable misbehavior

Some failures are visible to every peer from the published messages alone. The offending peers are excluded immediately and the rest rerun (Section 18.6.7) without revealing secrets and **reusing** their existing keys and messages:

Table 18.15 Misbehavior that leads to immediate exclusion

Stage	Condition (the sender is excluded)
KE	ECDH public key does not parse.
CT	Number of ciphertexts $\neq n$.
SR	`DCMix` not exactly $m_k \times M$.
DC	`DCNet` not exactly m_k vectors of M elements.
CM	Transaction hash differs from the locally built CoinJoin; or an input contributed by the sender fails script execution.

18.8.2 Timeouts

If, at the CT, SR, DC or CM deadline, messages of that stage are missing, the peers whose message is missing are excluded and the rest rerun, reusing their secrets. (Missing KEs are handled by session agreement, Section 18.6.5.) If fewer than `MinPeers` $= 4$ peers would remain after any exclusion, the session is abandoned. `mixing/mixclient/client.go · blameTimedOut, excludeBlamed`

18.8.3 Revealing secrets

A peer *triggers blame* when it detects a failure that cannot be attributed from public data: a ciphertext addressed to it fails to decapsulate, the solver fails or finds a repeated root, one of its own slot messages is not among

the roots, or one of its own outputs is missing from the CoinJoin. It publishes its RS (the secrets it committed to in its KE) with an empty `SeenSecrets` list, alongside or instead of its next protocol message.

A peer that finds any RS of its session at the end of a wait for CT, SR, FP or DC messages (such a wait ends early once every peer has sent either the awaited message or an RS), or at the DC-RS deadline before confirming, abandons the run and joins blame assignment (an RS arriving only while CMs are being collected is ignored): it publishes its own RS with `SeenSecrets` listing the hashes of the RS messages already received for the session. Its RS **MUST** reference at least one other peer's RS; otherwise it is indistinguishable from an initiator. Each peer then waits one stage timeout after publishing its own RS for the RS messages of all peers; peers that do not send one are excluded. `mixing/mixpool/mixpool.go · Receive` | `mixing/mixclient/blame.go · blame`

18.8.4 Blame procedure

With all n RS messages, each peer replays every other peer's computation. The checks are applied in the following order, and each group returns as soon as it has blamed anyone; the next run excludes the blamed peers and every peer generates **fresh** secrets. `mixing/mixclient/blame.go · blame`

1. For each peer: it sent a KE; the RS commitment (Section 18.7.3) equals the KE's `Commitment`; every revealed SR message is in $[0, F)$; recomputing $P = \mathrm{PRNG}(\mathrm{Seed}, 0)$ and the key pairs (Section 18.7.2) reproduces the KE's ECDH and sntrup4591761 public keys; and the RS has exactly m_k SR messages and m_k DC messages.
2. For each peer: re-encapsulating to all peers from the continuation of its P reproduces exactly the ciphertexts of its CT (which must exist).
3. Any SR message revealed by more than one peer: all those peers are blamed.
4. For each peer: it sent an SR of the right dimensions; with the recomputed ciphertexts addressed to it and its recomputed private key, its shared secrets can be derived (if a decapsulation fails, the reference client blames this peer, the recipient, not the peer whose ciphertext failed); and recomputing its `DCMix` from the revealed SR messages reproduces the published one.
5. If no roots were obtained in the run, step 6 is skipped.
6. For each peer: it sent a DC with m_k vectors; and recomputing each vector from its revealed DC messages, the slots of its revealed SR messages among the roots, and its pads reproduces the published vector.
7. If nobody has been blamed, every peer whose RS references no RS of the session other than itself, or references a hash that is not an RS of the session, is blamed (a false accusation of failure).

If blame identifies nobody, or fewer than four peers would remain, the session is abandoned.

18.9 The Mixpool: Relay Policy

Full nodes and wallets keep received mixing messages in a *mixpool*. A full node relays a message only after accepting it into its mixpool; a wallet reads session messages from its own. Everything in this section is policy: it is what dcrd enforces, and a node that wants its messages relayed by dcrd **MUST** satisfy it.

18.9.1 Relay

Messages are announced with `inv` vectors of type `MSG_MIX` (4) and fetched with `getdata` (Chapter 15). A dcrd node:

- requests an announced mixing message only when its chain is current and the message is neither in its pool, nor in a cache of recently removed messages (8500 entries, one-minute lifetime), nor in its filter of recently rejected message hashes (an age-partitioned Bloom filter of 62500 items, false-positive rate 10^{-7});
- serves `getdata` for mixing messages from its pool or the recently removed cache;
- does not announce mixing messages to peers with protocol version below 10 or that asked not to receive transaction relay;
- in blocks-only mode, ignores mixing messages and disconnects peers that announce them;

- when a KE is held as an orphan because its own PR is unknown, requests that PR from the peer that sent the KE;
- bans a peer that sends a message whose decoding fails with a wire-protocol error (any decode bound of Section 18.4), or one rejected with a *bannable* error (Table 18.17).

`server.go · onMixMessage, handleServeGetData, OnInv | internal/netsync/manager.go · OnMixMsg, needMixMsg`

18.9.2 Acceptance of every message

A message is accepted only if: its run number is 0; it has not already been accepted (a duplicate is silently ignored); and its signature is valid (Algorithm 18.2). Then, by type:

Pair request. In order: `MixAmount` $\leq$ `MaxMixAmount` $= 525000000000000$ and `MessageCount` ≤ 16; if there is change, its value is non-negative and not dust, does not exceed `InputValue`, its script version is 0 and its script is P2PKH or P2SH version 0; at least one UTXO; `MessageCount` > 0; `MixAmount` not dust; InputValue $-$ change $\geq$ MessageCount $\cdot$ MixAmount; the expiry window of Section 18.5.3; script class `P2PKH-secp256k1-v0`; the fee bounds of Section 18.5.5; no outpoint listed twice; every `Script` empty. A node with a UTXO set (dcrd) additionally requires, for every UTXO, that the output exists, is unspent both in the chain and in the mempool, has at least one confirmation, has script version 0, and passes Algorithm 18.3 against its script, and that the sum of the UTXO values equals `InputValue` (this comparison is skipped when the sum is 0). Finally the identity must not have sent any earlier message, and for every outpoint already claimed by an accepted PR, the new PR's expiry must be strictly greater than that PR's. `mixing/mixpool/mixpool.go · checkAcceptPR, acceptPR`

Key exchange. Algorithm 18.4 holds; Pos $<$ |SeenPRs|; the epoch is a multiple of 60 seconds, not more than 5 seconds in the future and not more than 20 minutes in the past relative to the node's clock; no other KE by the same identity exists in the session; and among the referenced PRs that the node knows, the PR at `Pos` has the KE's identity and all have the same pairing description. If the PR at `Pos` is unknown, the KE is stored as an orphan. The first accepted KE of a session creates the session, whose PR list is the KE's `SeenPRs` and whose expiry is the minimum expiry of the referenced PRs known at that time. `mixing/mixpool/mixpool.go · checkAcceptKE, acceptKE`

Other messages. The limits of Table 18.16; a 32-byte session ID; no other message of the same type by the same identity in the same session. If the identity has no KE in the session, the message is stored as an orphan.

For every identity, at most `maxMessagesPerIdentity` $= 7 \times 2 \times 64 = 896$ messages are accepted, counting the PR itself, so 895 session messages. Orphans are not counted until they are promoted.

Table 18.16 Mixpool limits on message dimensions (`mixing` package)

Message	Limit
PR	`MixAmount` ≤ 525000000000000; `MessageCount` ≤ 16 (`MaxMcount`).
CT	At most 64 ciphertexts and 64 seen KEs (`MaxPeers`).
SR	At most 16 rows, each at most 1024 elements (`MaxMtot`); at most 64 seen CTs.
DC	At most 16 vectors, each at most 1024 elements; at most 64 seen SRs.
FP	At most 1024 roots; at most 64 seen SRs.
CM	Transaction serialize size ≤ 100000 (`MaxMixTxSerializeSize`); at most 64 seen DCs.
RS	At most 16 SR messages and 16 DC messages; at most 64 seen RSs.

Table 18.17 Bannable mixpool errors

Error	Condition
`ErrInvalidSignature`	Invalid message signature.
`ErrChangeDust`	Change value negative or dust.
`ErrMixDust`	Mix amount is dust.
`ErrLowInput`	Change exceeds input value, or fee below the minimum.

Error	Condition
`ErrHighFee`	Fee above the maximum.
`ErrInvalidMessageCount`	Message count 0.
`ErrInvalidScript`	Change script neither P2PKH nor P2SH.
`ErrInvalidTotalMixAmount`	Inputs less change below $m \cdot$ MixAmount.
`ErrMissingUTXOs`	No UTXOs.
`ErrPeerPositionOutOfBounds`	KE `Pos` outside `SeenPRs`.
`ErrInvalidSessionID`	Non-PR message without a 32-byte session ID.
`ErrInvalidUTXOProof`	Duplicate outpoint, failed ownership proof, or UTXO sum differs from `InputValue`. Bannable only if the sending peer advertises the full-node service flag (it could have checked the UTXOs).

Other rejections (wrong run number, expiry out of range, unknown script class, an unconfirmed or spent UTXO, conflicting messages, invalid session order, and so on) are not bannable. The source notes for several of them that a peer may relay such a message in good faith, for example a KE whose PRs it does not know.

18.9.3 Orphans

The orphan pool holds at most 250 messages. When adding an orphan would exceed that, the node counts orphans per source peer and removes orphans of the source with the most orphans, then of the next, and so on (from each source as many as are needed, up to all of them), until 187 remain; the new orphan is then added. When a PR is accepted, orphan KEs of that identity that reference it are reconsidered; when a KE is accepted, the identity's orphans in that session are accepted. Orphans are removed 20 minutes after they were stored, orphan KEs also when their epoch is more than 20 minutes old, and all orphans of an identity when its PR is removed. `mixing/mixpool/mixpool.go · addOrphan, limitNumOrphans, reconsiderOrphans`

18.9.4 Expiry and removal

When a block extends the main chain, dcrd removes from the mixpool the session of any transaction in the block that is a known completed mix, and every PR with an outpoint spent by any other transaction in the block, together with all messages of that identity and the sessions they took part in. It also schedules expiry: after waiting until the second epoch boundary from now, it removes every session whose expiry is at or below the height recorded when the wait began, and every PR whose `Expiry` is at or below it. `mixing/mixpool/mixpool.go · RemoveSpentPRs, ExpireMessagesInBackground, expireMessagesNow`

18.9.5 Interaction with the mempool and voting

Two further policies protect mixes from disruption (they belong to Chapter 17 but are driven by the mixpool):

- The mempool rejects a transaction that spends an outpoint of a current pair request unless it is the completed mix of a known session.
- The mixpool *observer* examines, at each epoch boundary, the previous epoch. Among sessions of at least four KEs that reached the CT stage and revealed no secrets, a session is complete if every peer sent a CM. An identity that sent a KE and a CT in some session of a pairing but was not in a completed mix receives a *strike* if a mix of that pairing completed; if none completed, it receives a strike only if it was one of the peers whose message was missing at the first incomplete stage of a session in which all KEs were present. Strikes are recorded against the identity's PR outpoints (merged with any strike set sharing an outpoint), removed for the PR outpoints of the peers of a session found complete and for the inputs of a known mix transaction contained in a block, and forgotten once the latest strike is at least 24 hours older than the epoch being examined. Outputs with two or more strikes (`strikeLimit`) cause the client to exclude PRs spending them, the mempool to reject non-mix transactions spending them, and dcrd not to announce the winning tickets of a block containing such a spend, so that its voting wallets do not vote on that block.

`mixing/mixpool/observer.go · checkPrevEpoch, updateStrikes, MisbehavingBlock` | `internal/mempool/mempool.go · maybeAcceptTransaction`

18.10 Constants

Table 18.18 Mixing constants

Name	Value	Meaning
`F`	$2^{127} - 1$	Field prime of the slot-reservation DC-net.
`MixMsgSize` / `Msize`	20	Size of an XOR DC-net message (a HASH160).
`MixVersion`	10	First protocol version with mixing messages.
`MSG_MIX`	4	Inventory type of mixing messages.
`MinPeers`	4	Minimum peers to proceed past KE or rerun.
`MaxPeers`	64	Maximum peers in a mixed session (mixpool and client).
`MaxMcount`	16	Maximum message count per peer.
`MaxMtot`	1024	Maximum total messages per session.
`MaxMixTxSerializeSize`	100000	Maximum CoinJoin size in bytes.
`MaxMixAmount`	525000000000000	Maximum mix amount ($21 \times 10^{14}/4$ atoms).
`MaxMixPeers` (wire)	512	Wire bound on peer counts and seen lists.
`MaxMixMcount` (wire)	1024	Wire bound on message counts and roots.
`MaxMixFieldValLen`	32	Maximum bytes of a field element.
`MaxMixPairReqUTXOs`	512	Maximum UTXOs per PR.
`MaxMixPairReqScriptClassLen`	32	Maximum script class length.
`MaxMixPairReqUTXOScriptLen`	16384	Maximum UTXO redeem script length.
`MaxMixPairReqUTXOPubKeyLen`	33	Maximum UTXO public key length.
`MaxMixPairReqUTXOSignatureLen`	64	Maximum UTXO proof length.
`ScriptClassP2PKHv0`	`P2PKH-secp256k1-v0`	Only accepted script class.
`PRFlagCanSolveRoots`	`0x01`	PR flag: peer can solve roots.
`pairingVersion`	3 (4 on testnet)	PR `PairingFlags` used by the reference client.
epoch	600 s (180 s on testnet)	Epoch duration.
`timeoutDuration`	30 s (10 s on testnet)	Stage timeout τ.
`maxJitter`	$\tau/10$	Bound on per-identity plus per-message delay.
`msgJitter`	300 ms	Per-message random delay bound.
`earlyKEDuration`	5 s	KE accepted at most this early before its epoch.
`lateKEDuration`	20 min	KE accepted at most this late after its epoch.
`minconf`	1	Confirmations required of PR UTXOs.
`feeRate`	10000 atoms/kB	Fee rate for PR fee and dust checks.
`maxRelayFeeMultiplier`	10000	Upper fee bound factor.
`maxOrphans`	250	Orphan pool capacity (187 after eviction).
`maxMessagesPerIdentity`	896	Messages accepted per identity.
`strikeLimit`	2	Strikes that flag outputs as misbehaving.
`maxRecentlyRemovedMixMsgs`	8500	Recently removed message cache size (TTL 1 min).

18.11 Test Vectors

The vectors below were produced by running the `mixing` package of dcrd at the reference commit (with its `wire`, `chaincfg` and `secp256k1` dependencies taken from the same tree). Hashes are raw bytes in hexadecimal, not display order.

TEST VECTOR Session ID and unmixed order

Six PR hashes h_k ($k = 0..5$), each 32 bytes with first byte k and the rest zero.

```
epoch 0
  preimage prefix  6465637265642d6d69782d73657373696f6e || 0000000000000000
  sid              695794d3492979b67b51cd79fa330eaae643955ecea866b3e35d7abed2ec621e
  unmixed order    h1 h0 h3 h2 h5 h4
epoch 1763251200 (2025-11-16T00:00:00Z)
  preimage prefix  6465637265642d6d69782d73657373696f6e || 0000000069191400
  sid              0a2ebbf690ff74794c4fe8cd626187ee6464739fb0bb92e86d547841ce70a615
  unmixed order    h2 h3 h0 h1 h4 h5
```

TEST VECTOR Signed pair request

The PR of dcrd's own signature test: identity key with private scalar 1, `Expiry` =15, `MixAmount` =16, `ScriptClass` ="script class", `TxVersion` =17, `LockTime` =18, `MessageCount` =19, `InputValue` =20, one UTXO (hash of 31 zero bytes then `01`, index 2, tree 3, script `040506`, `PubKey` and `Signature` each 31 zero bytes followed by `07` and `08`, opcode 9), change (value 10, version 11, script `0c0d0e`), `Flags` =21, `PairingFlags` =22. This message is well-formed on the wire but not acceptable to a mixpool.

```
payload (266 bytes)
  1ffdddfb26b2955b3cf46c9e98671d11d1fd91619da4e85edb840786f661b6ab
  6824c83923f596a990613fce97d995d9d50a89ecb23841c9c10a80a47f18b50a
  0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
  0f000000 1000000000000000 0c73637269707420636c617373 1100 12000000
  13000000 1400000000000000 01
  0000000000000000000000000000000000000000000000000000000000000001
  02000000 03 03040506
  2000000000000000000000000000000000000000000000000000000000000007
  2000000000000000000000000000000000000000000000000000000000000008
  09 01 0a00000000000000 0b00 030c0d0e 15 16
signed-data hash   dc059c0a267ffe3519b6fd44200a62230af86646df1990cf0316e492de3235f5
digest preimage    decred-mix-signature,mixpairreq,<64 zeros>,0,dc059c0a...de3235f5
signature digest   bd117cb3e25960bfe803274052411ce01ed3dcf5cf1f0dfe61f5d8f8515f4b24
signature          1ffdddfb26b2955b3cf46c9e98671d11d1fd91619da4e85edb840786f661b6ab
                   6824c83923f596a990613fce97d995d9d50a89ecb23841c9c10a80a47f18b50a
message hash       78cbe3e350243d0d220e92aa188a7c9c419321fd9d8233e7a0730d16bce47f63
pairing            1000000000000000 0c73637269707420636c617373 1100 12000000 16
```

TEST VECTOR UTXO ownership proof

Private key scalar 1, expiry 15.

```
preimage   6d697870722d7574786f70726f6f662c5032504b482845432d5363686e6f72722d
           4443527630292c 0000000f
           0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
digest     992e8aa407418e1b8c80e8b09a5fc974e809bd4a91dfdcb1aade88cf1a2cdd1e
signature  8e04cc6dba1be56419f3373755cb40ea4584c5fbdfa7e9dd4061aff1cda38189
           6ec744946fc45fe70f954c1fb5919e2d4cffbe7dd27797cf03b26c2f7e1a01be
```

TEST VECTOR ECDH component of a pairwise key

Private scalars 2 and 3.

```
x(6G)        fff97bd5755eeea420453a14355235d382f6472f8568a18b2f057a1460297556
H(x(6G))     cedb592ff1f23633afcdb634a11364e81b9efdfbdf2feeedc3b62969f2988a4b
```

TEST VECTOR Message-pair PRNG (Algorithm 18.6)
`sid` = 32 bytes `11`, combined key = 32 bytes `22`, counter 1, nonce 2, $M = 3$.

```
seed preimage  11..11 (32) || 22..22 (32) || 01000000
seed           57fe2217c2c6b52edbd184ad99b76396bb57b71fd0ca1931ab9ed2abdf2487e9
SR secret      b312257a35d4a80604045b2e26c257799c118c562ce57da8a0bfd879549cfcc7
DC pad         77ba2e62c31f8cd28bcf08238c20fe7a9d6ddaae
               877e64054adc223569c8684d00a5e0bc4ea0c2e9
               d2a6ab5fd54d37e54ac00225e412fc80e4ce12aa
PRNG(key 000102..1f, n=0), first 64 bytes
  39fd2b7dd9c5196a8dbd0377b8dc4a498a35d86fbcde6accb2cc7d4cd8ea2492
  2b23cce7a26023ab3f0eef693ac87f64258235eab1f7a32dc22762a0485b410c
PRNG(key 000102..1f, n=1), first 32 bytes
  d838fb09536e2e3a10e8f23f486273a69f42d8e640d781ede384793c34c32564
```

TEST VECTOR Slot-reservation DC-net with three messages
Three peers with one message each ($M = 3$), $r_0 = 3$, $r_1 = 5$, $r_2 = 7$, and SR secrets $K_{0,1}$ = 32 bytes `01`, $K_{0,2}$ = 32 bytes `02`, $K_{1,2}$ = 32 bytes `03` (tiny values for illustration; real messages are uniform in $[0, F)$). Field elements in minimal big-endian hex.

```
H(K01 || le64(1))  8d0647c6c0a02d2043415ed185424366a964699e1388b4813588ef8b1f1f09a5
peer 0 pads   5cb3d0e7d64cf4d5783290eb9d1eee93 49e72f508b78bf99eabaac0ba4ade141
              17996cc15b8e659faae5a1e77ea8ecc5
peer 0 mix    5cb3d0e7d64cf4d5783290eb9d1eee96 49e72f508b78bf99eabaac0ba4ade14a
              17996cc15b8e659faae5a1e77ea8ece0
peer 1 mix    14421511451bf2841894680d1661e9b2 54b3ef8a640d1127a38dc5d8780f9cea
              6c7e965d9e0f0f9957befc2137d1dc65
peer 2 mix    0f0a1a06e49718a66f3907074c7f27c6 6164e125107a2f3e71b78e1be342821d
              7be7fce106628ac6fd5b61f7498538a8
sum           S1 = 15, S2 = 83, S3 = 495
coefficients  a0 = 105, a1 = F - 71, a2 = 15, a3 = F - 1
              (phi(x) = -(x - 3)(x - 5)(x - 7))
is_root(3), is_root(5), is_root(7) = true; is_root(4) = false
```

TEST VECTOR Secrets commitment
Identity of private scalar 1, `sid` = 32 bytes `11`, run 0, seed = 32 bytes `33`, SR messages [3], DC messages [20 bytes `44`].

```
hashed bytes  00 x 64 || 0279be66..16f81798 || 11 x 32 || 00000000 || 33 x 32
              || 01 01 03 || 01 14 44 x 20 || 00
commitment    1255ff216f8ea06b900073f40cc5c747727f00eb8c5f1a66554838be61c86cf8
```

TEST VECTOR Key derivation from a seed (Section 18.7.2)
Seed = 32 bytes `55`. Then one encapsulation, from the continuing PRNG, to the peer's own sntrup4591761 public key. The sntrup4591761 values depend on the comparison used by the sorting network (see the pitfall in Section 18.2.1).

```
ECDH private   b02358b37e4f6868fb48af8ab2750b062aa95a831bc21978a98e944264fa0e75
ECDH public    02c6febf4681819e413f0ace75200a67d0fe3d0c98a15856b3a56342573c52d11e
unsigned comparison (Go 1.26.0, 1.26.11, 1.27.4 on amd64):
  H(PQ public key)   7f9f922f9089ae5b09e67cb92da4586bfc40b272f45e8ee3426db0a0dd46f272
  PQ public [0:16]   3190d480e196ca264dffac85c074c554
  H(ciphertext)      c408d531daed383c66dd6394f171bf21bbe044d6414e940060efe62096664443
  shared key         23364e2d3cd93f7527e9e77bd7d18e8c3a94ca6063f8c466030c699d41b52970
Go 1.25 comparison (Go 1.24.13, 1.25.0, 1.25.14 on amd64; 1.25.14 on 386):
  H(PQ public key)   fd71ff70b51595ea21392e96c9b610d81b9469444668e385e9bba1f9d2cd8921
  PQ public [0:16]   2f1d55f11730c71c872cd85f55fafb34
```

```
  H(ciphertext)      0bf356f86e997d15c238b665f9844be366150a47b3bd8f7ed0f22829c02d53be
  shared key         3328923bfcceee6b4728e91ec67e72f17575ce64a9fbaa77eba2e9d4e416cd5e
signed comparison (Go 1.26.11, 1.27.4 on 386):
  H(PQ public key)   ddd0c09f8460c9c66d437692c4c0f60dc0c26ad6b8a0cabe55dab7551153ccc7
  PQ public [0:16]   eae67aa5191d2a37a72e01d63a4af017
  H(ciphertext)      0ae73fd9549edeae907b807a5992e4b4627fdce0b870c226559fe507a1dc9416
  shared key         f56dff234601d5ad1fe447faa04f4795569ee81e0a003f26f02aba4ed8f0f592
```

TEST VECTOR Policy arithmetic

```
MaxExpiry(1000): mainnet 1013, testnet3 1031, simnet 4601
smallest non-dust P2PKH value: 6030 atoms
size(1 in, 1 out, no change) = 217   min fee 2170   max fee 21700000
size(2 in, 4 out, change)    = 527   min fee 5270   max fee 52700000
size(5 in, 16 out, change)   = 1457  min fee 14570  max fee 145700000
```

CHAPTER 19

Governance and Ecosystem

The preceding chapters specify the consensus protocol. This chapter describes, at overview level, the systems built around it that give Decred its character: the governance process that decides what the treasury pays for and which rule changes are proposed, the voting service providers that let ticket holders vote without running servers, the wallet and its light-client mode, payment channels, atomic swaps and the decentralized exchange, and the timestamping service that anchors off-chain data in the chain.

> **NOTE**
> This chapter is informative. Nothing in it affects the validity of any block or transaction. Where a system relies on a consensus rule, the rule is specified in the chapter referenced; where a system has its own parameters (Politeia vote thresholds, VSP fees, DEX epochs), those are properties of that software, chosen by its operators, and can change without any change to the consensus rules.

Facts in this chapter are taken from the Decred documentation [37], the DCPs, dcrd, and the source and documentation of the software concerned ([55], [56], [22], [57], and the `dcrlnd` and `dcrtime` repositories). The documentation is in places older than the software; where the two disagree, the software is followed and the difference is noted.

19.1 The Governance Model

Decred's governance rests on a single credential: the ticket (Chapter 8). A stakeholder time-locks coins to buy a ticket, and the ticket confers a vote. That vote is exercised in four places, which differ in whether the outcome is enforced by consensus.

Block approval Each block includes votes from the tickets the lottery selected for it. Of the `TicketsPerBlock` $= 5$ tickets selected, a block at or above the stake validation height must contain votes from at least $\lfloor 5/2 \rfloor + 1 = 3$, and the votes decide whether the regular transaction tree of the previous block, including its proof-of-work subsidy, stands: it is approved only if a strict majority of the included votes approve it (Chapter 10). This is the check stakeholders hold over miners.

Consensus rule changes Votes also carry the ticket holder's preference on each open *agenda*, a dormant rule change shipped in node software and specified by a DCP. Agendas are tallied over rule change intervals of `RuleChangeActivationInterval` $= 8064$ blocks; an agenda locks in when at least `RuleChangeActivationQuorum` $= 4032$ non-abstaining votes are cast in an interval and at least `RuleChangeActivationMultiplier`/`RuleChangeActivationDivisor` $= 3/4$ of them choose it, and the new rules become active one interval later (Chapter 13).

Treasury spends Since DCP-0006 the treasury is an account that can be debited only by a *treasury spend* (tspend) transaction. A tspend is authored and signed by the holder of one of the well-known Politeia keys (`PiKeys`), but it is valid only after enough ticket votes approve it on chain (Chapter 12).

Proposals Decisions about the direction of the project, and in particular which work the treasury should fund, are made by off-chain ticket votes on proposals submitted to Politeia (Section 19.2). These votes are cryptographically verifiable but are not enforced by the consensus rules.

The first three are part of the consensus protocol; the fourth is a social process that culminates in the third. Table 19.1 compares them, and Figure 19.1 shows how a funding proposal and a consensus change each travel from idea to effect.

Table 19.1 The four uses of a ticket vote. Values are main-network values.

	Block approval	Rule change agenda	Treasury spend	Politeia proposal
Where	Vote transaction, every block	Vote bits in vote transactions	Treasury vote data in vote transactions	Signed message to Politeia server
Who votes	The (at most 5) tickets selected for the block	Every ticket that votes in the interval	Every ticket that votes in the window	Every ticket live in the snapshot
Duration	One block	8064 blocks per interval	$288 \times 12 = 3456$ blocks	2016 to 4032 blocks
Quorum	At least 3 votes in the block	4032 non-abstaining votes	1/5 of the maximum votes in the window	20% of eligible tickets (by convention)
Threshold	Strict majority approve	75% of non-abstaining votes, in integer arithmetic (Chapter 13)	60% yes, with votes still possible counted as no	60% of votes cast (by convention)
Enforced by	Consensus	Consensus	Consensus	Not enforced; signals intent

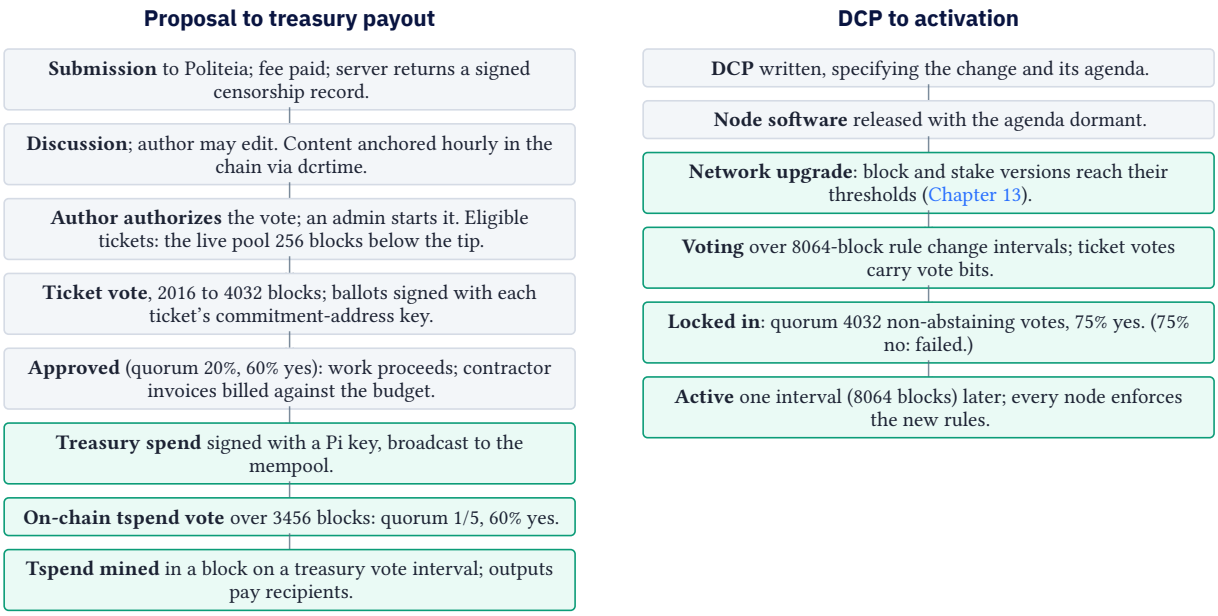


Figure 19.1 How a funding proposal (left) and a consensus rule change (right) take effect. Shaded steps are enforced by the consensus rules; the rest are off-chain processes.

HISTORY
Before DCP-0006 activated, the treasury share of each block subsidy was paid to a fixed multisignature address held by Decred Holdings Group LLC (DHG), and contractor invoices were paid by DHG by hand; DCP-0006 describes this process and replaces it with stakeholder-approved treasury spends. The documentation's description of Politeia still refers to payment by DHG (Politeia overview); that text predates DCP-0006.

19.2 Politeia

Politeia is the system in which proposals are submitted, discussed and voted on. Its README describes it as a store of off-chain data that is versioned and timestamped, "git plus timestamping" [55]: rather than putting governance data on chain, Politeia keeps it off chain and anchors commitments to it in the Decred chain. Proposals are of two kinds: those that seek stakeholder support for a course of action or a policy, and those that request treasury funding for a budget that the proposer draws down against as work is delivered.

19.2.1 Components and identities

The server side consists of `politeiad`, the data layer, and `politeiawww`, the user-facing web backend; `politeiad` relies on a `dcrtime` server (Section 19.7) for anchoring. Functionality is organized as plugins: general-purpose plugins such as `comments` and `ticketvote`, and an application plugin (`pi` for the proposal system) that adds behaviour specific to the application.

Every Politeia user has an Ed25519 key pair, generated in the browser at registration. The user signs every submission (proposal, edit, comment, comment up/down vote, vote authorization) with it, and the server returns a *receipt*, its own Ed25519 signature over the user's signature. Each side can therefore prove what the other said.

19.2.2 Censorship resistance

Administrators can censor submissions, for spam or for content the system cannot host, but only transparently. When a record is submitted the server returns a *censorship record* consisting of a random token identifying the record, the Merkle root (SHA-256) of the record's files, and the server's signature over the Merkle root and token. A user whose proposal is censored can publish the censorship record and the files: the server's signature proves that the server received exactly that content, and the anchoring described next proves when. Censored proposals remain visible to their author with the reason given. The `politeiaverify` tool checks censorship records, receipts, user signatures and timestamps from the bundles Politeia offers for download.

19.2.3 Anchoring in the chain

The current backend stores each record in an append-only, Merkle-tree-backed log (a Trillian log, replacing the original git backend in 2021). At minute 56 of every hour, `politeiad` collects every log that has leaves not yet anchored and submits a SHA-256 digest of each log's current root to `dcrtime`. `dcrtime` aggregates the digests it receives into a single Merkle root and publishes it in a transaction (Section 19.7). `politeiad` considers the anchor complete once `dcrtime` reports the block timestamp of the anchoring transaction, which it does only after the transaction has 6 confirmations, and then appends an anchor leaf to each anchored log. The result is a chain of evidence from any piece of Politeia data to a block: a Merkle inclusion proof into the log root, a Merkle path from the log root's digest to the `dcrtime` root, and the transaction that carries that root.

> **NOTE**
> The documentation describes the earlier git backend, in which data was committed to a public git repository and each commit hash was anchored hourly ([37], "dcrtime" and "Navigating Politeia Data"). The principle is unchanged; only the data structure differs.

19.2.4 Proposal lifecycle

1. **Registration and submission.** Creating an account and submitting a proposal each require a payment in DCR to an address the server provides (a registration "paywall" and per-proposal credits). The documentation gives both fees as 0.1 DCR, subject to change if they fail to deter spam or the price of DCR changes significantly. A funding proposal states the amount requested (in US cents), start and end dates, and a domain.
2. **Review.** An administrator reviews the proposal and either makes it public or censors it.
3. **Discussion.** A public proposal can be discussed and its author can publish new versions in response. Comment up/down votes are public and only affect display order. An administrator may mark an inactive proposal *abandoned* until the author authorizes voting.
4. **Authorization and start.** The author authorizes the vote by signing the proposal token, version and action; after this the proposal can no longer be edited. An administrator then starts the vote by signing the vote parameters: duration in blocks, quorum percentage, pass percentage and the vote options (`yes` and `no` with their bits).
5. **Voting.** Ticket holders vote from their wallets (Decrediton or the `politeiavoter` tool) for the duration of the vote.

6. **Outcome.** When the end height is reached the proposal is *approved* or *rejected*. An approved funding proposal becomes *active*, is billed against as work is delivered, and is finally marked *completed*, or *closed* by an administrator if the author fails to deliver.

A second vote type, the *runoff*, lets several proposals submitted in response to a request for proposals compete: each is voted on normally, and the one that meets quorum and pass requirements with the most net yes votes wins.

19.2.5 Ticket voting

When an administrator starts a vote, the server takes a snapshot of the live ticket pool at height $h - \texttt{TicketMaturity}$, where h is the best block height and $\texttt{TicketMaturity} = 256$ on the main network; the code's comment explains the offset as moving the snapshot into territory that will not be reorganized. The snapshot block's height and hash become the vote's start, the list of ticket hashes in the pool at that block is recorded as the eligible set, and the end height is the snapshot height plus the duration plus `TicketMaturity`. On the main network the duration must lie between 2016 and 4032 blocks; at the 5-minute target block time, 2016 blocks is one week. A ticket bought after the snapshot cannot vote, and a ticket that is called to vote on chain during the vote can still vote on the proposal until the end.

A vote, called a ballot entry, names the proposal token, the ticket hash and the chosen vote bit, and carries a signature over the concatenation `token || ticket || votebit` (each as its hex string). The signature is a Decred signed message (a compact, public-key-recoverable secp256k1 ECDSA signature over the BLAKE-256 hash of the length-prefixed string `"Decred Signed Message:\n"` followed by the length-prefixed message) made with the key of the ticket's *largest commitment address*: the address, among the ticket's reward commitments, that committed the largest amount (Chapter 8). The server verifies that the ticket is in the eligible set and has not already voted, looks up the commitment address from the ticket transaction, verifies the signature, and returns a receipt.

The commitment address, not the voting address, signs because the commitment address belongs to the owner of the funds. A ticket whose on-chain voting rights are delegated to a voting service provider (Section 19.3) is still voted on Politeia by its owner.

Results are verifiable by anyone. The vote parameters, the eligible ticket list and every signed ballot are public; each ballot can be checked against the ticket's commitment address taken from the chain, each ticket against the snapshot block, and all of it against the anchors. The approval rule is that the number of votes cast is at least the quorum percentage of the eligible tickets and the `yes` votes are at least the pass percentage of the votes cast. The percentages are parameters of each vote; the documentation and the Decred constitution give them as 20% quorum and 60% approval, with a default duration of 2016 blocks. To reduce timing correlation between a voter's tickets, `politeiavoter` can trickle votes over the voting period, which is effective only when it connects through Tor.

19.2.6 What is and is not binding

A Politeia vote changes no consensus state. Its force comes from the people who act on it: approved funding proposals are paid by treasury spends, which the Pi key holders author and which ticket holders must then approve on chain (Section 19.1). Stakeholders therefore vote on funding twice, once on the proposal and again on each tspend, and the second vote is the binding one. Policy proposals, including amendments to the Decred constitution, bind only by convention. Consensus rule changes are not decided on Politeia; they are decided by agenda votes (Chapter 13), although a proposal may be used to gauge support before a DCP is written.

19.2.7 Contractor management

Most treasury spending pays contractors: individuals and companies who work on the project at agreed rates and invoice monthly in US dollars, converted to DCR at the previous month's average rate. Since April 2019 contractors have submitted invoices through the Contractor Management System (CMS), a separate application built on Politeia, where administrators approve, dispute or reject them. A stakeholder-approved proposal (the Decred Contractor Clearance) defines how new contractors are admitted, by agreement of

established contractors in the relevant domain, with disputes escalating to a vote of all contractors and ultimately to stakeholders. The constitution states that contractors are autonomous: stakeholders approve or reject programs of work through Politeia but cannot direct individual contractors. The first payment to contractors by treasury spend was made in December 2021.

19.3 Voting Service Providers

19.3.1 Why they exist

A ticket earns its reward only if a vote for it appears in the block in which it is selected. The wallet holding the ticket's voting key must therefore be online, unlocked and connected to a full node at an unpredictable moment, and must broadcast the vote quickly enough for the next block to include it; a ticket whose vote is not included is *missed* (Chapter 8). A voting service provider (VSP) runs always-online voting wallets on behalf of ticket holders who do not want to.

19.3.2 Delegation without custody

The ticket's output structure is what makes delegation safe. Output 0, tagged `OP_SSTX`, is the *voting rights* output: its script (pay-to-pubkey-hash or pay-to-script-hash) is the script that must be satisfied to vote. The following outputs are *reward commitments*: each commits to an address and the amount contributed, and the consensus rules require every vote and revocation spending the ticket to pay the original amounts plus any reward to exactly those commitment addresses (Chapter 8). Control of the voting key is thus control over how and whether the ticket votes, and nothing else. A holder of the voting key cannot redirect the funds or the reward.

Since DCP-0009, a missed or expired ticket is revoked automatically: every block must contain revocations for the tickets that become missed or expired in it, and those revocations have empty signature scripts, pay no fee, and pay the commitment addresses. A ticket whose voter fails therefore returns its locked funds to the owner without any action by the owner or the VSP, losing only the vote reward.

19.3.3 vspd

The current VSP software, `vspd` [56], released in 2020, accepts any ticket: the ticket needs no VSP-specific outputs, and the VSP fee is paid by a separate ordinary transaction. Registration over vspd's HTTP API proceeds as follows.

1. The client fetches the VSP's information, including its fee percentage and its Ed25519 public key, with which the VSP signs every response.
2. The client requests a fee for a specific ticket, sending the ticket and its parent transaction. The ticket must be in the mempool, immature or live. The VSP replies with a fee amount, valid for one hour, and a fee address unique to that ticket, derived from an extended public key of a separate fee wallet.
3. The client sends a signed transaction paying that fee to that address, together with the private key of the ticket's voting address and its voting preferences: choices for each consensus agenda, and yes or no for individual treasury spends (by hash) or for all tspends signed by a given Pi key.
4. The VSP holds the fee transaction until the ticket has 6 confirmations, then broadcasts it; once the fee transaction itself has 6 confirmations, the VSP imports the voting key into each of its voting wallets.

Every client request that names a ticket carries a signature by the ticket's commitment address (or an alternate signing address registered once for the ticket), and every response carries the VSP's signature. This *two-way accountability* lets either side prove the other's statements: a user can show that the VSP acknowledged a paid fee and the preferences it was given, and the VSP can show the preferences the user actually set.

The fee is computed by the wallet library's stake pool fee function from the VSP's fee percentage p, the current ticket price v, a fixed relay fee $z = 10^4$ atoms and an estimate s of the vote subsidy when the ticket is likely to vote: the current per-vote subsidy reduced by the factor $100/101$ once for each of $\lceil$`TicketPoolSize` / `SubsidyReductionInterval`$\rceil$ subsidy reductions (2 on the main network). The fee is

$$f = \left\lfloor \frac{ps(v+z)}{s+v} \right\rfloor \tag{19.1}$$

evaluated in integer arithmetic with p truncated to hundredths of a percent. It is the share p of the expected reward, scaled by the ticket's contribution.

A vspd deployment consists of a front-end server and at least three voting servers, each running `dcrwallet` against its own `dcrd` (SPV is not supported for voting), ideally in different locations. The voting wallets run with manual ticket management so that only tickets registered and paid for through vspd are voted.

19.3.4 Trust model and failure modes

A VSP cannot spend a ticket's funds or redirect its reward, and it cannot make a ticket more or less likely to be selected; the lottery is not influenced by which wallet holds the voting key. Pooling at a VSP is therefore not pooling of rewards, and the number of tickets a VSP holds does not change how often any of them votes. What the user does trust the VSP to do is:

- **Vote at all.** If every voting wallet of the VSP is unavailable, the ticket is missed and later revoked automatically; the owner recovers the ticket price but not the reward. Some misses are caused not by the VSP but by miners: a block found very shortly after its parent may be mined before votes for it have propagated.
- **Vote as instructed.** The VSP could vote delegated tickets against their owners' preferences. Two-way accountability makes this provable, and the documentation recommends spreading tickets across smaller VSPs to limit the influence of any one of them.

Because the owner's wallet keeps the voting key, the owner can vote the ticket directly if a VSP disappears.

> **HISTORY**
> Before vspd, VSPs ran `dcrstakepool`, where the user and VSP created a 1-of-2 multisignature *redeem script* that held the ticket's voting rights, and tickets were purchased with dedicated outputs for paying the VSP fee. A user who lost the redeem script while the VSP was unavailable could neither vote nor revoke; automatic revocations (DCP-0009) were motivated in part by recovering those funds.

19.3.5 Solo staking

A ticket holder can instead vote alone: buy tickets whose voting address belongs to one or more always-online "hot" voting wallets, while the funds and the reward commitments stay with a separate wallet that need not be online. Running several voting wallets from the same seed on servers in different places gives redundancy; this is safe because voting creates no new addresses. Solo staking avoids the VSP fee and the trust in the VSP at the cost of operating the servers.

19.4 Wallets and Light Clients

`dcrwallet` [22] is the reference wallet daemon; the graphical wallet Decrediton and other official wallets use it underneath. It holds keys, builds and signs transactions, buys tickets, votes, mixes, and talks to VSPs, and is controlled over JSON-RPC and gRPC interfaces. Private keys are derived hierarchically by BIP-32 [24] along BIP-44 paths `m/44'/coin_type'/account'/branch/index` [26] and are never written to disk unencrypted (Chapter 5).

19.4.1 Two ways to follow the chain

By default the wallet relies on a trusted `dcrd` over RPC. Alternatively, in SPV mode it connects directly to peers of the peer-to-peer network and uses the version 2 committed compact filters (Chapter 16):

1. It downloads and validates the header chain, including proof of work.
2. For each block it downloads the block's filter together with an inclusion proof, and checks the filter against the commitment root in the block header introduced by DCP-0005. For blocks before that commitment existed, it checks the whole set of filters against a hash built into the wallet.

3. It matches the filters locally against its own scripts and downloads only the blocks that match, from which it extracts its transactions.

No address or extended public key leaves the wallet, unlike light wallets that query a server, and because the filters are committed in proof-of-work protected headers, a peer cannot hide a transaction by serving a false filter. An observer sees only which blocks the wallet fetches, which can be spread across peers. An SPV wallet cannot vote, because a vote approves or disapproves the previous block's regular transactions and only a fully validating node can judge them; SPV users delegate voting to a VSP or to their own full-node voting wallets.

19.4.2 Accounts and privacy

Funds are organized in BIP-44 accounts. When mixing is enabled the wallet uses dedicated accounts: a *mixed* account (and branch) to which mixed outputs and voting rewards are paid, a *change* account that receives unmixed change, which can itself be mixed into the mixed account, and a *voting* account from which voting addresses are derived. Mixing uses the StakeShuffle protocol, whose messages full nodes relay over the peer-to-peer network (Chapter 18). The protocol breaks the link between inputs and outputs of equal value but does not hide amounts, so the wallet mixes fixed denominations and keeps change away from mixed funds.

Mixing is integrated with staking. When buying tickets with mixing enabled, the wallet funds them through a *mixed split*: it joins a mix whose outputs are each exactly the amount needed for one ticket, so that the ticket purchases that follow cannot be linked to the coins that paid for them. Because every ticket in the same price window costs the same, ticket purchases form a natural anonymity set.

19.5 Payment Channels and the Lightning Network

The Lightning Network [58] moves payments off chain. Two parties lock funds in a 2-of-2 multisignature *funding* output and then exchange signed but unbroadcast *commitment transactions* that spend it, each reflecting the current division of the channel balance and any in-flight payments. Either party can close the channel by broadcasting the latest commitment transaction. Broadcasting an older, revoked state is a *breach*, which the other party punishes with a *justice transaction* that takes the whole channel balance, provided it notices within the time allowed. Payments are routed across chains of channels using hash time-locked contracts (HTLCs): each hop can claim its incoming payment only by revealing a preimage of the payment hash, and each hop's timeout is shorter than the one before it.

19.5.1 Base-protocol support

Lightning needs four things from the base protocol, all of which Decred provides.

Stable transaction identifiers Commitment transactions are signed before the funding transaction is broadcast and refer to the funding output by its transaction hash. In Bitcoin before segregated witness, a third party could re-encode a signature, changing the transaction hash and invalidating every pre-signed child. Decred separates every transaction into a *prefix* (inputs' outpoints and sequence numbers, outputs, lock time and expiry) and a *witness* (each input's value, block height, block index and signature script; the documentation describes the first three as fraud-proof data), and the transaction hash used in outpoints is the BLAKE-256 hash of the prefix serialization alone (Chapter 4). Nothing in the signature scripts can change it, and changing the prefix invalidates the signatures, which commit to the prefix (Chapter 6). Decred has had this property since its launch. The block Merkle tree commits to the full transaction by hashing the prefix hash together with the witness hash.

Relative time locks Revocable commitment outputs must be spendable by their owner only after a delay counted from when the commitment transaction is mined, so that the counterparty has time to react to a breach. DCP-0003 gives transaction-input sequence numbers consensus meaning as relative lock times (in blocks, or in units of 512 seconds) for transactions of version 2 or later, and introduces `OP_CHECKSEQUENCEVERIFY` (`0xb2`), which lets a script require such a delay (Chapter 7, Chapter 6). DCP-0004 later corrected the set of outputs visible to the sequence-lock check so that the feature is fully usable.

Absolute time locks HTLC timeouts use `OP_CHECKLOCKTIMEVERIFY` (`0xb1`), enforced since launch. The opcode compares its operand with the transaction's lock time; DCP-0003 changed the rule that decides when a time-based lock time has passed, which now uses the median time of the previous 11 blocks (Chapter 7).

Hash locks compatible with other chains Decred's `OP_HASH160` computes RIPEMD-160 of BLAKE-256, not of SHA-256 as in Bitcoin. DCP-0002 adds `OP_SHA256` (`0xc0`) so that Decred scripts can commit to the same SHA-256 hashes as other chains, which cross-chain swaps and interoperable payment hashes require.

DCP-0002 and DCP-0003 were deployed together under the `lnfeatures` agenda, active on the main network from block 189568; `dcrd` enables `OP_CHECKSEQUENCEVERIFY` and `OP_SHA256` in script validation for blocks for which the agenda is active (Chapter 13). DCP-0004 (agenda `fixlnseqlocks`) became active at block 342784.

19.5.2 dcrlnd

`dcrlnd` [59] is a port of the `lnd` implementation, changed for Decred's transaction serialization, witness format and signature process, and aiming at compliance with the Lightning specifications (BOLTs) [60]. Its HTLC scripts use `OP_SHA256 OP_RIPEMD160` where Bitcoin's use `OP_HASH160`, reproducing Bitcoin's RIPEMD-160-of-SHA-256 payment-hash commitment, and its delayed outputs use `OP_CHECKSEQUENCEVERIFY`. Nodes that cannot stay online can use watchtowers, services that watch the chain for breaches and broadcast justice transactions on their behalf; the documentation notes that watchtowers can recover only funds already committed in the channel, not in-flight payments.

19.6 Atomic Swaps and DCRDEX

19.6.1 Hash time-locked contracts

An atomic swap exchanges coins on two chains without a trusted intermediary; Decred's command-line tools for it are in the `atomicswap` repository [61]. The initiator chooses a secret, and both parties lock their coins in contracts on their respective chains that pay the other party on presentation of the secret, or refund the funder after a timeout. When the initiator redeems the participant's contract, the secret appears in the redeeming transaction's signature script, and the participant uses it to redeem the initiator's contract. The initiator's timeout is the longer, so the participant always has time to redeem after the secret is revealed. Both chains must support branching scripts, a common hash function, signature checks and a time lock; the shared hash is the reason for `OP_SHA256`.

Table 19.2 shows the contract DCRDEX uses on Decred, paid to by pay-to-script-hash. It is DCP-0002's example HTLC with an added check that the secret is exactly 32 bytes. Each opcode is listed with its value in dcrd's script engine (Appendix A).

Table 19.2 Atomic swap contract on Decred, as constructed by DCRDEX (`dex/networks/dcr/script.go`, `MakeContract`).

Script	Bytes	Effect
`OP_IF`	`63`	Redeem branch if the top stack item is true.
`OP_SIZE 32 OP_EQUALVERIFY`	`82 01 20 88`	The secret is exactly 32 bytes.
`OP_SHA256 <secret hash> OP_EQUALVERIFY`	`c0 20 ... 88`	SHA-256 of the secret equals the committed 32-byte hash.
`OP_DUP OP_HASH160 <recipient PKH>`	`76 a9 14 ...`	Push the recipient's 20-byte public key hash for the check below.
`OP_ELSE`	`67`	Refund branch.
`<lock time> OP_CHECKLOCKTIMEVERIFY OP_DROP`	`... b1 75`	The spending transaction's lock time has reached the contract's.
`OP_DUP OP_HASH160 <sender PKH>`	`76 a9 14 ...`	Push the sender's public key hash.
`OP_ENDIF`	`68`	
`OP_EQUALVERIFY OP_CHECKSIG`	`88 ac`	The supplied public key hashes to the selected hash and the signature is valid.

The recipient redeems with the signature script `<sig> <pubkey> <secret> 1 <contract>`; the sender refunds after the lock time with `<sig> <pubkey> 0 <contract>`. The last push in each is the serialized contract, which pay-to-script-hash checks against the output's script hash and then executes (Chapter 6).

19.6.2 DCRDEX

DCRDEX [57] is a non-custodial exchange in which a server matches orders but never holds funds; settlement is by atomic swaps directly between the two traders' wallets, each swap consisting of four on-chain transactions (two contracts and two redemptions).

- **Matching in epochs.** Orders are not processed as they arrive. Each order carries a commitment, the BLAKE-256 hash of a random 32-byte preimage chosen by the client. Orders received during an epoch of fixed length (set per market) are collected; at the end of the epoch clients reveal their preimages, and the server seeds a pseudorandom generator with the BLAKE-256 hash of the concatenated preimages (in order-ID order) and shuffles the orders before matching them one by one against the book. The preimages, seed and a checksum of the commitments are published so that clients can check that no order was omitted or reordered. The aim is to deny any client a deterministic latency advantage, and the server any undetectable discretion over matching.
- **Settlement.** The maker initiates: it creates the secret and broadcasts its contract first, with the longer lock time (20 hours in the specification's example, against 8 hours for the taker). The taker checks the maker's contract and its lock time, then broadcasts its own contract with the same secret hash. The maker redeems the taker's contract, revealing the secret, and the taker redeems the maker's. The server relays swap details and monitors compliance but never controls the funds: a contract that is not redeemed is refunded to its funder after its lock time.
- **Fees and accountability.** DCRDEX collects no trading fees, so there is no exchange token and no fee transaction; traders pay only the on-chain transaction fees, at a minimum rate set by the operator. Instead, a client must lock funds in a time-locked *fidelity bond* to trade, and a client that breaks the rules of conduct (failing to settle a match, excessive cancellations) loses trading privileges. On Decred a bond is an output paying to the following script, spendable by its owner only after the lock time:

```
<lock time> OP_CHECKLOCKTIMEVERIFY OP_DROP
OP_DUP OP_HASH160 <pubkey hash> OP_EQUALVERIFY OP_CHECKSIG
```

> **NOTE**
> The DCRDEX README calls the bond requirement a one-time registration fee in one place and bonds in another; the specification describes bonds. The README also describes Tatanka Mesh, a peer-to-peer successor in development in which a network of nodes, rather than a central server, maintains order books and reputation.

19.7 Timestamping with dcrtime

`dcrtime` [62] is a timestamping service, inspired by OpenTimestamps, that proves that data existed at a given time by anchoring its hash in the Decred chain. It was written for Politeia and is also offered as a free public service.

A client submits 32-byte SHA-256 digests. The server collects them for one hour; shortly after each hour it computes a SHA-256 Merkle root over the digests collected and has a wallet publish a transaction with an output `OP_RETURN <root>`. Once the transaction is mined, the server can return, for any digest, the Merkle path from the digest to the root, the transaction hash and the block time (the chain timestamp). A verifier recomputes the root from the digest and the path, finds the transaction in the chain and reads the root from its null-data output; the `dcrtime` client verifies the path, and a separate tool checks anchors against the chain. Any number of digests costs a single transaction per hour. The proof shows only that the data existed no later than the block containing the anchor; the collection time the server reports is the server's claim.

The deployment separates a proxy, which faces the internet, from the backend, which holds the wallet credentials and builds the anchoring transaction, so that the process with access to funds is not directly exposed.

19.8 Dependence on the Consensus Protocol

None of these systems adds consensus rules, but each relies on guarantees the rest of this document specifies:

- **Ticket structure and the lottery** (Chapter 8). Non-custodial VSPs exist because the voting-rights output and the reward commitments are separate and votes and revocations must pay the commitments. Politeia's eligibility rule depends on the live ticket set being a well-defined function of the chain, and its vote signatures on the commitment addresses recorded in each ticket. Automatic revocation (DCP-0009) bounds the damage a failed VSP can do.
- **Stake votes and the treasury** (Chapter 10, Chapter 13, Chapter 12). The proposal process has binding force only because the treasury can be debited solely by tspends that ticket holders approve in their vote transactions, and Pi keys are fixed in the network parameters.
- **Header commitments and compact filters** (Chapter 16). SPV wallets can trust filters because DCP-0005 commits to them in proof-of-work protected headers.
- **Transaction hashing and script** (Chapter 4, Chapter 6, Chapter 7). Payment channels and swaps depend on transaction hashes that exclude signature scripts, on `OP_CHECKSEQUENCEVERIFY` and relative lock times (DCP-0003, DCP-0004), on `OP_CHECKLOCKTIMEVERIFY`, and on `OP_SHA256` (DCP-0002).
- **Chain permanence.** Timestamps from `dcrtime`, Politeia's anchors and its ticket snapshot are only as durable as the blocks that contain them, which is why Politeia treats an anchor as final only once the anchoring transaction has 6 confirmations and takes its ticket snapshot `TicketMaturity` blocks below the tip. The proof-of-work and proof-of-stake rules that make deep reorganizations costly (Chapter 9, Chapter 10) underpin all of them.

PART V

Appendices

APPENDIX A

Opcode Reference

This appendix accounts for every one of the 256 possible opcode bytes of script version 0. It is a summary: the normative definitions are the rules of Chapter 6, referenced below, and where this table and those rules differ, the rules govern. The table describes the operation performed when an opcode is **executed**, that is, when it is reached by the program counter in an executing branch; Table A.1 describes how each class of opcode behaves when it is reached in an unexecuted branch and whether it counts toward the limit of 255 operations per script.

A.1 Conventions

Stack effects are written `[inputs] → [outputs]`, listing only the elements an opcode consumes and produces, bottom to top, with the top element last. In the arithmetic, bitwise and comparison opcodes, letters denote numeric operands, which must be minimally encoded script numbers of at most 4 bytes (at most 5 bytes for the lock-time operands, at most 1 byte for the signature type of `OP_CHECKSIGALT`); see Section 6.7.2. Booleans are pushed as `01` (true) and the empty string (false), and `ε` denotes the empty string. An opcode that needs more elements than the stack holds fails with `ErrInvalidStackOperation`. Flag values refer to Table 6.15.

The **processing** steps common to all opcodes (the checks for disabled and always-illegal opcodes, the operation count, the element size limit and the minimal-push rule) are given in Rule 6.6. Every executed push must be minimal (Rule 6.7).

Table A.1 Opcode classes

Class	When executed	When reached in an unexecuted branch	Counts as op
P push	Pushes its value; must be minimal.	No effect, but data longer than 2048 bytes still fails.	no
C conditional	Opens, toggles or closes a block.	Still processed: `OP_IF` / `OP_NOTIF` open a block without popping; `OP_ELSE` and `OP_ENDIF` act as specified.	yes
A active	Performs its operation.	No effect.	yes
G flag-gated	With its flag: as listed. Without: as class U.	No effect.	yes
N no-op	No effect, under all flags.	No effect.	yes
U upgradable no-op	No effect; fails only under the policy flag `0x01`.	No effect.	yes
R reserved	Fails (`ErrReservedOpcode`).	No effect.	yes, except `0x50`
I always illegal	Fails (`ErrReservedOpcode`).	Fails.	n/a (fails first)
D disabled	Fails (`ErrDisabledOpcode`).	Fails.	n/a (fails first)

The rule that decides whether an opcode counts toward the operation limit is purely numeric: every opcode whose byte value is greater than `0x60` counts, every other opcode does not. In addition, an executed `OP_CHECKMULTISIG(VERIFY)` adds its key count n.

A.2 Map of Byte Values

Table A.2 shows the class of every byte value at a glance. There are 96 push opcodes, 4 conditional opcodes, 70 active opcodes, 6 flag-gated opcodes, 7 no-operations, 61 upgradable no-operations, 9 reserved opcodes, 2 always illegal opcodes and 1 disabled opcode.

Table A.2 Class of every byte value (row: high nibble, column: low nibble)

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
1	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
2	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
3	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
4	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
5	R	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
6	P	N	R	C	C	I	I	C	C	A	A	A	A	A	A	A
7	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
8	A	A	A	A	A	A	A	A	A	A	A	A	A	N	N	A
9	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
a	A	A	A	A	A	A	A	A	A	A	A	D	A	A	A	A
b	U	G	G	U	U	U	U	U	U	U	N	N	N	N	A	A
c	G	G	G	G	U	U	U	U	U	U	U	U	U	U	U	U
d	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
e	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
f	U	U	U	U	U	U	U	U	U	R	R	R	R	R	R	R

A.3 Opcode Table

Ranges of consecutive bytes with identical behaviour are given in a single row; together the rows of Table A.3 cover all 256 values.

Table A.3 All opcode values

Byte	Name	Class	Stack effect	Semantics when executed
`0x00`	`OP_0`	P	`[] → [ε]`	Push the empty string (`OP_FALSE`).
`0x01` – `0x4b`	`OP_DATA_1` … `OP_DATA_75`	P	`[] → [d]`	Push the next n = byte value data bytes. Minimal unless the data is a single byte `01` – `10` or `81` .
`0x4c`	`OP_PUSHDATA1`	P	`[] → [d]`	1-byte length n follows, then n data bytes; push them. Minimal only for 76–255 bytes.
`0x4d`	`OP_PUSHDATA2`	P	`[] → [d]`	2-byte LE length n follows, then n bytes; push them. Minimal only for 256–65535 bytes.
`0x4e`	`OP_PUSHDATA4`	P	`[] → [d]`	4-byte LE length n follows, then n bytes. Never minimal: always fails when executed.
`0x4f`	`OP_1NEGATE`	P	`[] → [-1]`	Push `81` .
`0x50`	`OP_RESERVED`	R	—	Fails if executed (`ErrReservedOpcode`). Not counted as an operation. Counts as push-only.
`0x51` – `0x60`	`OP_1` … `OP_16`	P	`[] → [n]`	Push the number n = byte − `0x50` . `OP_1` is also `OP_TRUE` .
`0x61`	`OP_NOP`	N	`[] → []`	No operation; never discouraged.

Byte	Name	Class	Stack effect	Semantics when executed
0x62	OP_VER	R	—	Fails if executed (ErrReservedOpcode).
0x63	OP_IF	C	[x] → []	Pop x if executing; open a block, disabled if x is false (Rule 6.8).
0x64	OP_NOTIF	C	[x] → []	As OP_IF with the condition inverted.
0x65	OP_VERIF	I	—	Fails whenever reached, even unexecuted (ErrReservedOpcode).
0x66	OP_VERNOTIF	I	—	Fails whenever reached, even unexecuted (ErrReservedOpcode).
0x67	OP_ELSE	C	[] → []	Toggle execution of the innermost block; fails if no block is open.
0x68	OP_ENDIF	C	[] → []	Close the innermost block; fails if no block is open.
0x69	OP_VERIFY	A	[x] → []	Fail (ErrVerify) if x is false.
0x6a	OP_RETURN	A	—	Fail (ErrEarlyReturn).
0x6b	OP_TOALTSTACK	A	[x] → []	Move x to the alternate stack.
0x6c	OP_FROMALTSTACK	A	[] → [x]	Move the top of the alternate stack to the data stack; fails if empty.
0x6d	OP_2DROP	A	[a b] → []	Drop two.
0x6e	OP_2DUP	A	[a b] → [a b a b]	Duplicate two.
0x6f	OP_3DUP	A	[a b c] → [a b c a b c]	Duplicate three.
0x70	OP_2OVER	A	[a b c d] → [a b c d a b]	Copy the pair below the top pair.
0x71	OP_2ROT	A	[a b c d e f] → [c d e f a b]	Rotate three pairs.
0x72	OP_2SWAP	A	[a b c d] → [c d a b]	Swap two pairs.
0x73	OP_IFDUP	A	[x] → [x x] / [x]	Duplicate x if it is true.
0x74	OP_DEPTH	A	[] → [n]	Push the data stack depth (before the push).
0x75	OP_DROP	A	[x] → []	Drop.
0x76	OP_DUP	A	[x] → [x x]	Duplicate.
0x77	OP_NIP	A	[a b] → [b]	Remove the second element.
0x78	OP_OVER	A	[a b] → [a b a]	Copy the second element.
0x79	OP_PICK	A	[… n] → [… x_n]	Pop n (4 bytes); copy the element n below the top; $0 \le n <$ depth.
0x7a	OP_ROLL	A	[… n] → [… x_n]	Pop n (4 bytes); move the element n below the top to the top.
0x7b	OP_ROT	A	[a b c] → [b c a]	Rotate three.
0x7c	OP_SWAP	A	[a b] → [b a]	Swap.
0x7d	OP_TUCK	A	[a b] → [b a b]	Copy the top below the second.
0x7e	OP_CAT	A	[a b] → [a‖b]	Concatenate; result ≤ 2048 bytes (ErrElementTooBig).
0x7f	OP_SUBSTR	A	[s e b] → [s[b:e]]	Substring; start index on top; empty s gives empty; otherwise $0 \le b \le e \le \lvert s\rvert$ (Rule 6.14).
0x80	OP_LEFT	A	[s n] → [s[0:n]]	Left part; $0 \le n \le \lvert s\rvert$ unless s empty.
0x81	OP_RIGHT	A	[s n] → [s[n:]]	Right part; $0 \le n \le \lvert s\rvert$ unless s empty.
0x82	OP_SIZE	A	[x] → [x n]	Push $\lvert x\rvert$; x kept.
0x83	OP_INVERT	A	[a] → [~a]	Bitwise NOT of int32 a.
0x84	OP_AND	A	[a b] → [a&b]	Bitwise AND of int32 values.
0x85	OP_OR	A	[a b] → [a|b]	Bitwise OR of int32 values.

Byte	Name	Class	Stack effect	Semantics when executed
0x86	OP_XOR	A	[a b] → [a^b]	Bitwise XOR of int32 values.
0x87	OP_EQUAL	A	[a b] → [bool]	Byte-string equality.
0x88	OP_EQUALVERIFY	A	[a b] → []	OP_EQUAL then verify (ErrEqualVerify).
0x89	OP_ROTR	A	[v c] → [r]	Rotate uint32(v) right by $c \in [0, 31]$, result as int32 .
0x8a	OP_ROTL	A	[v c] → [r]	Rotate uint32(v) left by $c \in [0, 31]$, result as int32 .
0x8b	OP_1ADD	A	[a] → [a+1]	Increment (exact).
0x8c	OP_1SUB	A	[a] → [a-1]	Decrement (exact).
0x8d	OP_2MUL	N	[] → []	No operation (not a multiplication); never discouraged.
0x8e	OP_2DIV	N	[] → []	No operation (not a division); never discouraged.
0x8f	OP_NEGATE	A	[a] → [-a]	Negate.
0x90	OP_ABS	A	[a] → [|a|]	Absolute value.
0x91	OP_NOT	A	[a] → [a=0]	1 if $a = 0$ else 0 (numeric operand).
0x92	OP_0NOTEQUAL	A	[a] → [a≠0]	0 if $a = 0$ else 1.
0x93	OP_ADD	A	[a b] → [a+b]	Add (exact).
0x94	OP_SUB	A	[a b] → [a-b]	Subtract (exact).
0x95	OP_MUL	A	[a b] → [a·b]	Multiply, wrapping to int32 .
0x96	OP_DIV	A	[a b] → [a/b]	Truncating division; $b = 0$ fails (ErrDivideByZero).
0x97	OP_MOD	A	[a b] → [a%b]	Truncating remainder (sign of a); $b = 0$ fails.
0x98	OP_LSHIFT	A	[v c] → [v<<c]	int32 left shift, $c \in [0, 32]$; $c = 32$ gives 0.
0x99	OP_RSHIFT	A	[v c] → [v>>c]	int32 arithmetic right shift, $c \in [0, 32]$.
0x9a	OP_BOOLAND	A	[a b] → [bool]	1 if $a \neq 0$ and $b \neq 0$.
0x9b	OP_BOOLOR	A	[a b] → [bool]	1 if $a \neq 0$ or $b \neq 0$.
0x9c	OP_NUMEQUAL	A	[a b] → [bool]	1 if $a = b$.
0x9d	OP_NUMEQUALVERIFY	A	[a b] → []	OP_NUMEQUAL then verify (ErrNumEqualVerify).
0x9e	OP_NUMNOTEQUAL	A	[a b] → [bool]	1 if $a \neq b$.
0x9f	OP_LESSTHAN	A	[a b] → [bool]	1 if $a < b$.
0xa0	OP_GREATERTHAN	A	[a b] → [bool]	1 if $a > b$.
0xa1	OP_LESSTHANOREQUAL	A	[a b] → [bool]	1 if $a \leq b$.
0xa2	OP_GREATERTHANOREQUAL	A	[a b] → [bool]	1 if $a \geq b$.
0xa3	OP_MIN	A	[a b] → [min]	Smaller of a, b.
0xa4	OP_MAX	A	[a b] → [max]	Larger of a, b.
0xa5	OP_WITHIN	A	[x lo hi] → [bool]	1 if $\text{lo} \leq x < \text{hi}$.
0xa6	OP_RIPEMD160	A	[x] → [h]	RIPEMD-160.
0xa7	OP_SHA1	A	[x] → [h]	SHA-1.
0xa8	OP_BLAKE256	A	[x] → [h]	BLAKE-256.
0xa9	OP_HASH160	A	[x] → [h]	RIPEMD-160(BLAKE-256(x)).
0xaa	OP_HASH256	A	[x] → [h]	BLAKE-256(BLAKE-256(x)).
0xab	OP_CODESEPARATOR	D	—	Fails whenever reached, even unexecuted (ErrDisabledOpcode).
0xac	OP_CHECKSIG	A	[sig pk] → [bool]	ECDSA signature check (Rule 6.27).

Byte	Name	Class	Stack effect	Semantics when executed
0xad	OP_CHECKSIGVERIFY	A	[sig pk] → []	OP_CHECKSIG then verify (ErrCheckSigVerify).
0xae	OP_CHECKMULTISIG	A	[sigs m pks n] → [bool]	m-of-n ECDSA check, no dummy element; adds n to the operation count (Rule 6.28).
0xaf	OP_CHECKMULTISIGVERIFY	A	[sigs m pks n] → []	OP_CHECKMULTISIG then verify.
0xb0	OP_NOP1	U	[] → []	Upgradable no-operation.
0xb1	OP_CHECKLOCKTIMEVERIFY	G	[L] → [L]	Lock-time check with flag 0x02 (Rule 6.19); else upgradable no-op (OP_NOP2).
0xb2	OP_CHECKSEQUENCEVERIFY	G	[s] → [s]	Relative lock-time check with flag 0x04 (Rule 6.20); else upgradable no-op (OP_NOP3).
0xb3 – 0xb9	OP_NOP4 … OP_NOP10	U	[] → []	Upgradable no-operations.
0xba	OP_SSTX	N	[] → []	Stake tag; no operation. Recognized in P2SH detection; forbidden in redeem scripts.
0xbb	OP_SSGEN	N	[] → []	Stake tag; no operation. Recognized in P2SH detection; forbidden in redeem scripts.
0xbc	OP_SSRTX	N	[] → []	Stake tag; no operation. Recognized in P2SH detection; forbidden in redeem scripts.
0xbd	OP_SSTXCHANGE	N	[] → []	Stake tag; no operation. Recognized in P2SH detection; forbidden in redeem scripts.
0xbe	OP_CHECKSIGALT	A	[sig pk t] → [bool]	Alternative-suite check: t = 1 Ed25519, 2 Schnorr; 0 pushes false, others push true (Rule 6.29).
0xbf	OP_CHECKSIGALTVERIFY	A	[sig pk t] → []	OP_CHECKSIGALT then verify (ErrCheckSigAltVerify).
0xc0	OP_SHA256	G	[x] → [h]	SHA-256 with flag 0x20; else upgradable no-op (OP_UNKNOWN192).
0xc1	OP_TADD	G	[] → []	Treasury tag with flag 0x40 (no operation); else upgradable no-op (OP_UNKNOWN193).
0xc2	OP_TSPEND	G	[] → []	Treasury tag with flag 0x40; else OP_UNKNOWN194. One signature operation when treasury rules apply.
0xc3	OP_TGEN	G	[] → []	Treasury tag with flag 0x40; else upgradable no-op (OP_UNKNOWN195).
0xc4 – 0xf8	OP_UNKNOWN196 … OP_UNKNOWN248	U	[] → []	Unassigned; upgradable no-operations.
0xf9	OP_INVALID249	R	—	Fails if executed (ErrReservedOpcode).
0xfa	OP_SMALLINTEGER	R	—	Fails if executed (ErrReservedOpcode).
0xfb	OP_PUBKEYS	R	—	Fails if executed (ErrReservedOpcode).
0xfc	OP_UNKNOWN252	R	—	Fails if executed (ErrReservedOpcode).
0xfd	OP_PUBKEYHASH	R	—	Fails if executed (ErrReservedOpcode).
0xfe	OP_PUBKEY	R	—	Fails if executed (ErrReservedOpcode).
0xff	OP_INVALIDOPCODE	R	—	Fails if executed (ErrReservedOpcode).

A.4 Remarks

- **Names and aliases.** OP_FALSE is 0x00, OP_TRUE is 0x51, OP_NOP2 is 0xb1 and OP_NOP3 is 0xb2. When the corresponding flags are not set, 0xc0 – 0xc3 are called OP_UNKNOWN192 – OP_UNKNOWN195.

The names `OP_INVALID249`, `OP_SMALLINTEGER`, `OP_PUBKEYS`, `OP_UNKNOWN252`, `OP_PUBKEYHASH`, `OP_PUBKEY` and `OP_INVALIDOPCODE` (`0xf9` – `0xff`) are inherited labels with no function other than failing when executed.

- **Differences from Bitcoin.** `OP_CAT`, `OP_SUBSTR`, `OP_LEFT`, `OP_RIGHT`, `OP_INVERT`, `OP_AND`, `OP_OR`, `OP_XOR`, `OP_MUL`, `OP_DIV`, `OP_MOD`, `OP_LSHIFT` and `OP_RSHIFT` are enabled with the semantics of Section 6.8; `OP_2MUL` and `OP_2DIV` are no-operations; `OP_CODESEPARATOR` is disabled; `0x89` and `0x8a` are the rotations `OP_ROTR` and `OP_ROTL`; `OP_HASH160` and `OP_HASH256` use BLAKE-256; `OP_CHECKMULTISIG` takes no dummy element; and `0xa8`, `0xba` – `0xc3` are Decred-specific. In particular `0xa8`, which is `OP_SHA256` in Bitcoin, is `OP_BLAKE256` here, and Decred's `OP_SHA256` is `0xc0`, available only when its flag is set. `OP_INVERT`, `OP_AND`, `OP_OR` and `OP_XOR` act on 32-bit numbers, whereas Bitcoin's original versions acted on byte strings.
- **Parsing.** Every byte value parses as a one-byte opcode except `0x01` – `0x4e`, which carry inline data (Rule 6.2). A script must parse completely before any of it executes, so the bytes after an `OP_RETURN` must still form valid opcodes.

APPENDIX B

Index of Consensus Rules

Every numbered rule box in this document is listed below in order of appearance. Together they are the complete set of requirements a block must meet to be accepted by the reference implementation. A block is valid if and only if no rule rejects it. Where a rule matters in two contexts, for example a requirement on a stake transaction that is checked while validating a block, it is stated in both chapters and appears twice here; the two statements are equivalent.

3 Cryptographic Primitives 15 rules

3.1 BLAKE-256 13
3.2 Derived hash functions 16
3.3 BLAKE3 proof-of-work hash function 16
3.4 Parsing a secp256k1 public key 19
3.5 Strict public key encoding 19
3.6 ECDSA verification 21
3.7 Parsing a DER signature 22
3.8 Strict ECDSA signature encoding 22
3.9 Parsing an EC-Schnorr-DCRv0 signature 26
3.10 Parsing an EC-Schnorr-DCRv0 public key 26
3.11 EC-Schnorr-DCRv0 verification 26
3.12 Parsing an Ed25519 public key 28
3.13 Parsing an Ed25519 signature 28
3.14 Ed25519 verification 28
3.15 Merkle root 29

4 Data Encoding and Structures 11 rules

4.1 Decodability 33
4.2 Canonical variable-length integers 34
4.3 Serialization type 36
4.4 Outpoint tree byte 37
4.5 Transaction element counts 40
4.6 Witness count of a full serialization 40
4.7 Script lengths 40
4.8 Merkle roots before header commitments 45
4.9 Merkle roots after header commitments 45
4.10 Block transaction counts 46
4.11 Header size field 47

6 The Script System 36 rules

6.1 Non-zero script versions 72
6.2 Script parsing 72
6.3 Engine construction checks 74
6.4 Per-script end conditions 75
6.5 Combined stack size 75
6.6 Opcode processing 75
6.7 Minimal data pushes 76
6.8 Conditional opcodes 76
6.9 Final result 77
6.10 Boolean interpretation 79
6.11 Numeric operands 80
6.12 OP_PICK and OP_ROLL 82
6.13 OP_CAT 82
6.14 OP_SUBSTR 82
6.15 OP_LEFT and OP_RIGHT 82
6.16 OP_ROTR and OP_ROTL 83
6.17 OP_LSHIFT and OP_RSHIFT 84
6.18 OP_SHA256 85
6.19 OP_CHECKLOCKTIMEVERIFY 85
6.20 OP_CHECKSEQUENCEVERIFY 85
6.21 No-operation opcodes 86
6.22 Reserved and invalid opcodes 86
6.23 Hash type encoding 87
6.24 Strict DER signature encoding 87
6.25 Public key encoding 88
6.26 Subscript construction 88
6.27 OP_CHECKSIG 89
6.28 OP_CHECKMULTISIG 89
6.29 OP_CHECKSIGALT 90
6.30 SIGHASH_SINGLE with no matching output 92
6.31 Pay-to-script-hash evaluation 93
6.32 No stake opcodes in redeem scripts 94
6.33 Detecting tag opcodes in a script 95
6.34 Transaction signature operation count 96
6.35 Provably unspendable outputs 97
6.36 Strict null-data scripts 97

7 Transactions 43 rules

7.1 Outpoint tree must match the creating transaction 109
7.2 Outputs that never enter the UTXO set 109
7.3 Witness fields of spending inputs 110
7.4 Tree placement of transaction types 112
7.5 Inputs and outputs present 112
7.6 Maximum serialized size 112
7.7 Output amounts 113
7.8 Input witness amounts 113
7.9 No duplicate inputs 113
7.10 Maximum transaction version 114
7.11 Maximum output script version of regular transactions 114
7.12 No stake opcodes in regular outputs 114
7.13 Coinbase input 115
7.14 Vote stakebase and ticket input 115
7.15 Revocation version and input 115
7.16 Treasurybase input 115
7.17 Treasury spend input 116
7.18 Treasury add change 116
7.19 Non-null inputs 116
7.20 Expired transactions 117
7.21 Lock time finality 117
7.22 Transactions subject to sequence locks 119
7.23 Treasury spend sequence number 119
7.24 Sequence lock enforcement 120
7.25 Inputs exempt from UTXO lookup 121
7.26 Referenced output exists and is unspent 121
7.27 Stake transactions do not spend same-block outputs 122
7.28 Zero-value outputs are not spendable 122
7.29 Coinbase maturity 122
7.30 Maturity of outputs of expiring transactions 122
7.31 Treasury generation output maturity 122
7.32 Ticket outputs are spent only by votes and revocations 122
7.33 Treasury add outputs are unspendable 122
7.34 Vote and revocation output maturity 123
7.35 Stake change maturity 123
7.36 Referenced amount range 123
7.37 Input total 124
7.38 Outputs do not exceed inputs; fee 124
7.39 Connection order and output availability 126
7.40 Historical same-block stake spend exceptions 127
7.41 Inputs whose scripts are executed 128
7.42 Script execution parameters 129
7.43 Script failure invalidates the block 129

8 The Stake System 42 rules

8.1 Stake transaction type 135
8.2 Tagged script 135
8.3 Null data script 135
8.4 Ticket input script forms 137
8.5 Ticket commitment amounts 137
8.6 Ticket price 138
8.7 Treasury votes require the treasury agenda 140
8.8 Stakebase signature script 140
8.9 Ticket input not null 140
8.10 Revocation version 142
8.11 Revocation script validation 142
8.12 Ticket outpoint index 143
8.13 Ticket output available 143
8.14 Ticket submission output 143
8.15 Ticket spend maturity 143
8.16 Number of payments 143
8.17 Payment script 145
8.18 Payment value and fee limits 145
8.19 Stakebase amount 146
8.20 Stakebase totals 146
8.21 Ticket outputs are spent only by votes and revocations 147
8.22 Maturity of stake outputs 147
8.23 Stake transactions do not spend same-block outputs 148
8.24 Live pool order 150
8.25 Stake state transition 153
8.26 Tickets expiring in block h 154
8.27 Stake header fields below stake validation height 155
8.28 Minimum number of votes 155
8.29 Maximum number of votes 155
8.30 Maximum number of ticket purchases 155
8.31 Ticket purchase count 156
8.32 Pool size commitment 156
8.33 Final state commitment 156
8.34 Tree membership 156
8.35 Allowed stake transaction types 156
8.36 No votes or revocations below stake validation height 156
8.37 Non-empty stake tree 156
8.38 Vote count 157
8.39 Revocation count and limit 157
8.40 Votes vote on the parent 157
8.41 Parent approval 157
8.42 Votes and revocations spend eligible tickets 158

9 Proof of Work and Difficulty 6 rules

9.1 Target range 164
9.2 Proof-of-work hash 164
9.3 Difficulty bits 168
9.4 Testnet3 maximum-difficulty checkpoint 175
9.5 Testnet3 minimum block spacing at maximum difficulty 176
9.6 Stake difficulty 176

10 Blocks and Chain Selection 52 rules

10.1 Validity is inherited 183
10.2 Proof of work, context-free form 184
10.3 Timestamp precision 185
10.4 Timestamp not too far in the future 185
10.5 Stake header fields in the sanity stage 185
10.6 Parent known and not known invalid 186
10.7 Height 186
10.8 Timestamp after the median time past 186
10.9 Minimum block spacing at maximum difficulty (testnet3 only) 186
10.10 Difficulty, positional form 186
10.11 Block version 187
10.12 Fork rejection checkpoint 187
10.13 Maximum-difficulty checkpoint (testnet3 only) 187
10.14 Proof of work, contextual form 188
10.15 Difficulty and ticket price, exact 188
10.16 Stake version 189
10.17 Ticket pool commitments 189
10.18 Wire size limit 189
10.19 Merkle roots 189
10.20 At least one regular transaction 190
10.21 Header size field 190
10.22 Transaction sanity 190
10.23 Ticket purchases 190
10.24 No duplicate transactions 190
10.25 No expired transactions 191
10.26 Coinbase is the first regular transaction 191
10.27 Coinbase height commitment 191
10.28 Treasurybase placement 191
10.29 Transaction context checks 191
10.30 Tree composition 191
10.31 Votes and revocations in the block 192
10.32 Treasury transaction limits 192
10.33 Approval of the parent 192
10.34 Signature operations, fast count 192
10.35 Consensus block size 192
10.36 All transactions final 192
10.37 Vote and revocation eligibility 192
10.38 Header commitment root 194
10.39 Effect of disapproval 195
10.40 Signature operations, per tree 200
10.41 Transaction inputs 200
10.42 Stake transactions do not spend outputs of their own block 201
10.43 Regular transactions spend only earlier outputs of their own block 201
10.44 Each output is spent at most once per block 201
10.45 Historical same-block stake spends 201
10.46 Coinbase amount 202
10.47 Vote subsidies and treasurybase amount 202
10.48 Script validation 203
10.49 Relative lock times 203
10.50 Best chain 204
10.51 Recording invalidity 207
10.52 Assume-valid ancestors 208

11 Issuance and Block Subsidy 14 rules

11.1 Full block subsidy 214
11.2 Selection of the split variant 215
11.3 The vote subsidy is based on the height voted on 216
11.4 The first regular transaction is the only coinbase 219
11.5 Coinbase input 219
11.6 Treasury output before the treasury agenda 220
11.7 Coinbase height commitment 220
11.8 Coinbase input value 222
11.9 Coinbase output total 222
11.10 Vote stakebase value 223
11.11 Vote output limits 223
11.12 Treasurybase amount 224
11.13 Block-one coinbase 224
11.14 Block-one outputs 224

12 The Decentralized Treasury 30 rules

12.1 Treasury transactions only in the stake tree 232
12.2 No treasury transactions before activation 232
12.3 Permitted stake transactions before stake validation height 232
12.4 Coinbase detection with the treasury active 232
12.5 Treasury balance 233
12.6 Treasurybase required and unique 235
12.7 Treasurybase form 235
12.8 Treasurybase height commitment 235
12.9 Treasurybase amount 236
12.10 Treasurybase creates no outputs 236
12.11 Treasury add change 237
12.12 Treasury add inputs 237
12.13 Treasury add limit 237
12.14 Treasury add outputs are unspendable 237
12.15 Treasury spend input 239
12.16 Treasury spend key 239
12.17 Treasury spend signature 239
12.18 Treasury spend amounts 239
12.19 Treasury spend sequence number 240
12.20 Treasury spends only on a TVI 240
12.21 Full voting window after stake validation height 240
12.22 Treasury spend inside its window 240
12.23 Treasury spend not previously mined 240
12.24 Treasury spend approved 241
12.25 Treasury expenditure 241
12.26 Maturity of treasury payouts 241
12.27 Treasury spend expiry and window 242
12.28 Treasury vote output 244
12.29 Signature operations of treasury transactions 250
12.30 No treasury opcodes in regular outputs 250

13 Consensus Rule Changes 6 rules

13.1 Obsolete block versions 256
13.2 Vote bits and vote version 257
13.3 Well-formed deployments 258
13.4 Header stake version 260
13.5 Threshold state 263
13.6 Agenda activity for a block 265

14 Network Parameters 2 rules

14.1 Genesis block 286
14.2 Genesis outputs are not spendable 286

16 Compact Filters and Light Clients 3 rules

16.1 Version 2 block filter 327
16.2 Spent scripts 329
16.3 Header commitment root 331

APPENDIX C

Differences from Bitcoin

Decred's code base descends from an implementation of Bitcoin, and much of its structure will be familiar to anyone who knows that protocol. The familiarity is a hazard: many details differ, and an implementation that assumes Bitcoin's behavior where this document is silent will be wrong. This appendix is informative. It lists the differences that most often mislead, with a pointer to the chapter that specifies each one. It is a guide to the document, not a substitute for it.

C.1 Consensus Model

Table C.1 Consensus and governance

Aspect	Bitcoin	Decred
Block validity	Proof of work alone.	Proof of work, and votes from a majority of the five tickets drawn for the block (Chapter 8).
Finality of a block's transactions	Fixed once the block is mined.	The regular transaction tree of a block takes effect only if the votes in the next block approve it (Chapter 10).
Rule changes	Activated by miner signaling or by a date or height chosen by developers.	Decided by a stakeholder vote with a 75% threshold and a quorum, evaluated by a state machine in the consensus rules (Chapter 13).
Development funding	None in the protocol.	10% of every block reward accrues to a treasury spent only by stakeholder vote (Chapter 12).
Target block interval	10 minutes.	5 minutes on the main network (Chapter 14).
Issuance	Halves every 210000 blocks.	Multiplied by 100/101 every 6144 blocks, and divided between miners, voters and the treasury (Chapter 11).
Proof-of-work function	Double SHA-256 of the 80-byte header.	BLAKE-256 of the 180-byte header originally; BLAKE3 since DCP-0011, computed separately from the block hash (Chapter 9).
Difficulty adjustment	Every 2016 blocks, from the time taken by the previous 2016.	Originally every 144 blocks from a weighted average over 20 windows; since DCP-0011 recomputed for every block by an exponential formula (Chapter 9).

C.2 Hashes and Identifiers

Table C.2 Hash functions and identifiers

Aspect	Bitcoin	Decred
General-purpose hash	Double SHA-256.	A single BLAKE-256 with 14 rounds (Chapter 3).
Block identifier	Double SHA-256 of the header; also the proof-of-work hash.	BLAKE-256 of the header. Since DCP-0011 it is distinct from the proof-of-work hash.

Aspect	Bitcoin	Decred
Transaction identifier	Hash of the transaction without witness data (since segregated witness).	BLAKE-256 of the prefix serialization, which never included signature scripts (Chapter 4).
Merkle tree leaves	Transaction identifiers, with a separate witness commitment in the coinbase.	Full transaction hashes, which commit to prefix and witness together (Chapter 4).
`HASH160`	RIPEMD-160 of SHA-256.	RIPEMD-160 of BLAKE-256.
`OP_HASH256`	Double SHA-256.	Double BLAKE-256. SHA-256 is available through `OP_SHA256` (Chapter 6).
Address checksum	First four bytes of double SHA-256.	First four bytes of double BLAKE-256, over a two-byte version prefix (Chapter 5).
P2P message checksum	First four bytes of double SHA-256 (in the original, unencrypted transport).	First four bytes of a single BLAKE-256 (Chapter 15).

C.3 Transactions and Blocks

Table C.3 Structures

Aspect	Bitcoin	Decred
Block header	80 bytes, six fields.	180 bytes, eighteen fields, including the vote tally, ticket counts, ticket pool size, ticket price and lottery state (Chapter 4).
Transactions in a block	One list.	Two lists: the regular tree and the stake tree (Chapter 4, Chapter 10).
Transaction version field	32-bit version.	16-bit version and 16-bit serialization type.
Outpoint	Transaction hash and output index.	Transaction hash, output index and a tree byte.
Input witness	Signature script, or a witness stack.	Signature script together with the amount, block height and block index of the output being spent, which are checked against the ledger (Chapter 7).
Output	Amount and script.	Amount, 16-bit script version and script.
Expiry	None.	An optional height from which the transaction is no longer valid.
Maximum block size	A weight limit of four million units.	393216 bytes of serialized block on the main network (Chapter 10).
Coinbase maturity	100 blocks.	256 blocks on the main network.
Duplicate transaction check	Enforced (BIP 30). The check is skipped for most blocks since the coinbase began to carry the height (BIP 34), which makes duplicates impractical but not impossible.	Unnecessary from the start: every coinbase from height 2 carries the block height (Chapter 10, Chapter 11).

C.4 Script

The script language has the same shape as Bitcoin's, a stack machine with no loops, but differs in limits, in the set of enabled opcodes and in every detail of signature checking. Chapter 6 is the specification and Appendix A lists every opcode.

Table C.4 Script

Aspect	Bitcoin	Decred
Maximum script size	10000 bytes.	16384 bytes.
Maximum pushed element	520 bytes.	2048 bytes.
Maximum counted opcodes	201.	255.

Aspect	Bitcoin	Decred
Maximum stack depth	1000.	1024.
`OP_CHECKMULTISIG`	Pops one extra unused element.	Pops no extra element.
`OP_CODESEPARATOR`	Affects the signed script.	Disabled.
Splice, bitwise and arithmetic opcodes disabled in Bitcoin	Disabled: a script containing them is invalid.	Most are enabled, with 32-bit semantics defined in Chapter 6; rotation opcodes are added.
Signature schemes	ECDSA, and BIP 340 Schnorr in Taproot outputs.	ECDSA through `OP_CHECKSIG`; Ed25519 and a secp256k1 Schnorr scheme of Decred's own definition through `OP_CHECKSIGALT` (Chapter 3).
Signature and key encoding	Strict DER is consensus (BIP 66); low-S signatures and the rejection of hybrid public keys are relay policy for legacy scripts.	Strict DER, low S and strict public key encoding (compressed or uncompressed only) are all consensus: a violation fails the script (Chapter 6).
Hash opcodes	`0xa8` is `OP_SHA256`; `OP_HASH160` and `OP_HASH256` are built on SHA-256.	`0xa8` is `OP_BLAKE256`; `OP_HASH160` and `OP_HASH256` are built on BLAKE-256; `OP_SHA256` is `0xc0`, specified by DCP-0002 and enabled by the `lnfeatures` vote (Chapter 6).
Bitwise opcodes	`OP_INVERT`, `OP_AND`, `OP_OR` and `OP_XOR` operated on byte strings before they were disabled.	They operate on 32-bit numbers.
Signed messages	Double SHA-256 of the prefixed message.	A single BLAKE-256 of the prefixed message (Chapter 5); not consensus.
Signature hash	Several algorithms depending on output type.	One algorithm, which hashes a prefix digest and a witness digest (Chapter 6).
Stake and treasury opcodes	None.	Tag opcodes mark outputs that only tickets, votes, revocations and treasury transactions may create or spend (Chapter 8, Chapter 12).
Script upgrades	New witness versions and Tapscript leaf versions.	A script version on every output. Versions above those enabled by vote are rejected (Chapter 13).
Lock-time features	BIP 65, BIP 68, BIP 112 and BIP 113.	The same concepts with Decred-specific encodings and rules (Chapter 7). `OP_CHECKLOCKTIMEVERIFY` (BIP 65) has been enforced since launch; the rest were activated by DCP-0003.

C.5 Network and Wallets

Table C.5 Network and wallet conventions

Aspect	Bitcoin	Decred
Light clients	Compact block filters (BIP 157 and BIP 158) that headers do not commit to.	Version 2 filters with different parameters and contents, committed to by every header since DCP-0005 (Chapter 16).
Unknown P2P messages	Ignored.	Treated as misbehavior by the reference node (Chapter 15).
Hierarchical deterministic keys	BIP 32.	The same construction with different version bytes and a small number of differences in edge cases (Chapter 5).
Coin mixing	Not part of the node.	A peer-to-peer mixing protocol is carried by the node's P2P network (Chapter 18).

References

[1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] I. Bentov, C. Lee, A. Mizrahi, and M. Rosenfeld, “Proof of Activity: Extending Bitcoin's Proof of Work via Proof of Stake,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 3, pp. 34–37, 2014.

[3] A. Mackenzie, “MEMCOIN2: A Hybrid Proof-of-Work, Proof-of-Stake Crypto-currency,” 2013. [Online]. Available: https://decred.org/research/mackenzie2013.pdf

[4] C. Jepson, “DTB001: Decred Technical Brief,” 2015. [Online]. Available: https://decred.org/dtb001.pdf

[5] S. Bradner, “Key words for use in RFCs to Indicate Requirement Levels, RFC 2119,” 1997. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2119

[6] Decred developers, *dcrd: Decred daemon, the reference full-node implementation.* (2026). [Online]. Available: https://github.com/decred/dcrd

[7] D. Collins, “DCP-0000: DCP Process.” [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0000/dcp-0000.mediawiki

[8] H. Krawczyk, M. Bellare, and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication, RFC 2104,” 1997. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2104

[9] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, “SHA-3 proposal BLAKE, version 1.3,” 2010. [Online]. Available: https://www.aumasson.jp/blake/blake.pdf

[10] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn, “BLAKE3: one function, fast everywhere,” 2020. [Online]. Available: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf

[11] National Institute of Standards and Technology, “Secure Hash Standard (SHS), FIPS PUB 180-4,” 2015. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.180-4

[12] H. Dobbertin, A. Bosselaers, and B. Preneel, “RIPEMD-160: A Strengthened Version of RIPEMD,” in *Fast Software Encryption (FSE 1996)*, Springer, 1996.

[13] Certicom Research, “SEC 2: Recommended Elliptic Curve Domain Parameters, version 2.0,” 2010. [Online]. Available: https://www.secg.org/sec2-v2.pdf

[14] Certicom Research, “SEC 1: Elliptic Curve Cryptography, version 2.0,” 2009. [Online]. Available: https://www.secg.org/sec1-v2.pdf

[15] International Telecommunication Union, “Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), Recommendation ITU-T X.690,” 2021. [Online]. Available: https://www.itu.int/rec/T-REC-X.690

[16] T. Pornin, “Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA), RFC 6979,” 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6979

[17] C.-P. Schnorr, “Efficient Signature Generation by Smart Cards,” *Journal of Cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[18] P. Wuille, J. Nick, and T. Ruffing, “BIP 340: Schnorr Signatures for secp256k1.” [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki

[19] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[20] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA), RFC 8032," 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8032

[21] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology (CRYPTO '87)*, Springer, 1987.

[22] Decred developers, *dcrwallet: a secure Decred wallet daemon.* (2026). [Online]. Available: https://github.com/decred/dcrwallet

[23] Bitcoin Wiki contributors, "Base58Check encoding." [Online]. Available: https://en.bitcoin.it/wiki/Base58Check_encoding

[24] P. Wuille, "BIP 32: Hierarchical Deterministic Wallets." [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

[25] M. Palatinus and P. Rusnak, "BIP 43: Purpose Field for Deterministic Wallets." [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki

[26] M. Palatinus and P. Rusnak, "BIP 44: Multi-Account Hierarchy for Deterministic Wallets." [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

[27] P. Rusnak and M. Palatinus, "SLIP-0044: Registered coin types for BIP-0044." [Online]. Available: https://github.com/satoshilabs/slips/blob/master/slip-0044.md

[28] P. Wuille and G. Maxwell, "BIP 173: Base32 address format for native v0-16 witness outputs." [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki

[29] D. Collins, "DCP-0002: SHA256 Opcode." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0002/dcp-0002.mediawiki

[30] D. Collins, "DCP-0003: Relative Lock Time Support." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0003/dcp-0003.mediawiki

[31] G. Andresen, "BIP 16: Pay to Script Hash." [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki

[32] D. Collins, "DCP-0004: Fix LN Sequence Locks View." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0004/dcp-0004.mediawiki

[33] R. Staudt, "DCP-0009: Automatic Ticket Revocations." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0009/dcp-0009.mediawiki

[34] D. Collins, "DCP-0011: Change PoW to BLAKE3 and ASERT." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0011/dcp-0011.mediawiki

[35] D. Collins, "DCP-0001: New Stake Difficulty Algorithm." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0001/dcp-0001.mediawiki

[36] freetrader, J. Toomim, C. Culianu, and M. Lundeberg, "ASERT Difficulty Adjustment Algorithm (aserti3-2d)." [Online]. Available: https://upgradespecs.bitcoincashnode.org/2020-11-15-asert/

[37] Decred contributors, "Decred Documentation." [Online]. Available: https://docs.decred.org/

[38] D. Collins, "DCP-0010: Change PoW/PoS Subsidy Split To 10/80." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0010/dcp-0010.mediawiki

[39] D. Collins, "DCP-0012: Change PoW/PoS Subsidy Split To 1/89." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0012/dcp-0012.mediawiki

[40] M. Peereboom, "DCP-0006: Decentralized Treasury." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0006/dcp-0006.mediawiki

[41] M. Degiovani, "DCP-0007: Revert Treasury Expenditure Policy." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0007/dcp-0007.mediawiki

[42] D. Collins, "DCP-0013: New Max Treasury Expenditure Policy." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0013/dcp-0013.mediawiki

[43] P. Wuille, P. Todd, G. Maxwell, and R. Russell, "BIP 9: Version bits with timeout and delay." [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0009.mediawiki

[44] D. Collins, "DCP-0008: Explicit Version Upgrades." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0008/dcp-0008.mediawiki

[45] D. Collins, "DCP-0005: Block Header Commitments." [Online]. Available: https://github.com/decred/dcps/blob/master/dcp-0005/dcp-0005.mediawiki

[46] O. Osuntokun and A. Akselrod, "BIP 158: Compact Block Filters for Light Clients." [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0158.mediawiki

[47] O. Osuntokun, A. Akselrod, and J. Posen, "BIP 157: Client Side Block Filtering." [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki

[48] J.-P. Aumasson and D. J. Bernstein, "SipHash: A Fast Short-Input PRF," in *Progress in Cryptology (INDOCRYPT 2012)*, Springer, 2012.

[49] T. Ruffing, P. Moreno-Sanchez, and A. Kate, "P2P Mixing and Unlinkable Bitcoin Transactions," in *Network and Distributed System Security Symposium (NDSS 2017)*, 2017.

[50] DiceMix contributors, "DiceMix Light." [Online]. Available: https://github.com/BlockstreamResearch/dicemix/blob/master/doc/protocol.md

[51] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal, "NTRU Prime: Reducing Attack Surface at Low Cost," in *Selected Areas in Cryptography (SAC 2017)*, Springer, 2017.

[52] D. J. Bernstein, "ChaCha, a variant of Salsa20," 2008. [Online]. Available: https://cr.yp.to/chacha/chacha-20080128.pdf

[53] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols, RFC 8439," 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8439

[54] D. Chaum, "The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability," *Journal of Cryptology*, vol. 1, no. 1, pp. 65–75, 1988.

[55] Decred developers, *Politeia: a system for storing off-chain data that is both versioned and timestamped.* (2026). [Online]. Available: https://github.com/decred/politeia

[56] Decred developers, *vspd: a voting service provider for Decred.* (2026). [Online]. Available: https://github.com/decred/vspd

[57] Decred developers, *DCRDEX: the Decred decentralized exchange.* (2026). [Online]. Available: https://github.com/decred/dcrdex

[58] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments," 2016. [Online]. Available: https://lightning.network/lightning-network-paper.pdf

[59] Decred developers, *dcrlnd: Decred Lightning Network daemon.* (2026). [Online]. Available: https://github.com/decred/dcrlnd

[60] Lightning Network developers, "BOLT: Basis of Lightning Technology (Lightning Network specifications)." [Online]. Available: https://github.com/lightning/bolts

[61] Decred developers, *atomicswap: on-chain atomic swaps for Decred and other cryptocurrencies.* (2026). [Online]. Available: https://github.com/decred/atomicswap

[62] Decred developers, *dcrtime: a timestamping service anchored to the Decred blockchain.* (2026). [Online]. Available: https://github.com/decred/dcrtime