

Contents

LanternSeaCoin (LSC) — Reference Paper 2.0	1
How to read this document	1
0. Overview	2
0.1 Background and Related Work	3
0.2 System and Threat Model	4
1. Network Topology: LSC Nodes	5
2. Block Structure and the Segmented Data Model	11
3. Dual-Tier Pass-Through Operations	12
4. Blink Subsystem: Timing-Based Topology Detection	14
5. Coins, Movability, and Offline-Redeemable Stamps	14
6. The Chain-Native Clock: Bootstrap Phase	17
7. Time Threading and the UTC-Anchored OTP Chain	17
8. Offline Client Counting and Resync	18
9. Clock-Correction Governance	19
10. Open Trade-offs (Consolidated)	21
11. Summary	22
12. Mining Segments: Auxiliary Proof-of-Work	22
13. Dynamic Segment Registry (Development-Time Extensibility)	28
14. Concrete Parameters and Development Specification	30
15. Security Considerations	36
16. Automatic Checkpointing and Chain Archival	39
17. Network, Synchronization, Persistence, Node Process, RPC	42
18. Redemption: Offline, Online, Snapshots	43
19. Terminal and Air-Gap Layer	45
20. Banknote-Inspired Security Layers	47
21. Sovereignty and Dependencies	51
Annex A: Consolidated Parameters	52
Annex B: Formulas and Numerical Tables	53
References	54

LanternSeaCoin (LSC) — Reference Paper 2.0

A Stable-Node Ledger with Segmented Pass-Through Data, Offline-Redeemable Stamps, Auxiliary Proof-of-Work, a Chain-Native Clock, and Banknote-Inspired Security Layers

Version 2.0 · 5 October 2026 · Reference specification

Status: Conceptual / speculative design document. Not an existing, deployed, or audited cryptocurrency. Not investment or financial advice. This is an engineering specification, grounded in published cryptocurrency research and real-world precedent where noted.

How to read this document

This is the single, complete reference for LanternSeaCoin (LSC). It merges the original concept paper (§0–§16), the concrete protocol rules established during implementation, the redemption protocol, the terminal and air-gap layer, and the banknote-inspired security layers. Section numbers §0–§16 are preserved from earlier versions because source code and comments cite them; §17–§21 and the annexes are new.

- **Normative language.** MUST, MUST NOT, SHOULD, SHOULD NOT and MAY are used as in RFC 2119.

- **Notation.** H denotes BLAKE2b-256 unless stated otherwise. Formulas are given in TeX notation inside code blocks. Strings are byte buffers.
- **Parameters.** Binding values are listed in §14 and consolidated in Annex A.

§	Topic
0	Overview, related work, system and threat model
1	Network topology, slot scheduling, fork choice, reorganization and its stabilization
2	Block structure, hashing, segments
3	Dual-tier pass-through operations
4	Blink subsystem
5	Coins, stamps, maturity, transactions, conservation of supply
6–8	Bootstrap, time threading, OTP chain, offline resync
9	Governance, key rotation and revocation
10–11	Open trade-offs, summary
12	Mining segments, proof weights, reward distribution
13	Segment registry, series model, activation, ParameterRegistry
14	Concrete parameters and development specification
15	Security considerations and layered attack analysis
16	Checkpointing, finality, archival
17	Network, synchronization, persistence, node process, RPC
18	Redemption: offline, online, snapshots
19	Terminal and air-gap layer
20	Banknote-inspired security layers
21	Sovereignty and dependencies
Annex A	Consolidated parameters
Annex B	Formulas and numerical tables

0. Overview

LanternSeaCoin (LSC) is a conceptual ledger design built around nine core ideas:

1. A small, fixed, **stable set of validator nodes** ("LSC nodes") instead of a large open miner/staking pool.
2. Block data stored as many small **segments**, each passed through one of two fixed transformation tiers — no compression, no asymmetric keys, no reconstruction algorithm.
3. A **blink subsystem** reserving part of each block for timing-based network-topology probes.
4. **Stamps** — non-transferable claim records baked directly into a coin's own definition, enabling offline redemption, alongside ordinary movable coins.
5. A **chain-native clock**: no node or client clock is trusted; time is bootstrapped once from 10 self-reporting nodes and then threaded forward purely from chain structure, with any future change to the clock formula requiring unanimous node approval.
6. **Verifiable slot scheduling with deterministic fork choice and quorum finality**: a VRF decides who may produce a block, a total order over branches decides the chain, and a 16-of-24 checkpoint quorum makes blocks final (§1, §16).

7. **Auxiliary proof-of-work as anti-spam and reward basis:** miners submit small, cheaply verifiable work proofs that distribute the block reward in proportion to proven work; mined rewards are time-locked (stamped) until they mature (§12, §5).
8. **Banknote-inspired security layers and a terminal and air-gap layer:** 14 independent features on three verification tiers (§20), and a layer that keeps card data inside certified terminals and hands only metadata to an air-gapped device (§19).
9. **Versioned series:** new features are introduced by soft fork through new segment types, activation heights and a ParameterRegistry (§13).

There is deliberately **no privacy layer** (no ring signatures, stealth addresses, or hidden amounts) — transactions are plainly attributable on-chain. Section 0.1 explains why, against what CryptoNote/Monero actually do.

0.1 Background and Related Work

This design is not built in a vacuum. Two existing projects directly informed — and were directly diverged from — specific mechanisms below.

CryptoNote (Monero's underlying protocol)

CryptoNote, authored by the pseudonymous Nicolas van Saberhagen, is the protocol Monero and several other coins are built on. Its core privacy mechanism is the **one-time ring signature**: a protocol based on one-time ring signatures that gives users unconditional unlinkability, addressing the fact that ordinary signatures allow transactions to be traced to their senders and receivers. Structurally, this works by having a transaction input identify a set of possible outputs — the real one plus several decoy “mixin” outputs — with a ring signature proving the input is valid for one of them without revealing which.

Double-spend prevention in CryptoNote does **not** rely on revealing the sender. Instead, the protocol derives a “key image” from the spender's private key via a one-way function, and every node maintains the list of used key images, rejecting any transaction whose key image has already appeared — all without ever learning the private key itself. This is the specific mechanism CryptoNote researchers call a **traceable ring signature** — a modification that prevents double spends while still hiding the origin and destination of a transaction, built on prior academic work on traceable ring signatures by Fujisaki and Suzuki that CryptoNote's own author explicitly credits as the basis for the design, weakening full traceability while keeping just enough linkability to guarantee a key is used only once.

What LanternSeaCoin borrows from this: the *shape* of “prove non-reuse without revealing identity” — not the ring-signature cryptography itself, which LanternSeaCoin explicitly does not implement (§0 — no privacy layer). Instead, the OTP-chain / stamp `claim_proof` mechanism (§7, §5) borrows the same structural idea CryptoNote's key image uses: a value derived from something private (here, a stamp's mint-time chain position) that any node can check for prior use without needing to unmask who is making the claim. LanternSeaCoin's version is deliberately **not** cryptographically unlinkable — sender, receiver, and stamp ownership are plainly visible on-chain (§0) — only the double-spend-style *non-reuse check* is structurally similar, not the anonymity property.

What is explicitly rejected: ring signatures, one-time stealth addresses, and any mixin/decoy-output scheme. CryptoNote's whole point is unlinkability; LanternSeaCoin's whole point (by design) is a plain, publicly attributable ledger. Borrowing the double-spend-check *shape* while rejecting the privacy property is a deliberate, named split, not an oversight.

Feathercoin

Feathercoin is a long-running Bitcoin/Litecoin-derived altcoin, launched in April 2013, positioned around being lightweight and fast. Concretely:

- **Block cadence:** Feathercoin targets roughly one-minute block times, meaning transaction settlement is also about one minute, and its own mining-pool data confirms an actual observed block time close to that target. The exact target is 1.0 minute, with difficulty retargeted every block using 15-, 120-, and 480-block rolling averages and 25% damping to keep it stable.
- **Proof-of-work algorithm:** Feathercoin originally used Scrypt and later hard-forked to **Neo-Scrypt**, a memory-hard proof-of-work algorithm that can only be practically mined with graphics cards rather than specialized ASICs, intended to keep mining accessible to ordinary enthusiasts and thereby more decentralized.
- **Supply and issuance:** Feathercoin has a fixed maximum supply of 336 million coins, with the block reward halving every 2,100,000 blocks.
- **Chain-integrity mechanism:** Feathercoin implements what it calls **Automatic Checkpointing (ACP)** — a mechanism that creates a new checkpoint 5 blocks deep from the chain tip, specifically to protect the blockchain's history against 51%-style reorganization attacks.

What LanternSeaCoin borrows from this: the general Feathercoin ethos of “lightweight and fast rather than maximally decentralized” fits directly with LanternSeaCoin's own small stable-node design (§1) — both prioritize practical throughput/finality over an unbounded open miner set. The ~60-second block target used throughout this paper (§1) mirrors Feathercoin's real-world ~1-minute cadence almost exactly, chosen for the same reason: short enough for fast settlement, long enough to be practical for a small, known set of block producers to keep up with. Feathercoin's **Automatic Checkpointing** concept is the direct precedent for a possible LanternSeaCoin chain-integrity checkpoint (realized in §16) — periodically locking in recent history to blunt any attempt to rewrite it, which matters even more for a small-node design where a colluding subset is a more realistic threat than in a large open miner pool.

What is explicitly rejected: Feathercoin's actual proof-of-work mining model (NeoScrypt, GPU mining, open participation via hash power) is not used here as a block-production model — LanternSeaCoin does not select block producers by a hash race; block production rotates among the fixed LSC-node set (§1) by VRF schedule. Proof-of-work appears only as an auxiliary anti-spam and reward-distribution mechanism (§12). The borrowing is at the level of *design philosophy and cadence*, not shared consensus mechanics.

Lesotho Loti banknote security architecture

A description of the Loti banknotes provided by the author lists successive generations of security features (watermark, security thread, intaglio print, UV elements, microtext, latent images, optically variable ink, hologram, tactile marks, anti-copy and anti-IR features, micro-guilloche, protective coating) and separates the technical security of the notes from the lack of monetary sovereignty (peg to the rand, Common Monetary Area). LSC uses this as a design analogy: §20 maps 14 features to cryptographic constructions, §13.2 adopts the generation model, and §21 adopts the separation of instrument security and rule sovereignty. Historical details of the description were not independently verified, and no argument of this paper depends on them.

0.2 System and Threat Model

LSC consists of a fixed cohort of $N = 24$ scheduling nodes, any number of miners, clients and terminals, and assumes at most $f = 7$ Byzantine nodes.

0.2.1 Actors

Actor	Keys	Role
Scheduling node	Ed25519 and ECVRF	slot lottery, block production, checkpoint signature, revocation vote
Miner	address (Bech32)	submits work proofs, receives reward share
Wallet client	BIP-39, SLIP-0010, Ed25519	holds coins, signs transactions and claims
Secure-element device	device-bound Ed25519 key	holds a key and a monotonic counter that bound offline use
Payment terminal (certified)	terminal key, reader key management	reads cards, delivers metadata to the air-gapped device (§19)
Issuer	issuer key with rotation chain	issues and revokes tokens

0.2.2 Assumptions

- H1: BLAKE2b-256 and SHA-256 are collision resistant.
- H2: Ed25519 is existentially unforgeable (EUF-CMA).
- H3: ECVRF yields unique, pseudorandom outputs with proof.
- H4: At most 7 scheduling nodes act Byzantine (safety); at most 8 are unavailable (liveness).
- H5: The network is partially synchronous; a path exists between any two honest nodes.
- H6: External time is used only at bootstrap (median of 10 anchors).
- H7: A secure element and a certified card reader protect their keys and counters.

0.2.3 Attacker classes

Class	Goal	Main defence (section)
T1 Token forger	valid token without issuer	signature, domain tags, key registry (§14.1, §18)
T2 Replay	reuse across time, network or branch	request_id, network id, claim window, epoch binding (§18, §20.8)
T3 Clone	copy of a device or carrier	counter, one-time tag, credit limit (§18.4, §20.10)
T4 Partition and eclipse	isolate a node	fork choice, locator sync, finality (§1, §16, §17)
T5 Time manipulation	shift windows and maturity	median, drift cap, OTP chain (§6, §7)
T6 Miner collusion	divert reward, forge proofs	consensus rule §12.3, per-proof verification (§12)
T7 Storage, keys, channel	corrupt files, steal keys, eavesdrop on USB channel	authenticated encryption, block-store repair, channel protection (§17, §19.4, §20.13)
T8 Node misuse	equivocation, censorship, revocation abuse	quorum thresholds, rotation rules (§9, §16)

1. Network Topology: LSC Nodes

Instead of an open field of miners or a large PoS validator set, LanternSeaCoin defines a small, fixed cohort of **LSC nodes** (e.g., 12–48 nodes, fixed at 24 in §14.3):

- Each LSC node holds a long-lived, publicly known identity, rotated only via §9.1's key-rotation/revocation process, not per block.
- **Block production scheduling uses a concrete verifiable random function: ECVRF-EDWARDS25519-SHA512-TAI, per RFC 9381.** This replaces the earlier vague "round-robin or VRF-weighted" description flagged in §15.6 as a predictability risk. Concretely: each LSC node computes $(\text{output}, \text{proof}) = \text{ECVRF_prove}(\text{privkey}, \text{chain_time}[n] \parallel \text{block_height})$ for the upcoming slot; the node whose output is numerically lowest among all 24 produces that block, and includes proof in the block header so every other node can verify the selection was genuine (not manufactured) without needing to trust the producer's claim.
- **Why this resolves the predictability risk:** an outside attacker cannot compute any node's output in advance without that node's private key (that's the point of a VRF), so the schedule is unpredictable ahead of time to anyone except the node itself — closing the "precompute which node produces block N and target it in advance" attack named in §15.6 — while still being publicly verifiable after the fact.
- Fewer nodes means lower verification/storage redundancy and faster block finality, at the cost of weaker censorship-resistance than a fully open network like Bitcoin or Monero.

Block cadence: short target interval of ~60 seconds — deliberately close to Feathercoin's real-world ~1-minute block time (§0.1) — practical here because the small, known node set keeps propagation overhead low without needing proof-of-work at all.

1.1 Slot eligibility and leadership

Slot h has the input $\alpha_h = \text{enc}(t(h), h)$ (§14.1.2). Each node computes an output β_i and a proof π_i ; every other node verifies the proof with the public key. A node is **eligible** when β_i is below a threshold τ :

$$(\beta_i, \pi_i) = \text{Prove}(sk_i, \alpha_h), \quad \text{eligible}_i \iff \beta_i < \tau, \quad p = \tau/2^{256}$$

When eligibility proofs are exchanged (§1.2), the **leader** is the eligible node with the numerically lowest valid β — the rule described above. With N nodes of equal probability p , the number X of producers in a slot is binomial, and the fork probability per slot follows:

$$P(X = j) = \binom{N}{j} p^j (1-p)^{N-j}, \quad P(\text{fork}) = P(X \geq 2) = 1 - (1-p)^N - Np(1-p)^{N-1}$$

The probability of exactly one producer, $N \cdot p \cdot (1-p)^{N-1}$, peaks at $p = 1/N$ and is only 37.6 % for $N = 24$. Pure self-election therefore cannot deliver liveness and uniqueness at the same time. Slot outcomes in percent of slots:

$k = N \cdot p$	p per node	no producer	exactly one	fork (two or more)
1	4.2 %	36.0	37.6	26.4
2	8.3 %	12.4	27.0	60.6
3	12.5 %	4.1	13.9	82.0
6	25.0 %	0.1	0.8	99.1
12	50.0 %	0.0	0.0	100.0
24	100.0 %	0.0	0.0	100.0

1.2 Slot-proof gossip

From Series 2 (§13.2) every eligible node MUST publish (pk, π) in a window w_p (message slot-proof, 112 bytes; about 2.7 KB per slot for 24 nodes). The leader is the smallest valid β . Another node MUST NOT produce a block unless, after w_p , no smaller valid β is known. This reduces forks to partition cases. Fork choice (§1.3) remains mandatory.

1.3 Fork choice

A node MUST select its tip in this order:

1. Only tips consistent with all finalized checkpoints (§16.1).
2. From Series 3: only tips that contain every known sealed block (§1.5.5).
3. Greater height wins.
4. At equal height, the branch with the lexicographically smaller sequence of VRF outputs from genesis wins; this equals comparing β of the first diverging blocks.
5. Final tie-break: the smaller block hash.

$$T \succ T' \iff H > H' \vee (H = H' \wedge (\beta_0, B_0, \dots, \beta_H, B_H) <_{\text{lex}} (\beta'_0, B'_0, \dots, \beta'_H, B'_H))$$

Theorem 2 (independence of arrival order). The selected tip depends only on the set of known valid blocks (and, from Series 3, the set of known seals). *Proof sketch:* Pairs (β, B) are totally ordered because distinct blocks have distinct hashes. The lexicographic order on sequences of equal length is therefore a strict total order, and the best tip is the maximum of a finite totally ordered set, independent of enumeration order. Two tips of equal height share a prefix up to the common ancestor, so the sequence comparison equals the comparison of the first diverging blocks.

A verifier cannot know the proofs of all nodes; it can only establish that a proof is valid, not that its producer was the winner. Fork choice and finality (§16) close this gap.

1.4 Block acceptance and reorganization

Receive block (blockrec after inv and getdata)

- └ predecessor unknown? → ORPHAN: wait in pool, request predecessor via getdata
- └ block invalid? → INVALID: time, signature, ECVRF, segments, reward, chaining
- └ conflicts with checkpoint? → REJECTED_FINALITY
- └ not better than tip? → ACCEPTED_SIDE: side branch stored, remains candidate
- └ better than tip → ACCEPTED_TIP or REORG:
 - find common ancestor, restore state from cache, re-apply only d blocks,
 - reconcile mempool (valid operations return, conflicts drop),
 - check invariant (sum of coins equals sum of rewards), relay to peers

The state cache keeps the state of every adopted tip and of every 8th block, at most 64 states. A reorganization of depth d costs $O(d)$ block applications instead of $O(H)$. After every reorganization the invariant of §5.5 MUST hold.

As a heuristic with independent slots (f : fork probability per slot; ε : probability that a losing branch is extended before it learns of the winner):

$$P(\text{depth} \geq d) \approx (f \varepsilon)^{d-1}$$

For $f = 0.264$ ($k = 1$):

ϵ	$d = 2$	$d = 3$	$d = 5$	$d = 10$
0.1	2.6e-2	7.0e-4	4.9e-7	6.3e-15
0.01	2.6e-3	7.0e-6	4.9e-11	6.3e-24
0.001	2.6e-4	7.0e-8	4.9e-15	6.3e-33

1.5 Reorganization stabilization

LSC aims at rare and shallow reorganizations by construction, not by luck. This section follows the circulation strategy of the Central Bank of Lesotho (CBL) as the bank publishes it: currency is issued to satisfy public demand and the supply is managed so that enough cash is available at all times; a clean-note policy keeps quality up (mutilated notes are redeemed, only clean notes are reissued); old and new note series circulate side by side until the bank withdraws the old ones by legal notice, without a forced exchange; registers account for withdrawals and deposits; and the loti is pegged to the rand at par.

1.5.1 Mapping

CBL principle	LSC mechanism	Section
Issue currency to meet demand; manage supply so cash is always available	adaptive eligibility threshold; fallback rounds so that no slot stays empty	§1.5.3, §1.5.4
Clean-note policy: only clean notes are reissued	only the best known block per height is relayed; blocks with failing proofs are never relayed	§1.5.4
Old and new series coexist until withdrawn by legal notice; no forced exchange	a competing branch is withdrawn only by a quorum act (seal or checkpoint), never by a unilateral depth cap; rule changes apply after a fixed delay	§1.5.6
Parity with the rand as a fixed anchor	per-block endorsement seal as anchor: a sealed block is the fixed value every branch is measured against	§1.5.5
Registers of withdrawals and deposits	fork register: window statistics and equivocation evidence signed in checkpoints	§1.5.7
Security features built into the design	endorsements are domain-tagged Ed25519 signatures; a double endorsement is provable	§1.5.5

1.5.2 Goals

- **G1.** The fork probability per slot SHOULD be at most 1 %, given a proof-delivery failure probability $\epsilon_g \leq 1$ %.
- **G2.** A sealed block MUST NOT be reorganized while $f \leq 7$.
- **G3.** A reorganization of depth 3 or more SHOULD have a probability below 10^{-6} per fork.
- **G4.** No slot stays empty.

For comparison, pure self-election with $k = 3$ gives a fork probability of 82.0 % per slot, about 1,181 forks per day at 1,440 slots (§1.1).

1.5.3 Demand-matched issue of production rights (adaptive threshold) The expected number of eligible nodes per slot is $\lambda = N \cdot p$ (§1.1). The network regulates λ toward a target $\lambda^* = 3$, which leaves 4.1 % of slots without an eligible node and 13.9 % with exactly one; the leader is unique anyway once proofs are exchanged (§1.2). Every block records the eligibility proofs it saw, so the number c_h of proofs per produced block is verifiable on chain. Over a window of $W_c = 120$ blocks the mean m_w estimates the conditional expectation

$$E[X \mid X \geq 1] = \frac{N p}{1 - (1 - p)^N}$$

The window estimate $\hat{p}$ solves this equation for the observed m_w (it is monotone in p). The threshold changes only outside a dead band:

$$p_{w+1} = p_w \cdot \text{clamp}\left(\frac{\lambda^*}{N\hat{p}}, 0.95, 1.05\right) \quad \text{if } |N\hat{p} - \lambda^*| > 0.15 \lambda^*, \quad p_{w+1} = p_w \text{ otherwise}$$

For $\lambda = 3$ the standard error of m_w over 120 blocks is 0.14 (4.5 %), so the 15 % dead band is about 3.3 standard errors and sampling noise does not cause oscillation. The controller reacts to what noise does not explain: nodes that are revoked, admitted or offline change the active node count. Changes apply from height $h + 120$ (§13.4) and are written to the ParameterRegistry, so all nodes use the same threshold τ at every height.

1.5.4 Leader-first production, fallback rounds, relay policy Let ϵ_g be the probability that a given proof does not reach a given node within the proof window w_p . A node of rank j ($j - 1$ better proofs exist) produces despite a better proof only if it received none of them, which has probability ϵ_g^{j-1} . Therefore

$$P(\text{fork}) \leq P(X \geq 2) \sum_{j \geq 2} \epsilon_g^{j-1} = P(X \geq 2) \frac{\epsilon_g}{1 - \epsilon_g}$$

For $N = 24$ and $\lambda = 3$, $P(X \geq 2) = 0.820$:

ϵ_g	fork probability per slot	forks per day	reduction versus self-election
0.1	$\leq 9.1 \%$	≤ 131	factor 9
0.01	$\leq 0.83 \%$	≤ 12	factor 99
0.001	$\leq 0.082 \%$	≤ 1.2	factor 999

With the heuristic of §1.4 and $\epsilon = 0.01$, a reorganization of depth 3 has probability $(0.0083 \cdot 0.01)^2 \approx 6.9 \cdot 10^{-9}$ and depth 4 about $5.7 \cdot 10^{-13}$, which meets G3.

Fallback rounds. If no valid proof exists at the end of w_p , round $r + 1$ starts with $p_r = \min(2^r \cdot p, 1)$ and a new window w_p . The probability that a slot is still empty after r rounds is $4.1 \cdot 10^{-2}$, $1.0 \cdot 10^{-3}$, $6.0 \cdot 10^{-8}$ and 0 for $r = 1$ to 4; after at most three further rounds every node is eligible, which meets G4. Reference values: $w_p = 3$ s, and w_p MUST exceed the 99th-percentile propagation delay of a proof; four rounds fit into 12 s of the 60 s block interval.

Relay policy (clean notes). A node SHOULD relay a block only if it is the best known block at its height by §1.3 or if it is sealed. Competing blocks with larger β are not forwarded further, and blocks that fail validation are never relayed. This limits the spread of losing blocks. It is a policy and not a consensus rule, so Theorem 2 is unaffected.

1.5.5 Per-block endorsement seal (parity anchor) From Series 3 every node signs an endorsement for the block it considers the leader's block at height h after w_p (the valid block with the lowest β):

$$e_i = \text{Ed25519.Sign}(sk_i, \text{LSC-ENDORSE-v1} \parallel \text{enc}(h, B_h))$$

A node MUST endorse at most one block per height. A **seal** consists of $q = 16$ endorsements of the same block, each 65 bytes (64-byte signature, 1-byte node index), so 1,040 bytes per block or 0.05 % of block capacity. It travels in segment type 9 (SEAL) and in the message endorse (§17.2). Fork choice then excludes every tip that omits a known sealed block (§1.3, rule 2).

Lemma 4 (seal exclusivity). If $f \leq 7$, two different blocks of the same height cannot both be sealed. *Proof sketch:* Two seals need quorums Q_1 and Q_2 of at least 16 nodes. Their intersection has at least $2q - N = 8$ members, which exceeds f , so at least one honest node would have endorsed both blocks, which the one-endorsement-per-height rule excludes.

Sealing succeeds when at least 16 of the n_{up} available nodes receive the leader's proof. Each does so with probability $1 - \epsilon_g$, so $P(\text{seal}) = P(\text{Bin}(n_{up}, 1 - \epsilon_g) \geq 16)$:

ϵ_g	$n_{up} = 24$	$n_{up} = 21$	$n_{up} = 18$
0.1	0.9997	0.9856	0.7338
0.01	1.0000	1.0000	0.9993

A height that cannot be sealed (for example after a split vote) stays unsealed and the chain continues by fork choice; nothing waits for seals, so seals add no liveness requirement. Two valid endorsements of one key at the same height for different blocks are evidence for a NodeRevocationRFC (§9.1).

1.5.6 Withdrawal only by formal act; rule changes with delay A unilateral cap D_{max} on the reorganization depth is deliberately not used. After a healed partition in which neither side was sealed (for example a 12/12 split), nodes would refuse to merge the branches and the split would become permanent. The Loti behaves the other way round: older series stay legal tender until the bank withdraws them by legal notice. LSC follows this: a competing branch is withdrawn only by a quorum act, a seal (§1.5.5) or a checkpoint (§16.5). Parameter changes such as the threshold τ and the activation of series take effect at $h_{act} \geq h + 120$ (§13.3, §13.4), so nodes never disagree about eligibility at a given height.

1.5.7 Fork register From Series 2 every checkpoint carries window statistics: the mean number of proofs per block m_w , the number of heights that needed fallback rounds and the number of known endorsement equivocations. They are signed under the checkpoint quorum and form the audit basis for the controller (§1.5.3) and for revocation evidence. Nodes MAY additionally log local reorganizations (depth, time) off protocol; these are not consensus data.

1.5.8 Expected effect

Configuration	fork probability per slot	forks per day
all 24 nodes always eligible (stress test)	100 %	1,440
self-election, $k = 3$	82.0 %	1,181
slot-proof gossip, $\epsilon_g = 0.1$	≤ 9.1 %	≤ 131
slot-proof gossip, $\epsilon_g = 0.01$	≤ 0.83 %	≤ 12
slot-proof gossip, $\epsilon_g = 0.001$	≤ 0.082 %	≤ 1.2

Configuration	fork probability per slot	forks per day
plus endorsement seals (Series 3)	sealed blocks are never reorganized (G2)	—

The bounds assume independent proof delivery. ϵ_g is a property of the network and **MUST** be measured on a testnet before the bounds are relied upon (§15.9).

2. Block Structure and the Segmented Data Model

A block's data is not one bulk payload — it is deliberately **cut into many small, individually defined segments**:

```
Block {
  header:      { prev_refs, height, timestamp, lsc_node_signature }
  transactions: [ ... ]
  segments:    [ Segment_1, Segment_2, ..., Segment_N ] // many, not one blob
}
```

```
Segment {
  segment_id:      4 bytes
  segment_type:    enum { LIGHT_PASS, HEAVY_PASS, TIME_ANCHOR, BLINK }
  raw_len:         int
  transformed:     bytes // output of the pass-through op, see §3
  position_index:  int    // this segment's fixed slot in the block
}
```

- **Many small segments, not one large one.** Each segment is independently checkable against its declared `raw_len` and `position_index`, so a corrupted or missing segment is locally detectable rather than invalidating an entire monolithic payload.
- Segments are **not compressed and not reassembled through any reconstruction algorithm** (no RAID-style parity, no erasure coding). Each one simply passes through its designated transformation and sits at its fixed position — a reader either has each segment's transformed content, or it doesn't.
- Data capacity is spent deliberately on carrying more raw application data per block rather than minimizing footprint, since the small stable node set already keeps propagation cheap.

2.1 Block hash and chaining

The block hash B_h uses the unambiguous encoding of §14.1.2. It contains height, time, predecessor (empty at genesis) and the body root ρ_h :

$$B_h = H(\text{enc}(h, t(h), |P|, P, \rho_h)), \quad P = \begin{cases} \emptyset & h = 0 \\ \{B_{h-1}\} & h > 0 \end{cases}$$

The body root binds the reward distribution, operations, segments and the ECVRF proof π :

$$\rho_h = H(\text{enc}(\text{reward}, (a_i, v_i)_i, \text{ops}, (d_j)_j, \text{segments}, (\tau_k, \ell_k, \text{pos}_k, H(x_k))_k, \text{vrf}, \pi))$$

The producer **MUST** sign B_h with the tag LSC-BLOCK-SIG-v1.

Theorem 1 (chain integrity). If a block of height i is altered, every B_j with $j \geq i$ changes. An undetected change requires a collision of H ; for n hashes the probability is at most $n^2/2^{257}$, about 2^{-177} for $n = 2^{40}$. *Proof sketch:* B_{i+1} contains B_i ; induction over j .

2.2 Segment types

A block carries up to 2 MiB of payload in typed segments:

Type	Name	Content	Check	Limit
1	LIGHT_PASS	ChaCha20 transform with public key and nonce	length equals raw_len	none
2	HEAVY_PASS	Argon2id commitment, 32 bytes	length 32	at most 32 per block
3	TIME_ANCHOR	time anchor	structural	none
4	BLINK	aggregate of timing probes	plausibility (class counts sum to total, $\min \leq \text{median} \leq \max$)	at most 512 KiB (25 %)
5	MINING_PROOF	single work proof	recomputation with context	none
6	MICRO_PROOF	bundled micro-proofs	per-proof verification, reward rule	at most 500 per block
7	LINEAGE	coin-lineage root (§20.2)	recomputation	at most 1 per block
8	THREAD	MMR root and OTP binding (§20.3)	recomputation	at most 1 per block
9	SEAL	quorum seal (§20.9)	signature verification	at most 1 per block
other	unknown	future extension	skipped	none

LIGHT_PASS is not encryption: key and nonce are public, the step is reversible and cheap. HEAVY_PASS is the expensive tier. Segment positions MUST ascend without gaps. A malformed segment MUST lead to INVALID and MUST NEVER crash the node.

2.3 Rejection conditions

A block MUST be rejected for: timestamp unequal to $t(h)$, reward sum unequal to $R(h)$, invalid signature or ECVRF proof, wrong predecessor or height, invalid segment, violation of segment limits, a reward distribution that contradicts the micro-proofs, or a conflict with a finalized checkpoint.

3. Dual-Tier Pass-Through Operations

Two transformation tiers apply to LIGHT_PASS and HEAVY_PASS segments — no asymmetric keys, no external keyholder, nothing to encrypt/decrypt with a private key held elsewhere. **Both tiers now use established, independently reviewed primitives instead of a custom-built transform** — §15.1 previously flagged the earlier “chained modular-arithmetic” description as an unreviewed homegrown construction, and this replaces it rather than leaving it as a named risk:

Tier	Primitive	Size	Purpose
Light pass	ChaCha20 (RFC 8439), used as a fixed-key keystream transform with a protocol-wide public key/nonce scheme	small	A fast, widely-audited stream cipher used here purely as a deterministic byte-scrambling transform (not for confidentiality — the key is public protocol data, per §3's transparency-by-design goal). Cheap enough to apply to many small segments quickly.
Heavy pass	Argon2id (winner of the 2015 Password Hashing Competition, RFC 9106), run with tunable memory/time/parallelism cost parameters	large	A memory-hard, independently reviewed function — deliberately reused here for its “expensive and hard to shortcut” property rather than its original password-hashing purpose, giving the “costlier to compute” behavior §3 always wanted, backed by a real specification instead of an ad-hoc pipeline.

- Both tiers are self-contained, fixed protocol transformations — any node holding the same shared parameters (ChaCha20 key/nonce scheme; Argon2id cost parameters) can apply or verify them without needing a private key.
- **Why this specific swap:** ChaCha20 and Argon2id were chosen because each already has the exact property its tier needed — ChaCha20 is fast and simple for high-volume light segments, Argon2id is deliberately expensive and memory-hard for the bulk “substance” segments — without inventing new cryptographic logic to get there. This directly resolves §15.1: the pass-through tiers are no longer a novel, unreviewed construction; they're a documented, public assembly of two standard, published primitives.
- This remains a deliberate trade for transparency over confidentiality: unlike CryptoNote's ring-signature approach (§0.1), no confidentiality claims are made about LIGHT_PASS/HEAVY_PASS content — it is meant to be a data-transport and data-weight mechanism, not a privacy mechanism. Using a stream cipher and a memory-hard function with public keys/parameters does not change that; it changes only how trustworthy the transform's *behavior* is, not its confidentiality properties.
- **Residual note:** swapping in reviewed primitives removes the “unreviewed construction” risk, but the specific way they're *assembled* into a segment pipeline (ordering, how `raw_len` interacts with each primitive's block/parameter sizing) is still new and should get its own focused review before being relied on — using audited building blocks lowers risk, it doesn't eliminate the need to check the assembly.

4. Blink Subsystem: Timing-Based Topology Detection

A dedicated **~20-30% of each block's capacity** is reserved for BLINK segments — small, frequent probe records:

```
BlinkRecord {                                     // payload when segment_type = BLINK
  probe_id:      16 bytes
  send_ts:       timestamp                        // when this probe left the sender
  recv_ts:       timestamp                        // when it arrived at the recorder
  peer_hint:     partial_route                    // which peer(s) it passed through
  seed_out:      prng_output                      // see below
}
```

- LSC nodes periodically send small probe packets to peers and record `send_ts` / `recv_ts`. The timing gap plus `peer_hint` lets the network build a rolling picture of link latency and routing paths.
- **Timing-seeded PRNG:** each probe's `send_ts` (sub-millisecond jitter) feeds a pseudorandom generator producing `seed_out`, which decides which peer gets probed next — so the probing pattern is derived from real timing entropy rather than a fixed round-robin.
- Blink data is **not encrypted** — it's meant to be visible on-chain so any node can independently verify the network's observed topology and latency history (e.g., detecting a node gone dark, a degraded route, or a new peer).
- **Sizing rationale for 20-30%:** enough probe density for a statistically meaningful rolling topology map, without crowding out LIGHT_PASS/HEAVY_PASS segment space.

Reduced attribution to fix §15.4's reconnaissance exposure. §15.4 flagged that publishing raw, per-node timing and routing data for a small set of only 24 *identified* nodes (§14.3) makes targeted denial-of-service or traffic-analysis attacks against a specific node easier than the same telemetry would be on a large anonymous network. Two changes address this without giving up the subsystem's actual purpose (detecting dead nodes and degraded routes):

- **peer_hint is downgraded from a specific route to a coarse routing class** — e.g., "same region," "cross-region," "unknown" — rather than identifying the literal intermediate peer(s) a probe passed through. This still supports the subsystem's real goal (spotting a degraded *class* of route) without handing an attacker a live map of exactly which nodes talk to which.
- **Raw per-probe records are aggregated before publication.** Instead of writing each individual `BlinkRecord` directly on-chain in real time, LSC nodes publish a **rolling aggregate** per time window (e.g., a 1-minute bucket, aligned to the block cadence from §1): min/median/max latency and a coarse routing-class distribution, with `probe_ids` and exact per-record timestamps *not* included in the published aggregate. The raw records still exist locally (so a node can investigate its own recent history if needed) but only the statistical summary is broadcast network-wide.
- **Net effect:** the network still gets what it actually needs — a usable rolling picture of latency and route health, sufficient to detect a node going dark or a route degrading — without publishing a live, per-node, per-route dataset that doubles as a targeting map for a small, already-identified node set. This is a direct fix to §15.4, not just a mitigation noted in trade-offs.

Validation of the aggregate. A BLINK segment **MUST NOT** exceed 512 KiB (25 % of the 2 MiB block limit, §14.3). Nodes **MUST** check plausibility: the class counts sum to the total count, and $\min \leq \text{median} \leq \max$ for each latency statistic. Individual probes are never transmitted.

5. Coins, Movability, and Offline-Redeemable Stamps

A coin carries an explicit movability flag and zero or more **stamps**:

```
Coin {
```

```

amount:      value
owner:       address
movable:     bool
stamps:      [ Stamp_1, ..., Stamp_k ]    // optional, 0 or more
}

Stamp {
  stamp_id:      8 bytes
  issued_at:     block_height
  claim_window:  block_height_range
  claimed:       bool
  claim_proof:   optional_hash
  chain_index:   int64                // OTP-chain position at mint time, see §7
  otp_checkpoint: 32 bytes            // OTP-chain value at mint time, see §7
}

```

- **A stamp is not a snapshot.** A snapshot would be an external, point-in-time copy of a coin's state stored elsewhere. A stamp is baked into the coin's own definition from the moment it's minted, and travels with the coin.
- Each stamp represents a fixed, non-transferable claim right — redeemable once within its claim window, by presenting a claim_proof that matches the expected OTP-chain position (§7). This non-reuse check is structurally inspired by CryptoNote's key-image mechanism (§0.1) — checkable without revealing identity, and here without even needing a live connection to the LSC nodes at the moment of redemption.
- **Movability rule:** a coin with movable: false cannot be transferred between wallets at all — it exists solely to be redeemed via its stamp(s) (a payout coupon, a fixed claim right, a one-time voucher). A coin with movable: true is an ordinary transferable coin, and may or may not also carry stamps.
- **Wallets only ever hold movable coins.** Non-movable, stamp-only coins live at the protocol/issuance layer until claimed — claiming a stamp is what *produces* a movable coin (or a direct payout) into a wallet. This keeps wallet logic simple: everything a wallet holds is, by definition, movable.
- This gives LanternSeaCoin a two-layer coin economy: **non-movable, stamp-bearing coins** (claim rights, vouchers, offline-redeemable within a window) and **movable coins** (the liquid, wallet-held, freely transferable tier that a stamp claim converts into).

Design note: published research on offline e-cash and delegatable coin systems shows a recurring pattern — tokens usable without live network contact typically need either a trusted hardware enclave or a strict non-reuse rule to prevent double-spending, since there's no live ledger check at the moment of use. Real-world “non-fungible but grouped” token designs (e.g., collectible digital-stamp programs) show a workable precedent for a token that is individually unique but not freely tradeable. LanternSeaCoin's stamp design borrows that general shape, plus the CryptoNote-style non-reuse check described above, rather than inventing new offline-spend cryptographic guarantees entirely from scratch — see the open trade-offs in §10.

5.1 Coin lifecycle

A wallet holds freshly mined coins locked: they become spendable only after the maturity window and a signed claim. This is a core property of the design.

A coin has an identifier, an amount, an owner, the flag movable, an origin (MINED, MINED-SPLIT or TRANSFER), optionally a stamp, and the flag spent. A stamp carries its identifier, the height of issue, the start of the claim window, the OTP value at issue and the state claimed. From Series 2 a coin additionally carries its lineage (§20.2).

State	spendable	Condition	Transition
ISSUED	no	reward in block h, window still closed	from $t \geq w_c$: CLAIMABLE
CLAIMABLE	no	window open, stamp not redeemed	signed claim: CLAIMED
CLAIMED	yes	stamp redeemed	transfer: SPENT
TRANSFER coin	yes	output of a payment	transfer: SPENT
SPENT	no	spent	final

The window opens 3 hours after the block and is rounded up to the full hour (§12.3); the wait is therefore 3 to 4 hours, 180 to 240 blocks at 60-second blocks:

$$w_c = 3600 \left\lceil \frac{t(h) + 10800}{3600} \right\rceil$$

A transfer that spends a stamped coin **MUST** be rejected, even after the window has opened, as long as the claim is missing. Only the earning address **MAY** redeem; the signature covers LSC-STAMP-CLAIM-v1 and the stamp identifier. The claim is an operation in a block whose block time **MUST** be at least w_c , and it can happen once. Afterwards the coin is movable. Unclaimed stamps do not expire; they wait indefinitely for the claim.

5.2 Why mined coins are locked

Reward coins vanish with their block if a reorganization displaces it. The maturity $M \geq 180$ blocks exceeds the reorganization depths of §1.4 and three times the checkpoint interval; the reward is normally final when it is released. For a target probability ϵ_M :

$$M \geq 1 + \frac{\ln \epsilon_M}{\ln(f \epsilon)}$$

Alignment to the full hour bundles releases onto few fixed instants.

5.3 Wallet view

getbalance **MUST** count only movable, unspent coins. getwalletinfo additionally reports immature_balance, and listtransactions marks locked coins as immature. Seed, key derivation and custody follow §14.7.1 (BIP-39, SLIP-0010).

5.4 Transactions

A simple transaction has one input and one output; its amount **MUST** equal the amount of the input coin. Several inputs and outputs allow change and fees; the signature of every input **MUST** bind the entire output set. Signing uses LSC-COIN-SPEND-v1. From Series 3 a transaction **MAY** carry an expiry height h_{exp} (§20.8). Format details are in §14.7.

5.5 Conservation of supply

Theorem 3. After every block of height h :

$$\sum_{c \text{ unspent}} a(c) = \sum_{k=0}^h R(k)$$

Proof sketch by induction over blocks: A block creates coins whose amounts sum to $R(h)$; validation checks this. A claim changes no amount. A simple transaction requires amount equal to input amount, a multi-output transaction requires the sum of inputs to equal the sum of outputs. A reorganization re-applies state from verified blocks. Nodes **MUST** check the invariant after reorganizations.

6. The Chain-Native Clock: Bootstrap Phase

Client and node local clocks are **never trusted** by the protocol. Time enters the system exactly once, during a 10-block **bootstrap phase**:

```
BootstrapTimeAnchor {
    block_height:    int           // used only for block heights 1–10
    pushed_utc:      int64         // 1 through 10
    pusher_id:       lsc_node_id   // this LSC node's own UTC reading
    signature:       bytes
}
```

- For blocks 1–10, each producing LSC node self-pushes its own UTC reading, signed with its identity.
- Once all 10 exist, the network takes the **median** of the 10 self-pushed values (median, not mean, so a single outlier can't skew the reference) as `genesis_time_reference`.
- This median is the **one and only** externally-sourced time value the entire chain clock is ever built from. After block 10, no further external UTC input is accepted.

6.1 Robustness of the bootstrap median

Ten nodes sign one UTC timestamp u_i each; the median is the genesis time anchor G (tie-break per §14.4). A forged anchor makes the whole round fail.

$$G = \text{median}(u_1, \dots, u_{10})$$

With at most 4 wrong anchors, G lies between the smallest and the largest honest value. From 5 colluding anchors an attacker can shift G .

7. Time Threading and the UTC-Anchored OTP Chain

From block 11 onward, time is **threaded forward** purely from chain structure — never read from any clock:

```
chain_time[11] = genesis_time_reference + (11 * expected_block_interval)
chain_time[n]  = chain_time[n-1] + expected_block_interval + adjustment[n]
```

- `expected_block_interval` is the network's fixed target cadence (§1, ~60 seconds). `adjustment[n]` is a small, chain-derived correction term based on recent actual block-production speed, letting the threaded clock self-correct without ever consulting an external clock again.
- Every block still carries a `TIME_ANCHOR` segment:

```
TimeAnchorSegment {
    utc_timestamp:    int64           // payload when segment_type = TIME_ANCHOR
    drift_tolerance:  int16           // = chain_time[n] from block 11 onward
    otp_checkpoint:   32 bytes        // +/- seconds of threading-calculation slack
    chain_index:      int64           // current OTP-chain value at this block
}
// how many OTP steps since genesis
```

- A parallel **OTP chain** advances once per block via simple chained hashing:

```
otp_checkpoint[0] = H(genesis_seed || utc_timestamp[0])
otp_checkpoint[n] = H(otp_checkpoint[n-1] || utc_timestamp[n] || chain_index[n])
```

$H(\dots)$ is a standard one-way hash — deliberately simple so any node or client can independently recompute and verify it, in the same spirit as CryptoNote's use of a one-way function to derive a key image from a private key (§0.1) without needing anything more exotic. Each stamp (§5) records the `chain_index/otp_checkpoint` active at mint time as part of its claim proof: a replayed old claim will correspond to a stale `chain_index` and be visibly out of step once compared against real chain history.

- **drift_tolerance** now expresses uncertainty in the threading calculation itself (accumulated rounding/adjustment slack), not clock-to-clock skew, since there's no longer more than one independent clock to disagree with.

7.1 Concrete clock formula and drift bound

$$t(h) = G + h \cdot T + A_h, \quad T = 60 \text{ s}$$

A_h is the drift adjustment (the adjustment term above, accumulated). It changes only in windows of $W = 10$ blocks and by at most $\delta = 5 \text{ s}$, formed from the median of the observed deviations:

$$|A_{h+W} - A_h| \leq \delta = 5 \text{ s}$$

The maximum drift rate is $5 \text{ s} / (10 \cdot 60 \text{ s}) = 0.83 \%$. The comparison tolerance is 2 s . Block validation **MUST** take A_h into account.

7.2 Branch independence of the OTP chain

The OTP chain is a hash chain $o_h = F(h, o_{\{h-1\}})$ built on BLAKE2b-256, equivalent to the definition above when `utc_timestamp[n]` is the chain time of height n . It depends only on height and predecessor, not on block content, and is therefore identical on all branches.

8. Offline Client Counting and Resync

While offline, a client:

1. Remembers the last block it actually saw: its `chain_time`, `chain_index`, and `otp_checkpoint`.
2. May use its own device clock only to *guess* how many block-intervals have likely passed — a rough step-counter, never a trusted time source.
3. Advances its own **local copy** of the OTP chain forward that many steps, using the same threading formula as §7.
4. Any stamp claim made offline is stamped with the client's locally-advanced `chain_index/otp_checkpoint`.

Once back online, the client:

1. Fetches the real block history it missed (each with its own real `TIME_ANCHOR` segment).
2. Compares its locally-advanced chain values against the real ones at the matching `chain_index`.
3. If within `drift_tolerance` → the offline claim is accepted as-is.
4. If outside tolerance → the claim is held as **pending**, and the client re-derives the correct checkpoint from the now-available real chain data to retry.
5. The client's local chain realigns to the network's real chain from this point forward, until it goes offline again.

This narrows (but does not eliminate) the offline-abuse window from §5/§10: a stale or wildly drifted claim becomes detectable and self-correcting the moment connectivity returns.

8.1 Resync across reorganizations

A client that was offline for n intervals computes (h', t', o') locally and reconciles with the canonical chain on reconnect. If time and OTP value match within the tolerance, it keeps its estimate; otherwise it adopts the canonical state. The resync **MUST** work even if the chain reorganized in the meantime: the server finds the most recent locator entry on its canonical chain and replies with the hashes after it (§17.3).

9. Clock-Correction Governance

Because the threaded clock (§7) is the network's only time source after block 10, changes to the threading formula, adjustment term, or drift-tolerance sizing go through a formal, **unanimous** vote:

```
CorrectionRFC {
  rfc_id:          id
  proposed_change: description
  affected_params: [ param_names ]
  proposer:        lsc_node_id
  status:           enum { AWAITING, ACCEPTED, WITHDRAWN }
  votes:           [ (lsc_node_id, yes/no/unvoted) ]
  quorum_required: "ALL"           // fixed at 100%
}
```

- **Every** LSC node must cast an explicit **yes** for an RFC to pass — a single no, or a single non-response, blocks it.
- There is **no forced resolution deadline**. status remains AWAITING for as long as any node hasn't voted; an unvoted entry is neither yes nor no. The proposer may voluntarily withdraw a proposal, but the protocol itself never times one out into acceptance or rejection.
- Until an RFC passes, the existing threading formula (§7) keeps operating unchanged — there is never a moment where clock logic shifts without every node's explicit agreement, and a change only ever takes effect from a specified future block height, never retroactively.

Bootstrap eligibility is gated by voting participation:

```
BootstrapEligibility {
  lsc_node_id:      id
  eligible:         bool
  ineligibility_reason: optional string // e.g., "unvoted on RFC-0007"
}
```

- A node that fails to vote (yes or no) on an open CorrectionRFC is not penalized in its normal block-production role, but loses eligibility to serve as one of the 10 bootstrap nodes (§6) in any future bootstrap phase (e.g., a chain restart or new bootstrap round), until it re-establishes a clean voting record.
- This creates a direct incentive: a node that wants a voice in seeding the network's one authoritative time reference must actively vote on every clock-related proposal, not just the ones it agrees with.

9.1 Node Key Rotation and Revocation

§15.2 flagged that nothing in earlier sections defines what happens when an LSC node's key is compromised, or how a permanently-unresponsive node is ever removed. This adds both, delib-

erately using a **different vote rule than CorrectionRFC** — unanimity is wrong for this specific case, because a compromised or malicious node would simply vote “no” (or stay silent) on its own removal forever, exploiting the same unanimity property that protects the clock formula (§9, §10).

Routine key rotation:

```
KeyRotation {
  lsc_node_id:      id
  old_pubkey:       Ed25519 public key
  new_pubkey:       Ed25519 public key
  signature:        signed by old_pubkey, proving continuity of control
  effective_height: block_height
}
```

- Any LSC node may rotate its own signing key at any time by publishing a KeyRotation record signed by its *current* key — this requires no vote from other nodes, since it's an ordinary continuity operation, not a governance change.

Revocation (for a compromised or unresponsive node):

```
NodeRevocationRFC {
  rfc_id:          id
  target_node_id:   lsc_node_id          // the node being considered for removal
  reason:          enum { SUSPECTED_COMPROMISE, PERMANENTLY_UNRESPONSIVE, VOLUNTARY_EXIT }
  proposer:        lsc_node_id
  status:          enum { AWAITING, ACCEPTED, WITHDRAWN }
  votes:           [ (lsc_node_id, yes/no/unvoted) ]
  quorum_required: "ALL EXCEPT target_node_id" // deliberately excludes the node being voted
}
```

- **Quorum is unanimity among every node *other than the one being removed*** — this is the specific fix that prevents a compromised node from blocking its own revocation the way it could under a CorrectionRFC-style all-inclusive vote. With 24 nodes (§14.3), that means 23 must agree.
- On ACCEPTED, the target node's current signing key is added to a network-wide **revocation list** (published on-chain, checkable by any node), and any block, CorrectionRFC vote, or BootstrapTimeAnchor signed by that key after the revocation's *effective_height* is invalid.
- A revoked node's slot is not silently left empty — its removal must be paired with an already-agreed replacement-node identity in the same NodeRevocationRFC (a *replacement_node_id* field, omitted above for brevity but required in practice), keeping the total count at 24 (§14.3) rather than shrinking the active set over time.
- **Residual note:** this resolves the “no revocation mechanism” gap structurally, but the harder judgment call — *deciding* whether a node is actually compromised versus just slow, and getting 23 independent operators to agree on that in practice — is a social/operational process this paper cannot fully specify. The vote mechanism is safe; the real-world judgment behind casting each vote still requires competent, attentive node operators.

9.2 Effect of accepted proposals

An accepted CorrectionRFC MUST create an entry in the ParameterRegistry (§13.4) with an activation height; an accepted NodeRevocationRFC MUST take effect from its *effective_height*. Hard parameters require unanimity, soft features require $q = 16$ votes (§13.4). A revoked node's ECVRF proofs are no longer accepted by the registry.

10. Open Trade-offs (Consolidated)

- **Weaker censorship-resistance.** A small, known LSC-node cohort is easier to pressure, subpoena, or collude against than a large open miner/validator set — the trust model is closer to a federated/consortium chain than a fully trustless one, and closer to Feathercoin's checkpointed-but-still-open model than to it in practice (§0.1).
- **No privacy layer.** Sender, receiver, and amount are all plainly visible on-chain; this design makes no confidentiality claims, unlike the CryptoNote/Monero lineage it partially draws from (§0.1).
- **No data-reconstruction safety net.** A lost or corrupted segment (§2) is simply lost — there's no RAID-style parity to recover it. A production system would need to decide if a light redundancy scheme is worth adding without reintroducing full erasure coding.
- **Shared-parameter transformations need cryptanalysis.** Since LIGHT_PASS/HEAVY_PASS (§3) use fixed public parameters rather than external keys, the "heavy pass" tier's operations need real review to confirm they don't leak more than intended.
- **Offline stamp redemption still inherits classic offline e-cash double-spend risk** (§5, §8): without a live check at redemption time, a stamp could in principle be claimed more than once unless enforced by hardware (e.g., a secure enclave) or reconciled quickly on reconnect. The CryptoNote-style non-reuse check (§0.1, §5) narrows this window but, unlike a real key-image check, cannot be verified in real time by the whole network while offline — only after reconnection.
- **Bootstrap trust concentration.** The chain's entire sense of time traces back to only 10 self-pushed samples (§6), median-filtered but never re-anchored against outside truth afterward — a skewed or colluding bootstrap majority biases the clock permanently.
- **Unanimity risk: a single holdout can freeze clock governance forever.** If even one LSC node goes offline permanently or simply disagrees, the clock formula can never be corrected again (§9) — a real deployment would need a separate process for retiring/replacing an unresponsive node, not specified here.
- **Bootstrap-eligibility loss is currently one-directional.** No re-qualification rule is defined yet for a node that abstains once; this may be overly harsh as written.
- **Perverse incentive to always vote "no."** Because silence costs a node its bootstrap eligibility, a cautious node might reflexively vote "no" on every RFC just to stay eligible — combined with the unanimity requirement, this could make the clock formula effectively unchangeable in practice.
- **Checkpointing and archival — Fixed, see §16.** Feathercoin's Automatic Checkpointing (§0.1) was previously cited as relevant precedent without being built into the design; §16 now adds a 2/3-quorum, hourly checkpoint mechanism giving the chain hard finality boundaries, plus a configurable archive-export routine that always preserves complete, unpruned mining-segment data to operator-chosen storage.
- **The anti-spam MINING_PROOF segment (§12) does not add consensus security.** It raises the cost of spamming segment data but does nothing to address collusion among LSC nodes, the unanimity-freeze risk (above), or bootstrap trust concentration (§6) — those remain exactly as risky as before §12. Treating the mining segment as a general "safety" upgrade would be a misread of what it actually does. Offering three interchangeable algorithms (NeoScript-, RandomX-, and Equihash-style, §12) doesn't change this — it's a choice of anti-spam puzzle, not a choice of consensus security model.
- **The dynamic segment registry (§13) has no network-wide compatibility story yet.** A node simply ignoring segment types newer than its own build is fine during active development with a small, coordinated node set, but it is explicitly not a solution for a mature network where different LSC nodes might run different versions indefinitely — that needs a real activation/soft-fork process, which §13.1–§13.3 now define.
- **Proof-weighted reward splitting (§12.3) assumes honest difficulty reporting is enforceable.** The split formula itself is safe against a miner inflating its own share by submitting many cheap proofs (weight tracks real solved difficulty, not proof count), but it still depends on every node independently re-verifying each proof_hash against its claimed target — a node that skips this check could be fed a fabricated high-weight proof. This should be treated

as a mandatory validation step, not an optional one.

- **The 1-hour-aligned maturity checkpoint (§12.3) is a batching convenience, not a security boundary.** Rounding every mined coin's unlock time up to the next full chain-time hour makes maturity checks predictable, but it also means a coin mined just after an hour boundary waits almost a full extra hour compared to one mined just before it — a minor fairness asymmetry worth naming, not a flaw serious enough to abandon the batching benefit.
 - **Micro-proof aggregation (§12.4) lowers the participation bar, not the total work required.** Many small, low-difficulty proofs can add up to the same `block_target_weight` a single large proof used to represent, so this doesn't weaken the mining requirement itself — but it does mean a block can now legitimately contain many tiny payout entries, and a wallet or explorer surfacing mining rewards should expect a long tail of very small MinedCoin amounts rather than one lump sum per block, which is a UX/indexing consideration worth planning for, not a security flaw.
-

11. Summary

LanternSeaCoin (LSC) is a conceptual ledger built on a small, stable node set — deliberately closer to Feathercoin's lightweight, fast-settlement philosophy than to an open proof-of-work miner pool.

- **Data model.** Block data is stored as many independently checkable segments passed through one of two fixed transformation tiers (ChaCha20 and Argon2id): no compression, no asymmetric keys, no reconstruction algorithm. Segment types are versioned in a registry that new releases extend one type at a time (§2, §3, §13).
- **Scheduling and finality.** A VRF (ECVRF per RFC 9381) decides who may produce a block; a deterministic total order selects the chain; an hourly 2/3-quorum checkpoint gives hard finality (§1, §16).
- **Blink.** A subsystem lets the network map its own topology from real packet timing, published only as aggregates (§4).
- **Coins.** A two-layer economy of movable coins and non-transferable, offline-redeemable stamps, whose non-reuse check borrows the structural shape of CryptoNote's key image without its ring-signature privacy. Mined rewards are held under a chain-time-aligned, three-hour maturity stamp (§5, §12.3).
- **Clock.** No external clock is trusted after a 10-block self-reporting bootstrap; time is threaded forward from chain structure, and any change to the clock formula requires unanimous node agreement (§6–§9).
- **Mining.** An optional MINING_PROOF segment offers a choice of NeoScrypt, RandomX or Equi-hash purely as an anti-spam layer, scalable down into micro-proofs that low-power hardware can contribute, with proof-weighted reward splitting (§12).
- **Concreteness.** §14 pins the design to implementable parameters; §15 replaces every named security gap with a concrete fix built from established, independently reviewed standards and adds a layered attack analysis.
- **Extensions.** The network and node process (§17), redemption (§18), the terminal and air-gap layer (§19), the 14 banknote-inspired security layers (§20) and the sovereignty analysis (§21).

Every design choice trades something explicit for something else; those trade-offs are named throughout, most fully in §10. As §15.10 states, the specification closes every known gap on paper; it does not substitute for the empirical validation of §15.9, which no specification can complete by itself.

12. Mining Segments: Auxiliary Proof-of-Work

Two ways of introducing Monero- and Feathercoin-style mining into LanternSeaCoin were considered. Only one is adopted here; the other is documented as a rejected alternative, because the

choice materially changes what kind of coin this is.

12.1 Adopted: MINING_PROOF as an anti-spam segment

A new segment type is added to the enum from §2:

```
Segment {
  segment_id:      4 bytes
  segment_type:    enum { LIGHT_PASS, HEAVY_PASS, TIME_ANCHOR, BLINK, MINING_PROOF }
  raw_len:         int
  transformed:     bytes
  position_index:  int
}

MiningProofSegment {
  // payload when segment_type = MINING_PROOF
  algo:            enum { NEOSCRYPT_STYLE, RANDOMX_STYLE, EQUIHASH_STYLE }
  nonce:           bytes
  target:          bytes // difficulty target for this block's proof
  proof_hash:      32 bytes // resulting hash, must be <= target
  proof_params:    optional bytes // algorithm-specific parameters, e.g. Equihash (n, k)
}
```

- The block's producing LSC node (chosen by the existing schedule in §1 — this does not change) must additionally solve a small proof-of-work puzzle before the block is accepted by peers.
- **Algorithm choice is deliberately borrowed, not reinvented:** NEOSCRYPT_STYLE mirrors Feathercoin's actual memory-hard, GPU-friendly/ASIC-resistant NeoScript design (§0.1); RANDOMX_STYLE mirrors Monero's current (post-2019) CPU-friendly, ASIC-resistant RandomX algorithm; EQUIHASH_STYLE mirrors Zcash's Equihash — a memory-hard proof-of-work based on the computationally hard generalized birthday problem, built on an enhanced version of Wagner's algorithm, chosen specifically because generating a solution demands a large, tunable amount of memory while verifying one is fast (Zcash's own production parameters are $n=200$, $k=9$). All three share the same underlying goal — keeping proof generation memory- or CPU-bound rather than favoring specialized ASIC hardware — which is why all three are offered as interchangeable options behind the same MiningProofSegment shape rather than three separate segment designs.
- **What this buys:** the puzzle is set at a difficulty tuned only for *spam resistance*, not for competitive block-race security — its job is to make it computationally costly for a compromised or malicious node to flood the network with junk LIGHT_PASS/HEAVY_PASS/BLINK segments (§2–§4), not to decide which node gets to produce the next block. Block production selection stays exactly as defined in §1 (schedule/VRF among the fixed LSC-node set).
- **What this does not change:** the clock-governance model (§9), bootstrap-eligibility rules (§9), and the small fixed node roster (§1) are all untouched — a MINING_PROOF segment is just one more segment type alongside the others, checked the same way.

12.2 Rejected alternative: full open proof-of-work mining

The other option — replacing LSC-node block-production scheduling entirely with open, permissionless mining (the actual Feathercoin/Monero model, where anyone with hardware can compete to produce the next block) — is explicitly **not adopted**, for reasons worth stating plainly:

- It would eliminate the fixed, known node roster that §9's unanimous CorrectionRFC vote and bootstrap-eligibility system depend on — there is no fixed “everyone” to reach 100% agreement among in an open miner pool. Clock governance would need a completely different mechanism (e.g., stake- or hash-power-weighted voting), which is a different design, not an extension of this one.
- It reintroduces classic PoW attack surface (51%-style reorganization attacks) that the small stable-node model was specifically built to avoid via known identities and fast finality (§1, §10)

rather than hash-power competition.

- It would remove the fast, low-overhead finality this paper's whole design targets, replacing it with the probabilistic finality any open PoW chain has.

If this direction is ever wanted, it should be treated as a fork of LanternSeaCoin into a genuinely different coin, not as an addition to this one.

12.3 Mining Payout, Reward Splitting, and Maturity Hold

Solving a MINING_PROOF puzzle (§12.1) is not free-standing — it has to result in a coin landing on an address, and that needs a clear rule for when more than one contributor is involved. This section adds that rule, plus a maturity hold before a mining reward becomes spendable. **No new contract system is introduced for this** — mining maturity reuses the existing Stamp mechanism from §5, scoped specifically to mining payouts.

1. Payout addresses on the proof itself. MiningProofSegment (§12.1) gains a payout list:

```
MiningProofSegment {
  algo:          enum { NEOSCRYPT_STYLE, RANDOMX_STYLE, EQUIHASH_STYLE }
  nonce:         bytes
  target:        bytes
  proof_hash:    32 bytes
  proof_params:  optional bytes
  payouts:       [ (address, proof_weight) ]    // one or more contributors
}
```

- `proof_weight` is the actual computed difficulty of that contributor's solved proof (how far `proof_hash` fell under target) — **not** a self-declared number and **not** just a count of segments submitted. This matters for safety: if weight were self-reported or based on segment count, a miner could claim an oversized share by submitting many trivially-easy, low-value proofs. Weighting by actual proof difficulty means a contributor's share reflects real computational work.
- **Consensus validity rule (resolves §15.7):** a block is **invalid**, not merely "distrusted," if any payouts entry's declared `proof_weight` does not match an independent recomputation of `BLAKE2b-256(nonce || target || algo)` (or the equivalent check for `EQUIHASH_STYLE`'s native hash) against that entry's `proof_hash` and target. This is not phrased as something a node "should" do (§15.7's exact wording of the earlier gap) — it is a block-acceptance rule on the same footing as checking a signature or a segment's `raw_len`: **any node that skips this check and accepts an invalid block is itself producing a non-conforming implementation**, the same way skipping signature verification would be. Making this an explicit numbered rule here, rather than leaving it as descriptive prose, is the concrete fix.

2. Same block, one miner → whole reward to that address. If only one payouts entry exists for a block's MINING_PROOF segment(s), the full block reward goes to that one address — this is the common case and needs no splitting logic.

3. Same block, multiple miners → proportional split. If several distinct miners each submitted a valid proof toward the same block (e.g., a coordinated pool, or several independent contributors whose proofs both cleared target), the block reward *R* is split proportionally:

$$\text{share}(\text{address}_i) = R * \text{proof_weight}_i / \text{sum}(\text{all proof_weight}_j \text{ in this block})$$

- This directly answers "what if the miners aren't even" — they don't need to be. An uneven contribution simply produces an uneven, but proportionate, split. No voting, negotiation, or off-chain agreement is required; the split is computed the same way by every node from data already in the block.
- If all contributing miners explicitly agree in advance to route the whole reward to one shared address instead (e.g., a mining pool paying out separately off-chain), they can simply submit

a single payouts entry for that shared address — the protocol doesn't need to know or care whether it represents one miner or a pool's internal split.

4. Mined coins are minted as held, not liquid. Each share(address_i) becomes a coin using the existing structure from §5, tagged with its origin and carrying a maturity stamp:

```
Coin {
  amount:    share(address_i)
  owner:     address_i
  movable:   false                // held at mint time – not liquid yet
  origin:    "MINED"
  stamps:    [ MiningMaturityStamp ]
}

MiningMaturityStamp {                // a Stamp (§5), specialized for mining payouts
  stamp_id:   8 bytes
  issued_at:  block_height          // the block this reward was earned in
  hold_seconds: 10800              // 3 hours, expressed in chain_time (§7)
  claim_window: computed, see below
  claimed:    bool
  claim_proof: must prove control of address_i's key // restricts redemption to the earning mi
  chain_index: int64               // OTP-chain position at mint time (§7)
  otp_checkpoint: 32 bytes
}
```

- This is exactly the non-movable, stamp-bearing coin pattern already defined in §5 — mining payouts don't need a separate "hold" state machine, they just use the one that already exists, tagged origin: "MINED" so wallets and explorers can distinguish a mining reward from an ordinary transfer if they want to.
- **Only the earning address can redeem it.** claim_proof must demonstrate control of address_i's private key — since the coin is non-movable anyway (§5), this is a belt-and-suspenders check against a scenario where the coin's record is copied or replayed, not against theft of an already-liquid coin.

5. The 3-hour hold, aligned to 1-hour chain-time checkpoints ("wait till even"). The hold period is measured entirely in **chain-time** (§7's threaded clock, never a node's or client's own clock), and the claim window doesn't simply open exactly 3 hours later — it opens at the next full-hour chain-time boundary at or after that point:

```
raw_unlock_time = chain_time[issued_at] + hold_seconds    // + 3 hours of chain-
time
aligned_unlock  = smallest multiple of 3600 (chain-time seconds)
                  that is >= raw_unlock_time
claim_window.start = aligned_unlock
```

- This is the "clock sync on mining is done on 1-hour shift, wait till even" behavior: rather than every mined coin having its own slightly-different unlock instant, all coins mined within the same rolling hour become claimable at the same shared, round chain-time checkpoint. This makes maturity checks predictable and batchable (any node can compute "which coins just matured" once per hour rather than continuously), and avoids a swarm of near-simultaneous, slightly-offset unlock times that would be harder to reason about safely.
- Everything here is anchored to chain_time from §7 (itself never trusting an external clock after the 10-block bootstrap in §6) — so mining maturity inherits the same "no node or client clock is trusted" property as the rest of the paper, not a separate timing mechanism.

6. After claim, it's an ordinary coin — no special mined-coin spending path. Once MiningMaturityStamp is claimed within its window (using the same claim mechanism already defined in §5), the coin converts to movable: true and behaves exactly like any other coin from that point on. There is no separate "mined coin" transaction type, no coinbase-style special first spend, and

no change to how blocks reference each other (§2's prev_refs linking is completely untouched by any of this) — mining reward accounting is a coin-issuance and maturity detail layered on top of the existing block and coin model, not a restructuring of it. This is the deliberate “don't overtake the whole Bitcoin-like system” boundary: blocks stay linked exactly as defined in §2, and only the coin that emerges from mining gets this specific hold-then-liquid treatment.

12.4 Micro-Proof Aggregation for Low-Power Mining

§12.1's MiningProofSegment was originally sized as one block-level proof. This section shrinks it into many small, cheap **micro-proofs** so that low-power hardware can contribute, and combines them together into the block's mining requirement — turning “one big proof” into “a mined thing together,” directly feeding the proportional-split payout already defined in §12.3.

Smaller, cheaper individual proofs. A MicroProofSegment replaces the single large proof with a lightweight variant:

```
MicroProofSegment {                                // a smaller, low-difficulty MINING_PROOF variant
  algo:          enum { NEOSCRYPT_STYLE, RANDOMX_STYLE, EQUIHASH_STYLE }
  nonce:          bytes                               // shorter than a full block-level proof's nonce
  micro_target:   bytes                               // an individually low, easy-to-reach difficulty
  proof_hash:     16 bytes                            // truncated hash, sufficient at this lower difficulty
  payout:         (address, proof_weight) // this micro-proof's own single contributor
}
```

- micro_target is set far easier than the old single-proof target from §12.1 — low enough that ordinary consumer or low-power hardware (a phone, an old laptop, a small single-board computer) can find a valid solution in a reasonable time, using a scaled-down parameter set of whichever algorithm is chosen (e.g., smaller Equihash (n, k) values, a smaller Neo-Script/RandomX memory footprint) — the same three algorithms from §12.1, just tuned lighter.
- The segment itself is also physically smaller — a shorter nonce and a truncated proof_hash — since a lower-difficulty proof doesn't need the same collision-resistance headroom as a full block-level one. This keeps many of these fitting comfortably inside a block's segment budget (§2) instead of one segment consuming a large share of it.

Aggregation into one block-level result. A single block can now carry **many** MicroProofSegment entries (using the existing segments array from §2 — no new block field needed), and the block's overall mining requirement is satisfied once their combined weight clears a target, rather than by any single proof alone:

```
block_accepted_for_mining ⇐ sum(proof_weight_i for every MicroProofSegment in this block)
                             >= block_target_weight
```

- This is the literal “mined thing together”: no individual contributor needs to reach full block-level difficulty on their own — many small, independently-verifiable micro-proofs add up to the same effective security the single larger proof (§12.1) used to provide alone.
- Each MicroProofSegment's payout entry feeds directly into the same proportional-split formula already defined in §12.3 ($\text{share} = R * \text{proof_weight_i} / \text{sum}(\text{all proof_weight_j})$) — aggregation and payout splitting are the same mechanism now, not two separate systems. A contributor with a small, low-power device simply ends up with a small, proportionate share, exactly as intended.

Why this is still safe. Lowering the individual difficulty bar doesn't lower the block's actual security requirement, because acceptance depends on the *sum* clearing block_target_weight, not on any one proof being easy:

- A flood of many trivially-easy micro-proofs from one actor doesn't inflate that actor's effective share beyond the real computation performed, since proof_weight is still measured as actual

solved difficulty (§12.3) — ten proofs at one-tenth the weight each contribute the same total weight (and the same total reward share) as one proof at full weight, not more.

- Because segments remain capped by the block's overall segment budget (§2), there is a natural ceiling on how many MicroProofSegment entries a single block can hold — this bounds worst-case block bloat from a large number of small contributors without needing a separate, hand-tuned limit.
- Verification cost per micro-proof is cheap by construction (that's the same "low difficulty, low friction" property that makes it accessible to low-power miners in the first place), so a block carrying many of them doesn't meaningfully raise the cost of a node validating that block, compared to validating one larger proof.

12.5 Proof weights and reward distribution

The weight of a proof is its solved difficulty (§12.3). A proof with z leading zero bytes in algorithm a weighs

$$w = \kappa_a \cdot 256^z$$

where κ_a calibrates the cost per attempt between the algorithms. Let W_i be the sum of the weights of miner i 's valid proofs in the block, and $R(h)$ the block reward. The share of miner i MUST be:

$$s_i = \left\lfloor \frac{R(h) W_i}{\sum_j W_j} \right\rfloor, \quad r = R(h) - \sum_i s_i, \quad 0 \leq r < k$$

The remainder r goes to the first contributor; it is at most $k - 1$ base units for k miners. When all proofs of a block share the same algorithm and target, this reduces to counting proofs. This is a consensus rule (§12.3): if a block contains micro-proofs, every node MUST recompute each proof individually and recompute the distribution; if it deviates, the block is invalid. Without a valid proof, the producer keeps the reward. Weights are additive; splitting work across several identities does not change the sum, so there is no Sybil advantage. An invalid micro-proof does not count but does not invalidate the block.

12.6 Verification cost of the three algorithms

Algorithm	Effort per proof	Verification effort	Resource
NeoScript	memory-intensive hash per attempt	one hash	memory, CPU, GPU
Equihash(n,k)	Wagner algorithm over a list of $N = 2^{(n/(k+1)+1)}$ entries	2^k hashes	memory
RandomX	CPU-oriented program execution; official library bound via rffi and ctypes	one hash	CPU, cache

For Equihash(96,5), $N = 131,072$ and verification costs 32 hashes; for (200,9), $N = 2,097,152$ and verification costs 512 hashes. With z leading zero bytes in the target, a valid proof needs 256^z attempts on average. The ratio of forging to verification effort for Equihash is approximately

$$\rho = C_{\text{forge}}/C_{\text{verify}} \approx \frac{k N}{2^k}$$

about $2 \cdot 10^4$ for (96,5) and $4 \cdot 10^4$ for (200,9). NeoScript and RandomX are symmetric ($\rho \approx 1$); their deterrent is the memory and CPU demand per attempt.

12.7 Incentives and limits

A miner with work share ϕ receives $\phi \cdot R(h)$ in a block with proofs, up to rounding. A producer that also mines can omit other miners' proofs, because the rule verifies only the proofs included; miners SHOULD submit proofs through several paths, and slot-proof gossip (§1.2) limits the effect.

13. Dynamic Segment Registry (Development-Time Extensibility)

Up to this point, segment types have been listed as a fixed enum (§2, §12): LIGHT_PASS, HEAVY_PASS, TIME_ANCHOR, BLINK, MINING_PROOF. That fixed-enum approach doesn't scale across releases — every future addition would require every node to already understand it. Instead, segment types become a **versioned, dynamic registry**:

```
SegmentTypeRegistry {
  segment_type_id:  int           // numeric ID, not a fixed enum member
  name:             string        // e.g. "MINING_PROOF"
  introduced_in_version: semver    // e.g. "1.4.0"
  schema_ref:       string        // pointer to this type's payload schema
}
```

```
Segment {
  segment_type_id: int           // references the registry above
  raw_len:         int
  transformed:     bytes
  position_index:  int
}
```

- **New segment types are added by shipping a new registry entry with each release**, not by renegotiating a protocol-wide enum. A future release can introduce, say, a PRIVACY_TAG or AUDIT_LOG segment type without touching anything defined in §2–§12.
- **Older node builds never attempt to parse a segment type introduced after their own version.** If a node running version 1.3.x sees a segment_type_id whose introduced_in_version is 1.4.0, it does not decode, validate, or otherwise act on that segment's transformed bytes — it is explicitly out of scope for that build, not a compatibility target. This is a deliberate constraint, not an oversight: it keeps a running node's behavior fully defined by what it shipped with.
- **This is a development-rollout mechanism, not a permanent multi-version consensus guarantee.** It exists so that new segment types can be trialed and added incrementally during active development, release by release, without redesigning the block format each time. It is explicitly *not* claimed here to solve the harder problem of long-term network-wide backward/forward compatibility across many simultaneously-running node versions in a mature, stable network — that would need its own soft-fork/hard-fork activation process (e.g., requiring all LSC nodes to upgrade before a new segment type becomes accepted network-wide), which is left as future work.
- **Practical effect during development:** each rollout can add exactly one thing (one new segment type) at a time, tested and reviewed in isolation, rather than bundling multiple new block-format changes into a single large migration.

13.1 Soft-fork property

The registry lets nodes skip unknown segment types. A new rule that only restricts validity therefore remains valid for old nodes:

$$V_{\text{new}}(B) \Rightarrow V_{\text{old}}(B)$$

Old nodes accept every block that new nodes accept and stay on the same chain as long as at least $q = 16$ producers apply the new rule. This resolves the compatibility limitation named in §10 for the case of restricting changes.

13.2 Series model

A series s has a feature set F_s with $F_s \subset F_{s+1}$. Its rules apply from the activation height h_s . Between h_s and the sunset $h_s + \Delta_s$ the old and the new format are valid together; afterwards only the new one. This mirrors the coexistence of old and new banknote series until withdrawal:

$$R_s \text{ applies to block } h \iff h \geq h_s, \quad \text{old format valid} \iff h < h_s + \Delta_s$$

Series	Content	Introduction
1 Base	chaining, ECVRF scheduling, fork choice, finality, stamps, mining proofs, difficulty-weighted rewards, redemption, persistence, RPC, block store	genesis
2 Visibility	watermark (segment 7 LINEAGE), thread (segment 8 THREAD), dual-hash serial, slot-proof gossip with fallback rounds, adaptive eligibility threshold, encrypted key files, terminal channel	soft fork via new segment types
3 Seals	per-block endorsement seal and quorum seal (segment 9 SEAL), epoch and expiry height, Merkle snapshots, 8 decimal places	soft fork; constants fixed before mainnet
4 Agility	ParameterRegistry with effective governance, secure-element requirement above a threshold, algorithm change, peer discovery in the node, peer banning	governance decision

13.3 Activation by readiness signalling

Nodes signal readiness in their checkpoint signature. Once at least $q = 16$ signals are present at a window boundary, the rule applies after two further windows:

$$h_s = 60 \left(\lceil h_{\text{sig}} / 60 \rceil + 2 \right) \quad \text{given } \geq q \text{ signals}$$

13.4 ParameterRegistry

The registry maps each parameter name to a value and an activation height. An accepted proposal (§9.2) MUST write an entry with $h_{act} \geq h + 120$; hard parameters require unanimity, soft features require q votes. All nodes read the registry from the checkpoint-signed state and retrieve values through `param(name, height)`. The values of §14 and Annex A are the defaults from height 0.

13.5 Compatibility rules for implementations

1. Series 1 remains valid unchanged; extensions are additive modules and new segment types (lineage, thread, seal, dualhash, proofweight, params, slotgossip).
2. The block hash does not change, because the body root contains new segments through their digests (§2.1). Format changes are versioned by activation height: from h_s , format v2 applies.
3. Domain tags v1 remain; new tags are listed in §14.1.1.
4. Every new function MUST receive a self-test in the style of the existing modules.
5. RPython suitability remains: no new dependencies except `rpython.rlib`.

14. Concrete Parameters and Development Specification

Every earlier section describes *what* each mechanism does; this section pins down *exactly how*, with concrete choices instead of “a hash function” or “an address.” An implementer should be able to build against this section without guessing. §14.8 is honest about what still isn't closed even after this.

14.1 Cryptographic primitives

Purpose	Choice	Why
General-purpose hashing (OTP chain §7, segment integrity §2, Merkle-style structures)	BLAKE2b-256	Same hash family Equihash (§12.1) already uses internally — one hash primitive across the whole design instead of mixing families.
LSC node identity / signatures (block signing §1, CorrectionRFC votes §9, BootstrapTimeAnchor §6)	Ed25519	Small (32-byte public key, 64-byte signature), fast to verify, widely reviewed.
Coin ownership / spending authorization	Ed25519 , same as above	One signature scheme for both node and user keys — no separate curve/library to maintain.
Address derivation	<code>address = BLAKE2b-160(Ed25519_pubkey)</code> , then encoded per §14.2	160-bit output matches common address-length precedent (cf. Bitcoin/Ethereum-style pubkey-hash addressing) while staying inside the BLAKE2b family already chosen above.

Signature domain separation (resolves §15.3): every signed message is prefixed with a fixed, type-specific context string before hashing and signing, so a signature valid in one context can never be replayed as valid in another:

Message type	Domain tag (prepended before signing)
Block signature (§1)	"LSC-BLOCK-SIG-v1"
CorrectionRFC / NodeRevocationRFC vote (§9)	"LSC-GOV-VOTE-v1"
BootstrapTimeAnchor (§6)	"LSC-BOOTSTRAP-v1"
KeyRotation (§9.1)	"LSC-KEY-ROTATE-v1"
Coin-spend authorization (§14.7)	"LSC-COIN-SPEND-v1"
Stamp claim_proof (§5, §12.3)	"LSC-STAMP-CLAIM-v1"
Checkpoint signature (§16.1)	"LSC-CHECKPOINT-v1"

Concretely: $\text{signature} = \text{Ed25519_sign}(\text{privkey}, \text{domain_tag} \parallel \text{message_bytes})$. Reusing one signature scheme (Ed25519) across every message type in the system is only safe *because* every message type now has its own non-overlapping tag — this was the missing piece §15.3 identified, and it is now a required part of every signing/verification routine in this paper, not an optional hardening step.

14.1.1 Additional domain tags

Message type	Domain tag
Redeemable instrument (§18.1)	"LSC-NFC-TOKEN-v1"
Redeem request (§18.3)	"LSC-NFC-REQUEST-v1"
Snapshot (§18.6)	"LSC-NFC-SNAPSHOT-v1"
Key record (§18.5)	"LSC-NFC-KEY-v1"
Secure-element operation (§18.4)	"LSC-NFC-SE-v1"
Coin watermark (§20.2)	"LSC-WM-v1"
Thread accumulator (§20.3)	"LSC-THREAD-v1"
Dual-hash serial (§20.5)	"LSC-SER-v1"
Quorum seal (§20.9)	"LSC-SEAL-v1"
Claim secret (§20.7)	"LSC-CLAIMSECRET-v1"
Terminal channel (§19.4)	"LSC-TERM-v1"
Block endorsement (§1.5.5)	"LSC-ENDORSE-v1"

All tags MUST be pairwise prefix-free.

Lemma 1 (context separation). If the tags are prefix-free and Ed25519 is EUF-CMA secure, a signature over $\text{DOM}_c \parallel m$ is valid for no other context c' , except with negligible probability. *Proof sketch:* Different contexts yield different messages; a valid signature over a never-signed message would be a forgery.

$$\sigma_c(m) = \text{Ed25519.Sign}(sk, \text{DOM}_c \parallel m)$$

14.1.2 Unambiguous encoding Multi-field messages MUST be encoded with length prefixes, not separators:

$$\text{enc}(f_1, \dots, f_k) = \parallel_{i=1}^k (|f_i| \parallel f_i)$$

Lemma 2. enc is injective; field boundaries are unambiguous, and confusions such as $(a|b, c)$ versus $(a, b|c)$ are excluded.

14.1.3 Primitive inventory

Purpose	Primitive	Standard
Block, body, request hash	BLAKE2b-256	RFC 7693
Wallet, address checksum, PBKDF2	SHA-256, SHA-512, HMAC	FIPS 180-4, RFC 2104
Signatures	Ed25519	RFC 8032
Slot lottery	ECVRF-EDWARDS25519-SHA512-TAI	RFC 9381
LIGHT_PASS, storage encryption, terminal channel	ChaCha20	RFC 8439
HEAVY_PASS	Argon2id (64 MiB, 3 passes, parallelism 1)	RFC 9106
Seed and key derivation	BIP-39, SLIP-0010	BIP-39, SLIP-0010
Addresses	Bech32	BIP-173
Proof of work	NeoScript, Equihash(n,k), RandomX	reference algorithms

The security target is about 128 bits. Implementations **MUST** be checked against the test vectors of the cited standards and **SHOULD** be hardened against side channels.

14.2 Address format

```
raw_address = version_byte(1) || pubkey_hash(20 bytes) || checksum(4 bytes)
checksum    = BLAKE2b-256(version_byte || pubkey_hash)[0:4]
text_address = Bech32("lsc", raw_address) // BIP-173, human-readable part "lsc", e.g. lsc1q9
```

- `version_byte` = 0x00 for ordinary movable-coin addresses; 0x01 reserved for future address types introduced via the dynamic segment/format registry (§13's versioning philosophy applied here too).
- The 4-byte checksum is purely for typo detection — it is not a security mechanism. Bech32 (BIP-173) additionally protects the text form with its own BCH checksum.

14.3 Network-level constants

Parameter	Value	Source / rationale
Number of LSC nodes	24	Midpoint of the "12–48" range given in §1; concrete for implementation, tunable later only via the same CorrectionRFC unanimous-vote process (§9) extended to cover node-count changes.
Target block interval	60 seconds exactly	Matches Feathercoin's real ~1-minute cadence (§0.1), already used informally throughout; now fixed as a literal constant <code>expected_block_interval = 60</code> .
Max block size (all segments + header + transactions)	2 MiB	Concrete cap needed for §2's segment budget and §4's blink-subsystem percentage to mean anything precise.

Parameter	Value	Source / rationale
Blink subsystem allocation	25% of max block size (i.e., up to 512 KiB for BLINK segments)	Fixed midpoint of the "20–30%" range from §4.
Max MicroProofSegment count per block	500	Concrete instance of the "natural ceiling" described qualitatively in §12.4.
Max HEAVY_PASS segment count per block	32	Resolves §15.5: HEAVY_PASS (§3) now runs Argon2id, which is deliberately expensive — an explicit hard cap bounds the worst-case validation cost of any single block regardless of how it's filled, rather than leaving this as an open-ended DoS surface.
Argon2id cost parameters for HEAVY_PASS	memory: 64 MiB, iterations: 3, parallelism: 1 (per invocation)	Concrete tuning: expensive enough to give HEAVY_PASS real computational weight, bounded enough (combined with the 32-segment cap above) that a full block's worth of HEAVY_PASS segments has a known, calculable maximum validation cost.
drift_tolerance default	±2 seconds of chain-time	Concrete instance of the "small tolerance band" from §7, tunable only via CorrectionRFC.

14.4 Genesis and bootstrap constants

- `genesis_seed` (§7's OTP chain root) is a fixed, publicly documented 32-byte value chosen at genesis and published alongside the node software release — e.g., the BLAKE2b-256 hash of a specific, dated, publicly-checkable genesis text (analogous to Bitcoin's genesis-block coinbase message), so anyone can independently verify it wasn't chosen adversarially after the fact.
- The 10 `BootstrapTimeAnchor` producers (§6) are the first 10 LSC nodes in the initial, publicly published node roster — not chosen dynamically, so there's no ambiguity about who bootstraps a fresh chain or a future restart.
- Tie-breaking rule for the bootstrap median (§6): with 10 samples, the median is defined as the average of the 5th and 6th ordered values, truncated to the nearest whole second.

14.5 Reward schedule

Parameter	Value	Note
Initial block reward	50 LSC	Placeholder round number; real value would need economic modeling outside this paper's scope.

Parameter	Value	Note
Halving interval	every 2,100,000 blocks	Deliberately mirrors Feathercoin's actual halving interval (§0.1) exactly, given the shared "lightweight, fast-block" lineage.
Max supply	Reward × geometric halving series, about 210,000,000 LSC	Derived in §14.5.1 from the two values above.
Micro-proof aggregation reward	Same block reward, split per §12.3/§12.4's proof-weight formula	No separate reward pool for micro-proofs — they draw from the same per-block reward as a single large proof would have.

14.5.1 Base unit and issuance table One LSC consists of 10^8 base units. The reward halves by right shift every $H = 2,100,000$ blocks (about 3.99 years); the initial value is $R_0 = 50$ LSC = $5 \cdot 10^9$ base units:

$$R(h) = \lfloor R_0 / 2^{\lfloor h/H \rfloor} \rfloor, \quad S = H \sum_{k \geq 0} \lfloor R_0 / 2^k \rfloor \approx 2 R_0 H = 210,000,000 \text{ LSC}$$

The rounding deviation is at most 0.7 LSC. New supply per epoch:

Epoch	Reward in LSC	Created, million LSC	Cumulative
0	50	105.0	50.0 %
1	25	52.5	75.0 %
2	12.5	26.25	87.5 %
3	6.25	13.125	93.75 %
4	3.125	6.5625	96.875 %
5	1.5625	3.28125	98.4375 %

14.6 Wire format

- **Endianness:** little-endian for all fixed-width integer fields, network-wide.
- **Serialization:** fixed-width binary struct layout (as shown in every Segment/Coin/Stamp definition throughout this paper) rather than a self-describing format like JSON or Protobuf — this keeps parsing simple and deterministic, which matters for independent hash/signature verification across nodes.
- **Segment length field:** `raw_len` (§2) is a uint32; `position_index` is a uint16 (supporting up to 65,536 segments per block, well above the 500-MicroProofSegment cap in §14.3).
- **Versioning:** governed by the dynamic segment registry (§13) and the series model (§13.2): each segment type carries its `introduced_in_version`, and series activate by height.

14.7 Transaction format (ordinary transfers)

Not fully specified elsewhere in this paper — added here for completeness:

```
Transaction {
  inputs:    [ CoinReference ]    // coins being spent (must be movable: true, §5)
  outputs:   [ (address, amount) ]
  fee:       uint64                // paid to the block-producing LSC node
  signature: Ed25519 signature over (inputs || outputs || fee)
```

```

}

CoinReference {
  coin_id:    32 bytes    // BLAKE2b-256 of the coin's minting record
  owner_sig:  Ed25519 signature proving ownership
}

```

- Fees go to the LSC node that produced the block (schedule per §1), separate from the mining-related reward mechanics of §12 — an ordinary transaction fee is not part of MiningProof-Segment/MicroProofSegment payout splitting.

14.7.1 Wallet and Key Custody Model

§15.8 flagged that nothing specified how a user generates, stores, or backs up the Ed25519 keys behind an address (§14.2). This adopts established, widely-deployed standards rather than inventing a new scheme:

- **Seed generation:** a 24-word mnemonic phrase, generated and validated per the **BIP-39** wordlist/checksum standard (the same scheme used by Bitcoin, Ethereum, and most modern wallets) — chosen specifically because it's already familiar to anyone who has used a main-stream crypto wallet, and because its checksum catches most transcription errors when a user restores a wallet.
- **Key derivation:** individual Ed25519 keypairs are derived from that seed using **SLIP-0010** hierarchical deterministic derivation (the Ed25519-compatible counterpart to Bitcoin's BIP-32), following a fixed derivation path (e.g., `m/44'/<LSC coin type>'/0'/0/i` for the *i*-th address) — so a single 24-word backup can regenerate every address a user has ever used, rather than needing a separate backup per key.
- **Documented property, not an assumption:** losing the 24-word seed means permanent, irrecoverable loss of every coin controlled by keys derived from it — this is the same property essentially all non-custodial coin systems have, but §15.8 correctly pointed out it was never actually stated in this paper. It is now a documented, explicit property of the wallet model, not an implicit assumption.
- **Why established standards instead of a custom scheme:** BIP-39 and SLIP-0010 are both widely deployed and independently reviewed by the broader cryptocurrency ecosystem — adopting them here means LanternSeaCoin's wallet security inherits years of real-world scrutiny instead of introducing a new, unreviewed key-derivation scheme on top of everything else in this paper that already needed review.

14.7.2 Multi-input and multi-output transactions A transaction MAY have several inputs and outputs. The sum of the input amounts MUST equal the sum of the output amounts plus the fee; the signature of each input binds the entire output set, which prevents later redirection of change. Single-output transactions MUST keep the amount equal to the input coin's amount. An optional expiry height `h_exp` (§20.8) bounds the lifetime of a signed transaction in the mempool; it MUST NOT be bound to a block hash, because such a transaction would be invalid on a competing branch and could not return to the mempool after a reorganization (§1.4).

14.8 What concreteness does and does not fix

Pinning down the numbers and formats above removes *ambiguity* — two teams implementing against this section should now produce interoperable code. It does **not**:

- Substitute for a **cryptanalytic review** of the LIGHT_PASS/HEAVY_PASS transformation tiers (§3, §10) — BLAKE2b and Ed25519 are well-studied, but the specific pass-through pipeline built from them here is new and unreviewed.
- Validate the **VRF construction**: ECVRF per RFC 9381 is adopted (§1, §15.6); its integration into the scheduling pipeline still needs independent review.

- Provide a **complete formal safety proof**: fork choice is covered by Theorem 2 (§1.3) and finality by §16.5, but the liveness properties of the unanimous CorrectionRFC governance model (§9, §10) remain named risks, not proven-safe mechanisms.
- Replace **real-world testing**: network simulation under adversarial conditions, a genuine multi-node testnet, and independent security audit are all still required before any of this should hold real value, regardless of how precisely specified it now reads.

In short: this section makes the paper *buildable without guessing*, not *audited and safe to deploy*. Those are different bars, and only the first one is addressed here.

15. Security Considerations

Earlier sections named individual trade-offs where each mechanism was introduced (most densely in §10). This section originally collected the security-relevant gaps in one place; each has since been addressed with a concrete fix elsewhere in the paper, cited below rather than repeated. §15.9 covers the one category — empirical validation — that no amount of specification can close by itself, and gives it a concrete roadmap instead of leaving it as an open statement.

15.1 Unreviewed cryptographic constructions — Fixed, see §3

The custom “chained modular-arithmetic” transform is replaced by two established, independently reviewed primitives: **ChaCha20** (RFC 8439) for LIGHT_PASS and **Argon2id** (RFC 9106) for HEAVY_PASS. §3 also names the residual risk honestly: the primitives are proven, but the specific way they're assembled into this segment pipeline is still new and should get its own focused review before being relied on.

15.2 Node identity, compromise, and revocation — Fixed, see §9.1

§9.1 adds routine KeyRotation (self-service, no vote needed) and a NodeRevocationRFC process for compromised or unresponsive nodes. Critically, revocation quorum is **unanimity among every node except the one being removed** — deliberately different from CorrectionRFC's all-inclusive unanimity, specifically so a compromised node can't block its own removal. §9.1 is honest that the vote *mechanism* is now safe, but the real-world judgment of *whether* a node is actually compromised still depends on competent node operators — that part isn't something a spec can fully solve.

15.3 Signature domain separation — Fixed, see §14.1

Every signed message type (block signatures, governance votes, bootstrap anchors, key rotations, coin spends, stamp claims) now gets its own fixed domain-separation tag prepended before signing, so a signature valid in one context can never be replayed as valid in another.

15.4 Blink subsystem as a reconnaissance surface — Fixed, see §4

peer_hint is downgraded from a specific route to a coarse routing class, and raw per-probe records are aggregated into rolling statistical summaries (min/median/max latency, coarse route-class distribution) before publication, rather than broadcasting per-node, per-route data in real time. The subsystem still detects dead nodes and degraded routes — it no longer doubles as a live targeting map for a small, already-identified node set.

15.5 Computational denial-of-service via HEAVY_PASS — Fixed, see §14.3

An explicit hard cap of 32 HEAVY_PASS segments per block, plus fixed Argon2id cost parameters (64 MiB / 3 iterations / parallelism 1), bounds the worst-case validation cost of any single block to a known, calculable maximum.

15.6 Unspecified VRF and scheduling predictability — Fixed, see §1

Block-production scheduling now uses a named, concrete construction — **ECVRF-EDWARDS25519-SHA512-TAI per RFC 9381** — rather than an unspecified “round-robin or VRF-weighted” description. No outside party can predict which node produces an upcoming block without that node's private key, while the selection remains publicly verifiable after the fact.

15.7 Enforcement gap in mining proof verification — Fixed, see §12.3

Proof-weight verification is now stated as a hard **consensus validity rule** — a block with a `proof_weight` that doesn't match independent recomputation is invalid outright, on the same footing as a bad signature — rather than a “should” recommendation left to individual implementations.

15.8 No wallet or key-custody model — Fixed, see §14.7.1

Adopts **BIP-39** mnemonic seed phrases and **SLIP-0010** hierarchical key derivation — both widely deployed, independently reviewed standards already used across the broader cryptocurrency ecosystem — rather than a custom scheme. Key loss as permanent, irrecoverable fund loss is now an explicitly documented property, not an unstated assumption.

15.9 Empirical validation — still open, now with a defined path instead of an open statement

No amount of specification substitutes for actually testing the system, so this gap is addressed with a required roadmap rather than a fix:

Phase	Required before proceeding
1. Internal testnet	All 24 LSC-node roles run by the development team; adversarial test cases for every rule newly hardened in §15.1–§15.7 (e.g., deliberately submit an invalid <code>proof_weight</code> and confirm the block is rejected per §12.3's new rule).
2. Independent cryptographic review	External review specifically of the §3 pass-through pipeline assembly (§15.1's residual note) and the ECVRF/domain-separation integration (§1, §14.1) — not just confirmation that ChaCha20/Argon2id/Ed25519 themselves are sound, which is already established, but that <i>this design</i> uses them correctly.
3. Public testnet with external node operators	Real, independently-operated nodes exercising §9.1's revocation process and §9's unanimous CorrectionRFC under actual (not simulated) node behavior, including at least one deliberate unresponsive-node drill.

Phase	Required before proceeding
4. Formal audit of economic parameters	Independent review of §14.5's reward schedule and §12.3/§12.4's payout-splitting math under adversarial mining scenarios, before any real value is at stake.
5. Mainnet	Only after phases 1-4 are complete and their findings are incorporated — not on a fixed calendar date.

This doesn't close the gap today — it replaces “no testing has happened” with a concrete, ordered commitment for what must happen before it's reasonable to trust this system with real value.

15.10 Overall verdict

Every previously-identified design gap (§15.1-§15.8) now has a concrete fix, cited above, using established and independently reviewed building blocks (ChaCha20, Argon2id, Ed25519 with domain separation, RFC 9381 ECVRF, BIP-39/SLIP-0010) rather than custom or unspecified mechanisms. This meaningfully improves the paper's security posture over the previous revision. It does **not** mean the system is safe to deploy today: §15.9's roadmap has not been executed, the residual assembly-level review noted in §15.1 hasn't happened, and no real adversarial conditions have tested any of this. The honest state is: **the known gaps are now closed on paper; the unknown ones can only be found by actually running phases 1-4 of §15.9.**

15.11 Layered attack analysis

Every attacker class (§0.2.3) meets at least two independent layers; the remaining limits lie with key material and cohort collusion.

Attack	Layer 1	Layer 2	Layer 3	Limit
T1 Token forger	signature and domain tag (§14.1)	key registry, revocation (§18.5)	dual-hash serial (§20.5)	theft of the issuer key
T2 Replay	request_id idempotency (§18.3)	network id and claim window (§18.2)	epoch and expiry binding (§20.8)	transactions without expiry height
T3 Clone	counter duplicate (§18.4)	one-time tag (§20.10)	credit limit (§20.6)	loss up to $c \cdot (U_{\max} - u_0)$
T4 Partition	fork choice (§1.3)	locator sync (§17.3)	finality and seals (§16.5, §1.5.5)	eclipse of a node without honest peers
T5 Time manipulation	median and drift cap (§6.1, §7.1)	OTP chain (§7.2)	thread (§20.3)	anchor collusion at start
T6 Miner collusion	per-proof verification (§12.5)	consensus rule §12.3	proof-of-cost (§20.4)	ensorship by producers (§12.7)
T7 Storage, keys, channel	encrypt-then-MAC (§20.13)	checksums and repair (§17.5), channel counter (§19.4)	none	node key file without passphrase
T8 Node misuse	quorum 16 of 24 (§16.5)	revocation N-1 (§9.1)	seal levels (§20.9)	collusion of at least 16 nodes

Formal statements. Lemma 1 (context separation, §14.1.1), Lemma 2 (injective encoding, §14.1.2), Lemma 3 (dual hash, §20.5), Theorem 1 (chain integrity, §2.1), Theorem 2 (independence of arrival order, §1.3), Theorem 3 (conservation of supply, §5.5), Lemma 4 (seal exclusivity, §1.5.5), the fork bound with slot-proof gossip (§1.5.4), the safety threshold $f \leq 7$ of finality (§16.5) and the loss bound for clones (§20.6). They are given as proof sketches.

Security assumptions and limits.

1. Collusion of 16 or more nodes can finalize arbitrary history; there is no external verification authority.
2. Key material of nodes and issuers **MUST** be stored encrypted (§20.13); the security of the secure element and of card readers is an assumption (H7).
3. Primitives **MUST** be checked against test vectors and **SHOULD** be audited and side-channel hardened.
4. Overload protection: peer scoring and banning, mempool limit 5000, at most 500 hashes per inv.
5. Producers can omit other miners' proofs (§12.7).
6. The independence assumptions in §20.9 and §20.15 are optimistic, because shared software and operators correlate failures.
7. A producer that creates two different blocks for the same slot (equivocation) is handled by fork choice (hash tie-break); the cohort **MAY** decide on revocation with proof of two blocks for the same slot.

16. Automatic Checkpointing and Chain Archival

§10 previously named this as an open addition: Feathercoin's real **Automatic Checkpointing (ACP)** (§0.1) was cited as relevant precedent but never built into the design. This adds two mechanisms that sit at very different levels: **checkpointing (§16.1) is a mandatory consensus rule every node must follow. Archiving (§16.2) is an entirely optional, off-protocol local convenience that no node is required to implement, run, or even support** — a node that never archives anything is exactly as valid as one that does. The two are related in spirit (both are about chain history) but decoupled in practice, and §16.2 states that boundary explicitly rather than leaving it ambiguous.

16.1 Automatic Checkpointing

```
CheckpointRecord {
  checkpoint_height: block_height
  block_hash:      32 bytes          // BLAKE2b-256 commitment to the block at this height
  signatures:      [ (lsc_node_id, signature) ] // domain-tagged per §14.1's "LSC-
CHECKPOINT-v1"
  quorum_required: 16                // 2/3 of 24 LSC nodes – see rationale below
}
```

- A checkpoint is proposed and signed every **60 blocks (~1 hour of chain-time)** — deliberately aligned with the existing 1-hour maturity-checkpoint rhythm already used for mining-reward maturity (§12.3), so the network already has a natural once-per-hour rhythm to hang this on.
- **Quorum is 2/3 of LSC nodes (16 of 24) — deliberately lower than §9's unanimous CorrectionRFC and different from §9.1's "all-except-target" revocation quorum.** Checkpointing is a routine, frequent operation; requiring unanimity for something that happens every hour would make the whole mechanism fragile to a single slow or offline node, unlike the rare, maximally-consequential clock-formula changes §9 was designed to gate.
- **Once a height is checkpointed, it is final.** No prev_refs-based reorganization (§2) may ever reference a chain history that would rewrite anything at or below checkpoint_height.

This is the direct fix for the gap named in §10 and §0.1: it gives the loosely-braided chaining model a hard floor, the same way Feathercoin's real ACP protects against reorganization, adapted to this design's small fixed node set (2/3-of-24 sign-off) instead of Feathercoin's own open-network mechanism.

- **This is the "hard split point"** referenced below: each checkpoint height is a fixed dividing line between "permanently finalized history" and "still-recent, still-technically-adjustable history" above it.

16.2 Archive Export at Hard Split Points (Optional, Off-Protocol — Not a Node Requirement)

This is explicitly not a protocol rule, and not a property of what makes a node "full."

§16.1's checkpointing is a consensus rule every node must follow to validate correctly. Archiving is a separate, entirely optional, local convenience that the protocol neither requires nor is aware of — it exists purely for an operator's own benefit and imposes no obligation on anyone else's node.

A node that never archives anything, ever, is a completely valid, fully-conformant node indefinitely. It follows §16.1's checkpoint rule (mandatory, consensus-level) exactly the same as any other node, and simply never runs an export routine at all. Nothing in this paper's validity rules, block acceptance logic, or peer behavior depends in any way on whether a given node has ever produced an archive — other nodes have no way to know, and no need to know, whether a peer archives. The two mechanisms are deliberately decoupled: checkpointing exists so *the network* has hard finality; archiving exists so *an individual operator, if they want it*, has a complete, browsable copy of history somewhere. Neither requires the other.

With that boundary stated plainly, here is one reference design for operators who *do* want this — entirely optional to adopt, and the protocol dictates none of it. **One thing is deliberate, though: whether to archive is optional, but what an archive is structured around is not arbitrary — every archive interval is anchored to a real, on-chain CheckpointRecord (§16.1), never to a private, disconnected snapshot.** That's what lets an archive stay independently verifiable, possibly years later and entirely offline: anyone holding an archive can check its content_hash against the actual on-chain checkpoint it claims to correspond to. An archive not tied to a real checkpoint would just be an unverifiable local file; anchoring it to the chain is what makes an optional backup still trustworthy.

```
ArchiveManifest {                                // purely local/off-protocol bookkeeping — never
                                                // transmitted, never validated by peers, never
                                                // required for consensus
    archive_id:      id
    covers_heights:  block_height_range        // e.g. blocks 1–60, the interval since the last checkp
    checkpoint_ref:  checkpoint_height         // anchors this archive to a real, on-
chain §16.1 CheckpointRecord —
                                                // never a private or disconnected snapshot, see above
    content_hash:    32 bytes                  // BLAKE2b-
256 over the full archive file's contents
    segment_types_included: [ ALL ]           // this reference design always includes every segmen
    destination:     path_or_device           // entirely operator-chosen, see below
}
```

- **The protocol does not dictate cadence, format, destination, or whether to run this at all.** Everything below is one possible shape an operator could choose to implement locally — not a requirement this paper places on any node, and not something peers can observe or enforce.
- *If* an operator chooses to run it: contents can be complete and unpruned, explicitly including every mining segment type — full MiningProofSegment and MicroProofSegment records (nonce, proof_hash, proof_params, and every payout entry), rather than the summarized/committed form a hot-storage pruning policy (§16.3) might otherwise use. This is what

"archive... with all mining segments" means in practice, for an operator who opts into it — it is not something the network requires of them.

- *If* run, an archive interval would also naturally include every other segment type from that span (LIGHT_PASS, HEAVY_PASS, TIME_ANCHOR, BLINK) plus block headers.
- **Destination is 100% operator discretion** — a local path, a separate archive-only disk, or a removable/external hard drive — the protocol has no opinion and no way to check.
- *If* run automatically, it could reasonably trigger each time a new checkpoint forms (matching "backup routine on hard split points") — but manual, scheduled, partial, or never-run are all equally valid choices a node can make without becoming non-conformant.
- **A human-browsable index** (block height, chain-time, block hash, per-block mining-payout summary) is a reasonable addition for an operator who wants to "watch through" an archive directly — again, purely a local design choice, not a protocol feature.
- **Integrity, if desired:** an operator who does keep content_hash values can cross-check them against the corresponding on-chain CheckpointRecord (§16.1) later, entirely offline — but this is a convenience for the operator's own peace of mind, not something the network verifies or depends on.

16.3 Relationship to hot-storage pruning

This is deliberately a **separate concern** from any storage-reduction policy a node's live/hot storage might apply — and, like archiving, pruning policy is also entirely local and off-protocol:

- A node can run lean day-to-day by pruning old mining-segment raw bytes from its active storage once they're both checkpointed (§16.1, mandatory consensus fact) and past their maturity window (§12.3) — replacing them locally with a compact commitment, if the operator wants to.
- Independently, the same node (or a different, archive-dedicated machine) may — but is never required to — also retain a full, complete archive as sketched in §16.2 on separate storage.
- **All three of these are independent, optional operator choices**, layered on top of the one mandatory consensus fact (§16.1's checkpoints): prune hot storage or don't, archive or don't, run both on one machine or split them across two. None of these choices affects whether a node correctly validates the chain — that's governed entirely by §16.1, which every node must follow regardless of its storage or backup habits.

16.4 Trade-offs and residual notes

- **Choosing to keep full archives forever reintroduces the unbounded storage growth already discussed** — this is an accepted, deliberate cost of wanting complete mining-segment history preserved and browsable, not an oversight. The per-interval archive size is bounded (one checkpoint interval's worth of blocks), but the total across all archives kept still grows without limit over the life of the chain, same as before.
- **2/3 checkpoint quorum is a lower trust bar than §9's unanimous governance vote**, by design — but this means a colluding 16-of-24 supermajority *could* checkpoint a fraudulent history if that many nodes were compromised or acting together. This is the same class of small-node-cohort risk already named generally in §10, restated here specifically for checkpointing rather than left implicit.
- **The archive export routine itself is new, untested node software, for operators who choose to run it.** It falls under the same §15.9 empirical-validation roadmap (internal testnet, external review, public testnet, audit) as everything else in this paper — but since running it is entirely optional (§16.2), this risk is scoped only to the operators who opt in, not to the network's validity rules, which depend solely on §16.1's mandatory checkpointing.

16.5 Finality analysis

Every 60 blocks (about 1 hour) the nodes **MUST** sign (h, B_h) with the tag LSC-CHECKPOINT-v1. With $q = 16$ valid signatures out of $N = 24$ the block is final. A finalized block **MUST NEVER** be

replaced; branches that bypass it **MUST** be rejected, and a checkpoint on a side branch forces the switch to that branch. Finalized checkpoints **MUST** be stored persistently and survive restarts.

Two conflicting checkpoints of the same height need quorums Q_1 and Q_2 of at least q members each. Their intersection contains at least $2q - N = 8$ nodes that signed twice (provable equivocation):

$$|Q_1 \cap Q_2| \geq 2q - N = 8 > f \implies f \leq 7$$

Liveness requires $N - f \geq q$, hence $f \leq 8$. This matches the classical threshold $N \geq 3f + 1$ (Castro and Liskov) with $f = 7$. For other cohort sizes: $q = \lceil 2N/3 \rceil$ and $f_{\text{max}} = \lfloor (N-1)/3 \rfloor$.

16.6 Checkpoint contents from Series 2

From Series 2 (§13.2) a checkpoint **MUST** sign the tuple (h, B_h, R_h, o_h) , where R_h is the MMR root of §20.3 and o_h the OTP value, and **MAY** carry readiness signals for activation (§13.3). The archive manifest of §16.2 covers the height range, segment types and checksums.

17. Network, Synchronization, Persistence, Node Process, RPC

A node looks for peers locally first, synchronizes with a logarithmically sized locator, and survives restarts through a checksum-protected block store.

17.1 Peer discovery

Five methods are defined: **connect** (Ed25519 key as identity), **local** (SSDP-, mDNS- and WS-Discovery-style over UDP multicast), **mesh** (Freifunk- and batman-adv-style originator messages with multi-hop routing, quality degradation and replay protection), **peering** (GUnet-style beacons followed by a TCP handshake) and **internet** (DNS seeds and a Kademlia DHT). The manager **MUST** work local-first. NAT traversal, WebRTC, STUN and TURN are excluded. The node process **SHOULD** integrate the manager and additionally accept explicit `--connect` addresses.

17.2 Messages

Framing is Bitcoin-style (magic, command, length, BLAKE2b checksum) with a version/verack handshake:

Message	Purpose
getblocks	send a locator, receive the hashes after the fork point
inv	announce block or transaction hashes (at most 500)
getdata	request unknown hashes
blockrec	complete block with operations and segments
tx	operation for the mempool
mempool	request pending operations
slotproof	ECVRF proof of a slot (from Series 2, §1.2)
endorse	block endorsement signature (from Series 3, §1.5.5)
ping, pong, notfound, bye	control

17.3 Locator

The locator contains the last 10 hashes without gaps, then hashes at doubling distances, at most 64 entries. The server **MUST** find the most recent locator entry on its canonical chain and reply with the hashes after it. The resync therefore also works when the client sits on a discarded branch.

17.4 Sync and relay

Blocks without a known predecessor wait in the orphan pool, and the node requests the missing predecessors with `getdata`; a block that is already waiting **MUST NOT** be requested again. With full `inv` lists the sync proceeds in several rounds. New tips and transactions are relayed to all peers except the source.

17.5 Block store

An append-only log stores all accepted blocks, including those on side branches, and all finalized checkpoints:

$$\text{entry} = \text{LSB1} \parallel \text{type}[1] \parallel \text{len}[4] \parallel \text{payload} \parallel H(\text{type} \parallel \text{len} \parallel \text{payload})[:16]$$

On load the store distinguishes OK, TORN_TAIL (the last entry is incomplete; only it is lost) and CORRUPT (damage in the middle; the file is truncated there and the node fetches the rest by sync). Every block read **MUST** pass the same checks as a new block.

17.6 Node process

A node connects store, ChainState, listener, relay, production and RPC (`--datadir`, `--cohort`, `--key`, `--listen`, `--connect`, `--rpc-port`, `--produce-interval`, `--run-seconds`). A lock **MUST** protect the ChainState. A node with a key computes its proof for the next height, builds a block when eligible and announces it with `inv`. The key file **MUST** be stored encrypted (§20.13). A node **SHOULD** score peers and ban misbehaving ones, and **MUST** bound the mempool size (5000) and message sizes.

17.7 RPC

An HTTP interface in the style of bitcoind (JSON-RPC, basic authentication) offers: `getinfo`, `getblockchaininfo`, `getblockcount`, `getbestblockhash`, `getblockhash`, `getblock`, `getchaintips`, `getmempoolinfo`, `getrawmempool`, `getbalance`, `getwalletinfo`, `listunspent`, `listtransactions`, `getnewaddress`, `validateaddress`, `sendtoaddress`, `getpeerinfo`, `getconnectioncount`, `getnetworkinfo`, `getmininginfo`, `getdifficulty` (constant 1), `help` and `stop`. `sendtoaddress` **MUST** place a payment into the mempool and **MUST NEVER** modify the ledger directly.

18. Redemption: Offline, Online, Snapshots

A stamp, or an instrument that refers to a stamp, is redeemed in two phases: offline redemption creates and persists a locally verifiable claim request, and online redemption reconciles and settles it exactly once against the canonical ledger. Offline redemption alone is not final; final settlement requires reconciliation. This section defines the common protocol that every physical carrier uses. The formats of the carriers themselves are outside the scope of this paper.

18.1 Model

A redeemable instrument (a token, a voucher or a card-held right) refers to a stamp (§5) and names its beneficiary. The issuer is the owner of the coin that carries the stamp and signs the instrument with `LSC-NFC-TOKEN-v1` over these fields: version, network, type, instrument and stamp identifiers, issuer, beneficiary, amount, issue height, claim window (start, end), chain index, OTP value, snapshot height and snapshot hash, and flags. Payout goes to the beneficiary address; the flag `FLAG_BEARER` makes the presenter the recipient instead. Online redemption consists of a claim and a transfer and **MUST** be atomic: if a step fails, a rollback restores the ledger exactly.

18.2 Claim window and chain context

Each instrument carries a claim window [start, end] in blocks (reference value 1440 blocks), the chain index and the OTP value of its issue time (§7.2), and a reference to a snapshot (§18.6). A redemption before start **MUST** be rejected as not yet valid, and one after end as expired. At redemption the verifier compares the locally estimated chain context (index, chain time, OTP value) with the canonical one (§8.1). If the deviation exceeds the drift tolerance of 5 blocks, the request stays pending until reconciliation.

18.3 Redemption

A redeem request is identified by

$$\text{request_id} = H(\text{enc}(\text{net}, \text{token}, \text{stamp}, \text{redeemer}, \text{nonce}))$$

which does not depend on the wall clock and **MUST** be unique.

Offline redemption. The device creates the request without network access and **MUST** store it before it reports success; if storing fails, it **MUST** report failure. The request is locally verifiable but not final.

Online redemption. Settlement is idempotent over the request_id: an identical request returns the identical result, and the same request_id with a different payload is rejected as a replay. The service reconciles the request with the issued instrument (amount, stamp, recipient); a redirected recipient is rejected. A request whose context deviates beyond the drift tolerance stays PENDING. A fee is paid through a second output of the multi-output transfer to a collector (§14.7.2).

18.4 Secure-element profile

The element holds a device-bound key and a monotonic counter c ; the request nonce is $\text{se} : c$. The offline credit u is counted in the element and **MUST NOT** exceed $U_{\text{max}} = 500$; a single instrument is limited to 100, and above an amount of 50 a secure element is mandatory. The credit is reset only with a server signature over (device, counter). A one-time tag permits exactly one write and is burned after redemption. Two different requests with the same counter are rejected as a replay and mark the device as cloned.

18.5 Keys

A rotation **MUST** be signed by both the old and the new key. A revocation targets an instrument or a key. A key reported as compromised invalidates all of its instruments, a superseded key only those issued later. A key **MAY** revoke only itself and older keys of the same chain. Distribution happens by merging (monotonic, idempotent, independent of order).

18.6 Snapshots

A snapshot is a signed description of the ledger state at a height; it gives offline verifiers a compact reference but does not replace consensus or finality. It binds height, block hash, chain time, OTP value, the hash of the predecessor snapshot, state_root, stamp_root, coin_root and total_supply. A snapshot is produced every 60 blocks, at the checkpoint interval, so every snapshot coincides with a checkpoint (§16.5). It is final from the quorum $[2n/3] = 16$ signatures. Height regression and predecessor hash **MUST** be checked. From Series 3, coin_root is the root of a Merkle tree (§20.7).

19. Terminal and Air-Gap Layer

LSC works with certified payment terminals and keeps card data out of its own software: terminals read the carriers, and only transaction metadata reaches an air-gapped device that holds the wallet.

19.1 Basic rules

- **R1.** Card payment data (PAN, track data, PIN, EMV cryptograms) **MUST** remain inside the certified reader or terminal. It **MUST NOT** enter LSC wallets, block stores, logs or transactions.
- **R2.** The air-gapped device receives transaction metadata only (§19.4).
- **R3.** Real card payments require a contract with a payment processor, certified terminals and PCI compliance. Development and tests **MUST** use test cards and the terminal's test mode.
- **R4.** For girocard payments in Germany, a network operator certified by the Deutsche Kreditwirtschaft **MUST** be used (example from the market research: Computop); the girocard requires PIN entry.

19.2 Terminal reference classes

The data stem from market research in October 2025 and rely on manufacturer information; they **MUST** be verified before deployment.

Class	Example	Cards	Connection	Notes
Dongle reader	MagTek eDynamo	magnetic stripe, EMV contact	micro-USB, optional Bluetooth LE	MagneSafe, 3DES, DUKPT, PCI certified; no NFC, no PIN pad; SDKs for Android, iOS, Windows
Kiosk PED	MagTek DynaFlex II PED	magnetic stripe, EMV contact, contactless, optional barcode	USB, variant with WLAN	PCI PTS 6.x, AES-256 or 3DES, DUKPT, PIN-on-glass
Android POS	PAX A920 with dock L920P	EMV, NFC, magnetic stripe	micro-USB charging port, dock with USB host, RS232, Ethernet	PCI PTS 5.x, printer; A920Max with USB-C
Android POS	Castles S1, Urovo	EMV, NFC, magnetic stripe	USB, RS232, Ethernet via dock	PCI PTS
USB reader	ID Tech VP3300	EMV, magnetic stripe	USB	PCI PTS, SCRA protocol
PIN-on-mobile	Ingenico Moby/5500	EMV, magnetic stripe, NFC	USB-C, adapter required	PCI SCRIP, PIN pad, clip for smartphones
Tag reader	ACR122U	NFC tags only (ISO 14443, MIFARE)	USB	not certified, no payments

An uncertified reader (ACR122U) **MAY** be used only for LSC tokens, access and loyalty cards and for development, and **NEVER** for payment cards. Devices without USB peripherals in their production version (Clover Flex) are unsuitable for this architecture.

19.3 Architecture

Card or QR code
|
Certified terminal / reader (PCI) → Processor / network operator (encrypted, P2PE)
| metadata only (TxMeta), USB or serial, authenticated and encrypted
Air-gapped device: wallet, audit log, QR display (no network, no Bluetooth)
| QR code or USB storage (batch hand-over)
Online system: blockchain connection, batch upload, reconciliation with the processor

Alternatively, the payment device MAY stay separate from the air-gapped device and exchange data through QR codes only (two-device variant), which keeps card data and wallet completely apart. The hybrid mode permits a brief network connection through a USB hub for authorization and disconnects immediately afterwards.

The reference air-gapped device is an Android device with USB host support (Replicant 6.0, Samsung Galaxy S3 or Note 2). The operator MUST verify beforehand that the device supports USB OTG (kernel option CONFIG_USB_OTG, test with a USB mouse).

19.4 Channel between terminal and air-gapped device

The channel carries only the metadata packet $TxMeta = \text{enc}(\text{version}, \text{term_id}, \text{ctr}, \text{amount}, \text{currency}, \text{chain_time}, \text{card_class}, \text{auth_result}, \text{ref})$. `card_class` names the carrier type (magnetic stripe, chip, contactless, QR code); `ref` is the transaction reference assigned by the processor and contains no card-derived value. Following the DUKPT principle (one derived key per transaction), every packet is protected with its own key:

$$K_{tx} = H(\text{enc}(\text{DOM}_{TERM}, K_{pair}, \text{ctr})), \quad c = \text{ChaCha20}(K_{tx}^e, \text{ctr}, m), \quad t = H(K_{tx}^m \parallel \text{ctr} \parallel c)$$

K_{pair} is set at pairing over a physical connection or a QR code. The counter `ctr` MUST increase strictly; a packet whose `ctr` is not greater MUST be discarded. The channel MUST NOT carry card data (R1).

19.5 Flows

- **Card payment.** The terminal reads the card and authorizes through the processor (offline where configured); it sends `TxMeta` to the air-gapped device, which logs it and produces the receipt. The online system uploads the transactions later as a batch.
- **LSC payment by QR.** The merchant shows a payment request (address, amount, reference, expiry height) as a QR code. The customer signs the transaction in the wallet and hands it over by QR or NFC. The online system places it in the mempool; confirmation depends on the value tier (§20.9). Coupling with other cryptocurrencies is outside this specification.

19.6 Security requirements

1. Terminal firmware MUST NOT run in developer mode in production; an enabled ADB can violate PCI compliance.
2. The USB channel MUST carry metadata only and MUST be authenticated and encrypted (§19.4).
3. Audit logs MUST be kept separately on terminal and air-gapped device.
4. The air-gapped device MUST run without network and Bluetooth, except in hybrid mode per §19.3.
5. Transaction data are subject to the GDPR; retention and deletion MUST be regulated.
6. In production the entire chain (terminal, processor, network operator) MUST be certified; development and tests use test cards and test mode.

20. Banknote-Inspired Security Layers

LSC adopts from the Lesotho Loti not the look but the principle: 14 mutually independent features on three verification tiers, all of which a forger must overcome at once. The feature list comes from a description of the Loti banknotes provided by the author and serves as a design analogy.

20.1 Mapping and tiers

Each feature receives a cryptographic construction with the same property, such as "not imitable without special equipment" or "visible only when checked". The mapping is an analogy, not a technical equivalence. The three tiers correspond to layperson, merchant and machine:

- **L1** (device without network, layperson): signature, window, network id, idempotency.
- **L2** (device with data connection, merchant): additionally snapshot reference, secure-element attestation, dual-hash serial, watermark.
- **L3** (full node, machine): additionally quorum seal, thread inclusion, proof of work, algorithm agility.

The banknote series model (successive generations that circulate side by side until the central bank withdraws the older ones by legal notice) corresponds to the series model of §13.2; the separation of instrument security from rule sovereignty (the Loti's peg to the rand) corresponds to §21; the circulation strategy of the Central Bank of Lesotho (demand-matched issue, clean-note policy, withdrawal only by formal notice, registers) is mapped onto reorganization stability in §1.5.

No	Banknote feature	Tier	LSC construction	Attack
1	watermark, multi-tone	L2	provenance watermark: hash lineage of every coin in three tones	T1, T6
2	security thread (window, 3D)	L3	thread accumulator: MMR root with OTP binding, signed in checkpoint windows	T4, T5
3	intaglio, deep print	L2	proof of cost: verification and forging effort far apart	T6
4	UV, two-colour serial number	L2	dual-hash serial from BLAKE2b and SHA-256	T1
5	UV fibres	L2	device-bound randomness: secure-element key (PUF analogy)	T3
6	microtext	L3	domain tags in every signature, micro-proofs	T2, T6

No	Banknote feature	Tier	LSC construction	Attack
7	latent image	L2	commit-reveal claim, Merkle opening	T1, T3
8	OVI, colour shift	L2	epoch and context binding of signatures	T2
9	hologram, patch	L3	quorum seal with value-tiered verification levels	T1, T8
10	anti-copy, void	L1	clone detection, one-time tag, idempotency	T2, T3
11	anti-IR	L3	algorithm agility through the segment registry	T6
12	micro-guilloche	L3	ECVRF proofs	T4, T8
13	protective varnish	L1	authenticated encryption at rest and on the terminal channel, store repair	T7
14	tactile features	L1	verification without infrastructure: compact signed token, one signature check	T1

20.2 Watermark: provenance

From Series 2 every coin MUST carry a lineage chain. The mint value binds the block, the index i of the reward share and the amount a ; every transfer appends the transaction digest $d(tx)$ and the output index j :

$$w_0 = H(\text{enc}(\text{DOM}_{WM}, B_h, i, a)), \quad w_{k+1} = H(\text{enc}(\text{DOM}_{WM}, w_k, d(tx), j))$$

The three "tones" are three verification depths: w_{mint} binds the issue, w_{claim} (derived from w_{mint} and the claim) the maturity, w_{last} the latest transfer. A verifier checks only the last step in $O(1)$ or the entire lineage in $O(k)$. Equal values for different coins would be a hash collision. The amount can also be audited: the amounts of all coins with mint height h sum to $R(h)$. Cost: 32 bytes per coin and one hash per output. The field `Coin.lineage` is additive; the value follows from block content and is reorganization-safe. Segment type 7 (LINEAGE) carries the root.

20.3 Security thread: accumulator

Blocks are kept as leaves of a Merkle Mountain Range (MMR). The root binds count and peaks:

$$R_n = H(\text{enc}(\text{DOM}_{TH}, n, \text{bag}(\text{peaks}))), \quad |\pi_j| = \lceil \log_2 n \rceil \text{ hashes}$$

From Series 2 checkpoints MUST sign the tuple (h, B_h, R_h, o_h) . These are the "windows": the root is exposed only there, and a verifier recomputes in between. The proof that block j belongs to the canonical chain comprises $\lceil \log_2 n \rceil$ hashes (for 1,000,000 blocks, 20 hashes, 640 bytes), so

light clients can verify without the full chain. The binding to o_h makes the thread time-dependent ("moving"). A forged inclusion needs a hash collision or the compromise of 16 of the 24 signers. Segment type 8 (THREAD) carries the root.

20.4 Intaglio: proof of cost

Expensive to produce, cheap to verify. The ratio of forging to verification effort is given in §12.6. The two segment tiers act like offset and deep print: LIGHT_PASS is cheap and reversible, HEAVY_PASS is an Argon2id commitment (64 MiB, 3 passes).

20.5 UV and two-colour serial number: dual hash

$$\sigma(x) = H_B(\text{enc}(\text{DOM}_{SER}, x)) \parallel H_S(\text{enc}(\text{DOM}_{SER}, x)), \quad H_B = \text{BLAKE2b-256}, \quad H_S = \text{SHA-256}$$

Lemma 3. A collision of σ is simultaneously a collision of H_B and of H_S ; therefore $\text{Adv}_{\text{coll}}(\sigma) \leq \min(\text{Adv}_B, \text{Adv}_S)$. *Proof sketch:* $\sigma(x) = \sigma(y)$ with $x \neq y$ makes both halves equal. The statement holds for collisions only; no gain beyond the stronger function is claimed (Joux 2004). Cost: 32 bytes per serial. Use: coin serial, token identifier, block references in checkpoints.

20.6 UV fibres: device-bound randomness

A secure element provides a key that never leaves the device. It carries $\text{device_id} = H(\text{enc}(\text{DOM}_{SE}, \text{pk_d}))$, a strictly monotonic counter c_d and the credit counter $u_d \leq U_{\max}$. If an attacker creates c clones with identical state, each can spend the remaining credit before a reconciliation takes place:

$$L_{\text{clone}} \leq c(U_{\max} - u_0)$$

Detection is certain as soon as two requests with the same (device, counter) reach the service.

20.7 Latent image: commit-reveal and Merkle opening

The claim value is revealed only on redemption. The stamp or token carries only the commitment C ; the secret k has at least 128 bits of entropy:

$$C = H(\text{enc}(\text{DOM}_{CL}, k, \text{owner})), \quad \Pr[\text{guess after } q \text{ attempts}] \leq q \cdot 2^{-128}$$

In a bearer token, k sits in a covered field whose reading burns it (one-time tag). Whoever sees only the payload cannot redeem. The second form is the Merkle opening: a state commitment coin_root is the root of a Merkle tree over the sorted coins, and a wallet proves with $\lceil \log_2 m \rceil$ hashes that its coin is part of state S_h without revealing other coins.

20.8 OVI: epoch and context binding

$$m' = \text{enc}(\text{net}, s, \lfloor h/E \rfloor, m), \quad \text{valid} \iff h_{\text{start}} \leq h \leq h_{\text{exp}}$$

Network (net), series (s) and epoch ($E = 1440$ blocks, one day) MUST be part of the signed message. The same bytes are worthless in another network, another series or after expiry. Transactions MAY carry an expiry height h_{exp} . A transaction MUST NOT be bound to a block hash (§14.7.2).

20.9 Hologram: quorum seal and value-tiered verification

The seal consists of layers:

- **S1:** signature of the issuer or owner.
- **S2:** S1 and inclusion in a block with a valid ECVRF proof.
- **S3:** S2 and a per-block endorsement seal (§1.5.5) or finalization by the checkpoint quorum (16 of 24).
- **S4:** S3 and thread inclusion (§20.3) and archive anchor.

With independent compromise of nodes with probability p , the forgery probability of S3 is:

$$P_f(S3) = \sum_{j=q}^N \binom{N}{j} p^j (1-p)^{N-j}, \quad N = 24, q = 16$$

p per node	P_f(S1), one key	P_f(S3), 16 of 24
0.01	1.0e-2	6.8e-27
0.05	5.0e-2	7.6e-16
0.10	1.0e-1	3.3e-11
0.20	2.0e-1	9.1e-7

Value rule. An amount v MUST be accepted at the lowest level ℓ for which $v \cdot P_f(\ell) \leq \epsilon$. For $p = 0.05$ and $\epsilon = 1$ unit of expected loss, S1 suffices up to $v = 20$, S3 practically without limit. Independence is an idealization; the table is an optimistic bound. The offline policy of §18.4 (secure element above amount 50, snapshot reference above amount 50) maps L1 and L2; S3 and S4 apply from Series 3.

20.10 Anti-copy and void: detection instead of prevention

An offline copy cannot be prevented; the void pattern appears only when the note is checked. LSC detects: an identical request yields the same result (idempotent settlement), the same device counter with a different request is detected as a clone, a second write to a one-time tag fails, and a second redemption of the same stamp is rejected. Damage stays limited by the loss bound of §20.6; the ledger can never pay out the same stamp twice, because it releases the stamp only once.

20.11 Anti-IR: algorithm agility

The set of permitted algorithms A_s belongs to series s ; a change is a new segment type or new parameters with an activation height, and old nodes skip unknown types (§13.1). Because miners must support several algorithms, specialized hardware must cover several resource profiles (memory, cache, latency). A change proceeds by CorrectionRFC and ParameterRegistry (§13.4).

20.12 Micro-guilloche: ECVRF

A guilloche pattern is complex, arises from a short key and cannot be reproduced without it. The ECVRF proof has the same properties: for every pair (pk, α) there is exactly one output β , it is pseudorandom without sk , and anyone verifies it publicly. A proof is 80 bytes; the winning probability per slot is $p = \tau/2^{256}$ (§1.1).

20.13 Protective varnish: authenticated encryption and repair

Stored data MUST be protected in the encrypt-then-MAC pattern with separate keys; the same pattern protects the terminal channel (§19.4):

$$k_e = H(\text{ENC}\|K), \quad k_m = H(\text{MAC}\|K), \quad c = \text{ChaCha20}(k_e, n, m), \quad t = H(k_m\|n\|c)$$

The file consists of magic, nonce n , ciphertext c and tag t . Writing is atomic through a temporary file; the block store carries checksums and repair states (§17.5). Node and issuer keys MUST be stored encrypted in this way.

20.14 Tactile features: verification without infrastructure

A verifier without network and without a full node MUST be able to check a signed token: the check costs one Ed25519 verification, one hash and a window comparison. This fits a small tag, a card and a simple device.

20.15 Interplay

An attack of class T_x must break all layers that cover that class. If the layers are independent, the individual probabilities multiply:

$$P_{\text{forgery}}(\ell) = \prod_{i \in \text{layers}(\ell)} P_i$$

Independence holds approximately; §15.11 lists the cases where it is violated.

21. Sovereignty and Dependencies

LSC has no peg; its external dependencies are the genesis time anchor, key, card and mining hardware, the software supply chain and the cohort itself.

According to the Loti description, the notes are technically secure, but monetary policy is not independent: the rand is legal tender, Lesotho belongs to the Common Monetary Area (CMA), interest policy follows South Africa, and the central bank cannot abandon the peg. Translated: the security of an instrument says nothing about who sets the rules. The table separates both for LSC.

Dependency	Loti analogy	LSC mechanism	Limit
Time anchor	1:1 peg to the rand	one-time median of 10 anchors, then chain time	shiftable from 5 colluding anchors; affects only the start
Money-supply rule	interest policy follows South Africa	reward schedule in consensus, no external authority	a change needs the unanimity of the 24 nodes
Rule authority	central bank cannot abandon the peg	the cohort sets rules through governance	with collusion of at least 16 nodes the rules change
Key and card hardware	printing works and substrates	secure-element vendors, certified readers and terminals	vendor backdoor, certification dependency

Dependency	Loti analogy	LSC mechanism	Limit
Payment processor	not applicable	cards run through the processor, LSC sees metadata only (§19)	processor and network-operator dependency
Mining hardware	printing processes	three algorithms with different resource profiles	concentration on one hardware type
Software	not applicable	RandomX library, RPython toolchain	supply chain
Network	CMA infrastructure	local first, DNS seeds only on the internet path	DNS censorship affects only the internet path

LSC is sovereign in the sense that after start no external authority prescribes rules, supply or time. It is not decentralized in the sense of open participation: 24 nodes carry scheduling, finality and governance, so sovereignty is cohort sovereignty. A larger cohort changes only constants:

$$q = \lceil 2N/3 \rceil, \quad f_{\max} = \lfloor (N-1)/3 \rfloor \quad (N = 24 : q = 16, f_{\max} = 7)$$

The procedure for admitting new nodes MUST run through a quorum of the cohort; the start cohort and revocation are fixed in §9.1 and §14.4.

Annex A: Consolidated Parameters

Parameter	Value	Section
Number of LSC nodes N	24	§14.3
Block interval	60 s	§14.3
Maximum block size	2 MiB	§14.3
Blink allocation	524,288 bytes (512 KiB, 25 %)	§14.3
Maximum micro-proofs per block	500	§14.3
Maximum HEAVY_PASS per block	32	§14.3
Drift tolerance	2 s	§14.3
HEAVY_PASS Argon2id	64 MiB, 3 passes, parallelism 1, tag 32 bytes	§14.3
Base units per LSC	10^8 (8 decimal places)	§14.5.1
Initial block reward	50 LSC = $5 \cdot 10^9$ base units	§14.5
Halving interval	2,100,000 blocks (about 3.99 years)	§14.5
Mining hold	10,800 s (3 h), aligned up to 3,600 s	§12.3
Checkpoint interval	60 blocks	§16.1
Checkpoint quorum	16 (equal to $\lceil 2N/3 \rceil$)	§16.1
Drift adjustment	at most 5 s per 10 blocks	§7.1
State cache	every 8 blocks, at most 64 states	§1.4
Mempool limit	5000 operations	§17.6
inv limit, locator limit	500, 64	§17.2, §17.3

Parameter	Value	Section
Segment types	1 LIGHT_PASS, 2 HEAVY_PASS, 3 TIME_ANCHOR, 4 BLINK, 5 MINING_PROOF, 6 MICRO_PROOF, 7 LINEAGE, 8 THREAD, 9 SEAL	§2.2
Target eligible nodes per slot λ^*	3	§1.5.3
Proof window w_p (reference value)	3 s; MUST exceed the 99th-percentile proof delay	§1.5.4
Fallback rounds	threshold doubles per round, at most 4 rounds	§1.5.4
Controller window, dead band, clamp	120 blocks, 15 %, ± 5 % per window	§1.5.3
Endorsement quorum, size	16, 65 bytes each	§1.5.5
Redemption claim window	1440 blocks	§18.2
Redemption drift tolerance	5 blocks	§18.2
Snapshot interval	60 blocks	§18.6
Offline limits	100, 500 LSC	§18.4
$\text{max_offline_amount}$, max_offline_total		
Secure element and snapshot reference required above	50, 50 LSC	§18.4
Epoch E	1440 blocks	§20.8
Activation delay	2 windows (120 blocks)	§13.3, §13.4

Annex B: Formulas and Numerical Tables

C.1 Formula overview

Quantity	Expression	Section
Chain time	$t(h) = G + h \cdot T + A_h$, $T = 60$ s	§7.1
Drift bound	$ A(h+W) - A(h) \leq 5$ s per $W = 10$ blocks	§7.1
Block hash	$B_h = H(\text{enc}(h, t(h), P , P, \rho_h))$	§2.1
Fork probability	$P(X \geq 2) = 1 - (1-p)^N - N \cdot p \cdot (1-p)^{(N-1)}$	§1.1
Fork bound with slot-proof gossip	$P(\text{fork}) \leq P(X \geq 2) \cdot \epsilon_g / (1 - \epsilon_g)$	§1.5.4
Observed proofs per block	$E[X \text{ given } X \geq 1] = N \cdot p / (1 - (1-p)^N)$	§1.5.3
Finality safety	$ Q1 \cap Q2 \geq 2q - N = 8 > f$	§16.5
Miner share	$s_i = \lfloor R(h) \cdot W_i / \sum W_j \rfloor$	§12.5
Reward	$R(h) = \lfloor R0 / 2^{\lfloor h/H \rfloor} \rfloor$	§14.5.1
Maturity window	$w_c = 3600 \cdot \lceil (t(h) + 10800) / 3600 \rceil$	§5.1
Conservation of supply	sum of unspent coins = sum of $R(k)$, $k \leq h$	§5.5
Forging versus verification effort	$\rho \approx k \cdot N / 2^k$	§12.6

Quantity	Expression	Section
Loss through clones	$L \leq c \cdot (U_{\max} - u_0)$	§20.6
Forgery of seal level S3	sum over $j \geq q$ of $C(N,j) \cdot p^j \cdot (1-p)^{(N-j)}$	§20.9
Terminal channel key	$K_{\text{tx}} = H(\text{enc}(\text{DOM_TERM}, K_{\text{pair}}, \text{ctr}))$	§19.4

C.2 Locator and MMR proof

The locator holds the last 10 hashes without gaps, then doubling distances; the MMR proof has $\lceil \log_2 n \rceil$ hashes of 32 bytes each.

Chain length n	Locator entries (bytes)	MMR proof (hashes, bytes)
100	16 (528)	7 (224)
10,000	23 (759)	14 (448)
1,000,000	29 (957)	20 (640)
10,000,000	33 (1,089)	24 (768)

C.3 Collision probability

For n hashes of length b bits, $P(\text{collision}) \approx n^2 / 2^{(b+1)}$.

Hash length b	n = 2 ³⁰	n = 2 ⁴⁰
128 bits (reference value)	2 ⁻⁶⁹	2 ⁻⁴⁹
256 bits (BLAKE2b-256, SHA-256)	2 ⁻¹⁹⁷	2 ⁻¹⁷⁷

References

- van Saberhagen, N. *CryptoNote v2.0* (whitepaper), 2013.
- Noether, S. *Review of CryptoNote White Paper*, Monero Research Lab.
- Noether, S. et al. *Ring Confidential Transactions*, Ledger journal, 2016.
- Möser et al. *An Empirical Analysis of Traceability in the Monero Blockchain*, arXiv:1704.04299.
- Feathercoin project documentation, feathercoin.com and Feathercoin community forum.
- Biryukov, A. and Khovratovich, D. *Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem*, 2015 (used as Zcash's proof-of-work algorithm).
- Various offline e-cash and non-fungible/grouped-token research, cited by topic in §5.
- Bernstein, D. J. *ChaCha*, a variant of *Salsa20*, and RFC 8439 (*ChaCha20 and Poly1305 for IETF Protocols*) — used for LIGHT_PASS (§3).
- Biryukov, A., Dinu, D., Khovratovich, D. *Argon2*, and RFC 9106 (*Argon2 Memory-Hard Function*) — used for HEAVY_PASS (§3).
- Bernstein, D. J. et al. *Ed25519: High-speed high-security signatures* — used for all signatures (§14.1).
- Goldberg, S. et al. RFC 9381 (*Verifiable Random Functions (VRFs)*) — ECVRF-EDWARDS25519-SHA512-TAI used for block-production scheduling (§1).
- BIP-39 (*Mnemonic code for generating deterministic keys*) and SLIP-0010 (*Universal private key derivation from master private key*) — used for the wallet/key-custody model (§14.7.1).
- Castro, M. and Liskov, B. *Practical Byzantine Fault Tolerance*, OSDI 1999 — the $N \geq 3f + 1$ threshold used in §16.5.
- Micali, S., Rabin, M., Vadhan, S. *Verifiable Random Functions*, FOCS 1999.

- Wagner, D. *A Generalized Birthday Problem*, CRYPTO 2002.
- Joux, A. *Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions*, CRYPTO 2004 — dual-hash serial (§20.5).
- Dwork, C. and Naor, M. *Pricing via Processing or Combatting Junk Mail*, CRYPTO 1992.
- Merkle, R. *A Digital Signature Based on a Conventional Encryption Function*, CRYPTO 1987 — Merkle trees and the MMR variant (§20.3, §20.7).
- Nakamoto, S. *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008.
- RFC 2104 (HMAC), RFC 2119 (requirement keywords), RFC 7693 (BLAKE2), RFC 8032 (EdDSA), FIPS 180-4 (SHA), BIP-173 (Bech32).
- ISO/IEC 7816 (chip cards), ISO/IEC 14443 (contactless cards) and the EMV specifications — terminal and air-gap layer (§19).
- RandomX (specification and library) and NeoScript (reference implementation) as proof-of-work algorithms.
- The author's description of the Lesotho Loti banknote security features (§20, §21); historical details in it were not independently verified.
- Market research on card readers and payment terminals, October 2025 (§19.2).
- Central Bank of Lesotho, *Currency* (centralbank.org.ls/currency): issuing, supply management, clean-note policy, redemption, currency registers (§1.5).
- Lesotho Times and AllAfrica, *CBL strengthens Maloti banknotes' security features*, December 2025: earlier series remain legal tender until withdrawn by legal notice (§1.5.6).
- Africa Press, *Central Bank of Lesotho launches new notes: clean-banknote policy, sorting machines, no forced exchange* (§1.5).
- Wikipedia, *Lesotho loti*: parity with the South African rand within the Common Monetary Area (§1.5.1, §21).

This document is an original conceptual synthesis for discussion purposes, grounded in the published CryptoNote whitepaper and academic commentary on it, Feathercoin's own published parameters, and general patterns in offline e-cash and non-fungible token research (cited by topic, not reproduced verbatim). It does not represent an existing project, is not audited, and should not be used as the basis for an actual financial or infrastructure product without independent cryptographic and systems-security review.