Decentralized State Machine: A Concise, Post–Quantum Specification

Brandon “Cryptskii” Ramsay

December 10, 2025

The modern internet relies on centralized trust infrastructures such as certificate authorities, OAuth servers and validator networks. These architectures impose bottlenecks, enable censorship and remain vulnerable to quantum computing. The Decentralized State Machine (DSM) is a cryptographically self–verifying framework that replaces consensus, accounts and third–party authorization with deterministic hash chains and Merkle commitments. This document synthesizes the DSM architecture and its recent extensions into a concise, implementable specification. It formalizes bilateral state progression *without clocks or heights*, introduces the Tripwire fork–exclusion theorem, incorporates the Dual–Binding Random Walk (DBRW) anti–cloning mechanism, details an offline recovery protocol using encrypted NFC capsules, and specifies a physics–backed geo–emissions scheme for location–dependent rewards. All algorithms are post–quantum secure and admit efficient verification on resource–constrained devices. Terminology is aligned with current usage: we use *inclusion proof* (not “membership proof”), we avoid “relationship keys,” and ordering is by *hash adjacency*, not time.

1 Introduction

The original vision for a peer–to–peer electronic cash system promised direct transactions without financial intermediaries. Conventional blockchains approximate this ideal yet still require global consensus, miners or validators, and incur probabilistic finality. DSM dispenses with global state entirely by localizing state to bilateral relationships and enforcing forward–only progression through cryptographic commitments. Each participant maintains independent hash chains for every counterparty; transactions are validated by the involved parties alone. This architecture eliminates reorganization, censorship and liquidity constraints while enabling true offline operation.

Paradigmatic shift (beyond the account model). DSM is not a better account ledger; it replaces the *account model* underlying the modern internet. In DSM, identity and ownership are not mutable rows curated by institutions but *immutable mathematical objects* bound to users' cryptographic state. Devices attach to a user's *genesis* via the Device Tree, and each device's Per–Device SMT defines the user's bilateral relationships from first principles—eliminating custodial account recovery, authorization servers, and third-party revocation lists. The result is a continuously evolving, self-verifying user-controlled state, rather than institution-controlled accounts.

2 Cryptographic Foundations

2.1 Straight Hash Chains

A hash chain encodes ordering by linking each state to the hash of its predecessor. Formally, a *straight hash chain* $C = (S_0, S_1, \dots)$ satisfies

$$S_{n+1} = \mathsf{Build}(S_n, \text{payload}_{n+1}), \qquad h_n := H(S_n), \quad h_{n+1} := H(S_{n+1}), \tag{1}$$

with *adjacency* requiring that h_n is embedded in S_{n+1} and verified under canonical encoding. Every state commits to its entire history; reversing or editing requires a collision or second preimage.

To accelerate lookups, an optional sparse index over checkpoints $\{S_k, S_{2k}, \dots\}$ can be maintained; this is an implementation optimization that does not affect acceptance rules.

2.2 Merkle Trees and Sparse Merkle Trees

DSM uses two Merkle structures:

- **Device Tree (standard Merkle).** A standard Merkle tree whose leaves are *device identifiers* DevID owned by a single *genesis* account G. This tree binds all devices to G.
- **Per–Device SMT.** For each device A, a *Sparse Merkle Tree (SMT)* indexes A's bilateral relationships. Each leaf represents one relationship ($A \leftrightarrow B$) and stores the current relationship commitment (e.g. the current chain tip digest $h^{A\leftrightarrow B}$).

In both structures, internal nodes store the hash of their two children and inclusion proofs are logarithmic in the tree depth.

SMT Parameters: The zero leaf value is `ZERO_LEAF` (exactly 32 zero bytes). Relationship keys are derived as $H(\text{“DSM/smt-key\textbackslash 0”} \| \text{DevID}_A \| \text{DevID}_B)$ where $\text{DevID}_A < \text{DevID}_B$ lexicographically to ensure canonical ordering.

Zero leaf.

$$\texttt{ZERO\_LEAF} := \underbrace{\texttt{0x00}\text{ repeated 32 times}}_{\text{exactly 32 zero bytes}}.$$

Default nodes.

$$\text{DEFAULT}[0] := \texttt{ZERO\_LEAF}, \qquad \text{DEFAULT}[d{+}1] := H\Big(\text{DEFAULT}[d] \parallel \text{DEFAULT}[d]\Big)\ \forall\, d \geq 0,$$

with

$$H(L, R) := \text{BLAKE3}\Big(\texttt{"DSM/smt-node\textbackslash 0"} \parallel L \parallel R\Big).$$

Keyspace & bit order. SMT keys are 256-bit big-endian; bit i walks from MSB to LSB (MSB-first).

Non-inclusion proof (SMT). Structure: proof encodes $\Big(k,\ v_{\text{path}},\ \texttt{siblings}[]\Big)$ where:

- k is the 32-byte key queried.
- v_{path} is either
  1. the existing leaf $\Big(k', v'\Big)$ at the first divergence bit from k, or
  2. the explicit marker `absent` if the path only hits DEFAULT nodes.
- `siblings[]` is the ordered list of 32-byte sibling hashes from root to leaf.

VerifyNonInclusion(*root*, k, *proof*):

1. Walk `siblings[]` using MSB-first bit order to reconstruct the path.
2. Case (a): show $k' \neq k$, and that inserting $\Big(k,\ \texttt{ZERO\_LEAF}\Big)$ at the divergence changes the path; the reconstructed root equals *root*.
3. Case (b): reconstruct *root* solely from DEFAULT nodes and `siblings[]`.

CBOR encoding (normative).

```
{ k: bstr .size 32,
  v_path: bstr|null,       # null means "absent"
  siblings: [* bstr .size 32] }
```

2.3 Two–Layer Commit Path (Genesis to Device to Relationship)

Each accepted relationship state has a compact commit path:

$$h^{A\leftrightarrow B} \xrightarrow{\pi_{\text{rel}}} r_A \quad \text{and} \quad \text{DevID}_A \xrightarrow{\pi_{\text{dev}}} R_G, \tag{2}$$

where r_A is A's Per–Device SMT root and R_G is the Device Tree root for genesis G. Here π_{rel} and π_{dev} are *inclusion proofs*. This ties every bilateral update to both the device and its genesis without any global ledger.

2.4 Genesis, Device IDs, and Domain Separation

Each user has a genesis digest $G \in \{0,1\}^\lambda$. Each device holds a long–term post–quantum attestation keypair $(\text{sk}_A, \text{pk}_A)$ (SPHINCS+), a stable device identifier DevID_A (e.g. a domain–separated hash of pk_A and attestation), and a Per–Device SMT over relationships. All hashes and signatures are domain–separated (e.g. labels like "DSM/receipt").

2.5 Genesis State Creation

Let $b_1, \ldots, b_t$ be independent entropy contributions and A contextual binding parameters. Define

$$G = H(b_1 \| \cdots \| b_t \| A). \tag{3}$$

3 Two Merkle Structures: Storage and Replication

Device Tree (standard Merkle). The Device Tree (leaves = device IDs) is *fully replicated* across storage nodes *and* across *all* devices under G. Adding a device is an online event: a new leaf is inserted and the updated root R_G is propagated to all devices.

Per–Device SMT. Each device A maintains its own Per–Device SMT root r_A. These per–device SMTs are *not* mirrored across the user's other devices. Storage nodes keep *aggregated* mirrors (e.g., latest r_A and compact indices) for availability and recovery.

4 State Transition Protocol (Clockless Ordering)

Fix parties (A, B) with local parent tip h_n for the relationship $C_{A\leftrightarrow B}$ at device A.

4.1 Pre–commitment and Attestation

Initiator prepares a precommit with fresh entropy e:

$$C_{\text{pre}} = H\Big(h_n \| \text{payload} \| e\Big). \tag{4}$$

Counterparty verifies and co–signs C_{pre}. The successor S_{n+1} embeds h_n; $h_{n+1} = H(S_{n+1})$.

4.2 Receipt Construction (Per–Device SMT Replace)

A updates its Per–Device SMT by replacing the leaf for $(A \leftrightarrow B)$ from h_n to h_{n+1}, producing r'_A. The stitched receipt encodes

$$\begin{aligned}\tau_{A\to B} = \ \text{enc}\Big(&\text{“DSM–StitchedReceipt–v1”},\ G,\ \text{DevID}_A,\ \text{DevID}_B,\\ &h_n,\ h_{n+1},\ r_A,\ r'_A,\\ &\pi_{\text{rel}}\Big(h_n \in r_A\Big),\ \pi'_{\text{rel}}\Big(h_{n+1} \in r'_A\Big),\ \pi_{\text{dev}}\Big(\text{DevID}_A \in R_G\Big)\Big),\end{aligned}$$

and is signed by both parties (Sec. 11.1). *Note:* the Device Tree R_G does not change for relationship updates; it is referenced only for device→genesis binding.

4.2.1 Canonical Commit Form (Frozen)

The canonical commit form defines the byte-exact serialization used for hashing and signing operations, separate from Protobuf transport envelopes. All cryptographic commitments (receipt signatures, precommitments, and inclusion proofs) use this deterministic encoding.

ABNF/CBOR-Deterministic Definition The canonical form uses CBOR with deterministic encoding rules (RFC 8949 §4.2.1):

```
receipt-commit = [
  genesis: bstr .size 32,           ; G (32-byte hash)
  devid-a: bstr .size 32,           ; DevID_A (32-byte hash)
  devid-b: bstr .size 32,           ; DevID_B (32-byte hash)
  parent-tip: bstr .size 32,        ; h_n (32-byte hash)
  child-tip: bstr .size 32,         ; h_{n+1} (32-byte hash)
  parent-root: bstr .size 32,       ; r_A (32-byte SMT root)
  child-root: bstr .size 32,        ; r_A’ (32-byte SMT root)
  rel-proof-parent: bstr,           ; pi_rel(h_n in r_A) (variable length)
  rel-proof-child: bstr,            ; pi’_rel(h_{n+1} in r_A’) (variable length)
  dev-proof: bstr                   ; pi_dev(DevID_A in R_G) (variable length)
]
```

Encoding Rules

1. All fields are encoded as CBOR arrays with fixed field order
2. Binary strings (bstr) use definite length encoding
3. Proofs are encoded as CBOR byte strings containing the Merkle proof data
4. No optional fields or extensions are permitted
5. The resulting CBOR is hashed with BLAKE3 to produce the commitment

Structure (normative). *receipt-commit* := CBOR array (major type 4) of length exactly 10, with items indexed a_i for $i \in \{0, \dots, 9\}$.

Encoding rules (normative).

- Deterministic CBOR (RFC 8949 §4.2.1).
- All ten fields are CBOR `bstr` (major type 2). For a_i with $i \in \{0, \dots, 6\}$, length = 32 bytes (exact).
- For a_i with $i \in \{7, 8, 9\}$ (proofs), use *definite-length* `bstr` only; indefinite-length strings are forbidden.
- The outer array is *definite-length*; indefinite-length arrays are forbidden.

Hashing (normative).

$$\mathsf{commit} := \text{BLAKE3-256}\Big(\texttt{"DSM/receipt-commit\textbackslash 0"} \parallel \texttt{canonical\_cbor\_bytes}\Big).$$

The ASCII domain tag plus NUL (`\0`) is prepended byte-for-byte to `canonical_cbor_bytes` prior to hashing.

Test Vectors One canonical test vector is provided for implementation validation. All implementations MUST produce identical CBOR encodings, hex representations, and BLAKE3 digests for this vector.

Minimal Receipt (Empty Proofs):

```
Receipt fields (all 32-byte values in hex):
  G: 0000000000000000000000000000000000000000000000000000000000000000
  DevID_A: 1111111111111111111111111111111111111111111111111111111111111111
  DevID_B: 2222222222222222222222222222222222222222222222222222222222222222
  h_n: 3333333333333333333333333333333333333333333333333333333333333333
  h_{n+1}: 4444444444444444444444444444444444444444444444444444444444444444
  r_A: 5555555555555555555555555555555555555555555555555555555555555555
  r_A’: 6666666666666666666666666666666666666666666666666666666666666666
  pi_rel(h_n in r_A): (empty)
  pi’_rel(h_{n+1} in r_A’): (empty)
  pi_dev(DevID_A in R_G): (empty)

CBOR encoding (canonical, definite-length):
8a58200000000000000000000000000000000000000000000000000000000000000000
58201111111111111111111111111111111111111111111111111111111111111111
58202222222222222222222222222222222222222222222222222222222222222222
58203333333333333333333333333333333333333333333333333333333333333333
58204444444444444444444444444444444444444444444444444444444444444444
58205555555555555555555555555555555555555555555555555555555555555555
58206666666666666666666666666666666666666666666666666666666666666666
40 40 40

Hex representation: 8a582000000000000000000000000000000000000000000000000000000000000000

BLAKE3-256("DSM/receipt-commit\\textbackslash0" || CBOR):
9786d0731e1fa88f6f0c4dd5e2a176984825e22ee2051027811f9965a47ecacb

Reproduction recipe:
1. Construct CBOR array with 10 elements as shown above
2. Use RFC 8949 canonical encoding with definite-length bstr
3. Compute BLAKE3-256("DSM/receipt-commit\\textbackslash0" || encoded_bytes)
4. Verify output matches hex digest above
```

4.3 Verification Rules

To accept a claimed update $(h_n \to h_{n+1})$ rooted at $(r_A \to r'_A)$ under G, a verifier checks:

1. Both signatures verify under the presented SPHINCS+ public keys (Sec. 11.1).
2. π_{rel} proves $h_n \in r_A$ and π'_{rel} proves $h_{n+1} \in r'_A$ (Per–Device SMT inclusion).
3. π_{dev} proves DevID_A is included in R_G (Device Tree inclusion).
4. Recomputing the Per–Device SMT leaf replace yields r'_A byte–exactly.

5. The parent tip h_n has not been previously consumed for this relationship.

There are *no timestamps, heights, or counters* in acceptance predicates.

The acceptance predicate is fully algorithmic:

$$\mathrm{addr}_{A\to B} := \texttt{b0x[}\,\text{BLAKE3-256}\Big(\texttt{"DSM/addr-G\textbackslash 0"} \parallel G \parallel \mathrm{salt}_G\Big)_{0..31}\ ;\ \text{BLAKE3-256}\Big(\texttt{"DSM/addr-D\textbackslash 0"} \parallel \text{DevI}$$

4.4 Deterministic Transition Guarantees (Adjacency Only)

Let $V(S_n, S_{n+1})$ be the predicate that the receipt for S_{n+1} is valid given S_n. Then:

$$V(S_n, S_{n+1}) \Rightarrow \mathsf{EmbedParent}(S_{n+1}) = h_n, \tag{5}$$

$$V(S_n, S_{n+1}) \wedge V(S_n, S'_{n+1}) \Rightarrow S_{n+1} = S'_{n+1}, \tag{6}$$

$$V(S_n, S_{n+1}) \Rightarrow \mathsf{PerDeviceReplace}(r_A, h_n \mapsto h_{n+1}) = r'_A, \tag{7}$$

$$\neg\exists\, S'_{n+1} \neq S_{n+1} : V(S_n, S'_{n+1}) \text{ and } \mathsf{Accept}(S'_{n+1}) = 1. \tag{8}$$

For token balances (Sec. 8), admissible successors preserve supply locally and globally.

5 Online and Offline Transport

5.1 Online Unilateral Transport: `b0x[...]`

Online sends are delivered unilaterally to a deterministic prefix:

$$\boxed{\texttt{b0x[}\,\mathrm{hex}\Big(\text{BLAKE3-256}(\texttt{"DSM/addr-G\textbackslash 0"} \parallel G \parallel \mathrm{salt}_G)\Big)\ \mathrm{hex}\Big(\text{BLAKE3-256}(\texttt{"DSM/addr-D\textbackslash 0"} \parallel \mathrm{DevID}_B \parallel} \tag{9}$$

where G is the recipient's genesis, DevID_B the recipient device, h_n the sender's current parent tip for $(A \leftrightarrow B)$, nonce an ephemeral per-send value, and salt_G, salt_D are per-user blinding salts. All three components are blinded to prevent correlation attacks while maintaining deterministic addressing. The recipient applies the candidate iff it is adjacent to its local parent and the included proofs in Sec. 4.2 verify. Otherwise it is queued (waiting for predecessors) or rejected.

5.2 Offline Bilateral

Offline requires both parties live (e.g. Bluetooth/NFC). The parties exchange precommitments, finalize, and countersign the receipt locally (no `b0x`).

5.3 Modal Synchronization Lock

Let $\mathsf{Pending}_{A \leftrightarrow B}$ hold if an accepted but unsynchronized online projection exists for (A, B) in either party's `b0x` or local queue.

Theorem 1 (Pending–Online Lock)**.** *If* $\mathsf{Pending}_{A \leftrightarrow B}$ *holds, initiating an* offline *transaction for* (A, B) *is invalid until synchronization clears the pending items. Relationships* (A, C) *for* $C \neq B$ *proceed unaffected.*

Sketch. Both online and offline consume the same parent. Proceeding offline while a conflicting online projection exists risks parent divergence; adjacency uniqueness would be violated. Disjoint relationships commute. □

6 Tripwire Theorem and Causal Consistency

6.1 Atomic Interlock Tripwire

The Tripwire theorem formalizes fork exclusion in stitched DSM updates.

Theorem 2 (Atomic Interlock Tripwire)**.** *Assume SPHINCS+ is EUF–CMA and H is collision resistant. The probability that an adversary generates two distinct receipts that both consume the same parent tip and both verify is negligible.*

Sketch. Two accepted successors to the same parent require either a signature forgery or a collision in the chained hash or Merkle commit path. □

Intuition: Tripwire as a Ledger Replacement. Each device A maintains a Per–Device SMT with root r_A that commits to all bilateral relationships $(A \leftrightarrow B)$ as leaves, with each leaf storing the current relationship tip $h^{A \leftrightarrow B}$. Whenever (A, B) updates their chain, both parties update their local SMT roots, and stitched receipts prove inclusion of the old and new tips under r_A (and, symmetrically, under r_B if desired), plus inclusion of DevID_A in the Device Tree root R_G.

Suppose an adversary attempts to double-spend on (A, B) by producing two conflicting successors that both claim to consume the same parent h_n. Any honest device that later interacts with A (or B) demands inclusion proofs under the presenting device's current r_A (or r_B). Maintaining both forks would require either:

- two incompatible leaves for the same relationship under a single SMT root, or
- two incompatible SMT roots for the same device key that both verify against the same stitched history.

Either case forces a collision in the hash chain or Merkle path. Thus, even devices that never transacted directly (e.g. Alice with Charlie) are wired into a shared global invariant via shared counterparties: per-device SMT roots and stitched receipts form a web of "tripwires" that collectively forbid double-spend, without a global public ledger.

6.2 Causal Consistency

Stitched receipts induce a DAG of Per–Device SMT roots across devices. A root r_D is accepted iff for every referenced relationship tip along the path into r_D, there exists a valid *inclusion proof* demonstrating its presence in the corresponding Per–Device SMT and a Device Tree inclusion for the signing device. This enforces causal consistency without a global sequence.

6.3 First–Contact Binding

When an isolated device presents its first countersigned receipt, it irrevocably binds to that branch: future states must extend it, or verification fails unless H or SPHINCS+ is broken.

7 Architectural Rationale and Differentiators

This section concisely integrates key architectural context from the long-form paper into the implementable specification. It explains *why* DSM adopts these design choices and highlights the practical consequences for deployment and operations.

7.1 Subscription-Based Economic Model (Gasless Operation)

DSM replaces per-transaction gas with a *subscription-based* model that aligns cost with persistent resource use rather than event frequency.

- **Storage-proportional fees.** Users fund storage and availability via periodic subscriptions that scale with retained state (device tree entries, per-device SMT heads, and retained proofs), not with the number of state transitions.
- **One-time creation fees.** Token/minted-asset creation incurs a one-time fee that covers indexing and archival commitments.
- **Operator sustainability.** Storage nodes are paid for capacity, durability, and retrieval bandwidth rather than transient compute. This cleanly answers "how are storage nodes paid?" and removes incentives to throttle usage via gas.

Result: users experience gas-free transactions; developer UX is predictable; and economics track the real cost drivers (storage and bandwidth), not click-volume.

7.2 Deterministic Smart Commitments vs. Turing-Complete Contracts

DSM intentionally *does not* expose a Turing-complete contract VM. Instead it uses *deterministic smart commitments*—bounded, verifiable state machines assembled from pre-commitments and stitched receipts.

- **Security by construction.** By excluding unbounded control flow, DSM removes entire bug classes (reentrancy cascades, halting/DoS via infinite loops, gas griefing) and enables straightforward formal auditing of admissible transitions.
- **Expressiveness via pre-commitment forking.** Complex, multi-path workflows are expressed by preparing multiple *pre-commit* digests (branch candidates) and later ratifying *exactly one* adjacent successor. Tripwire (fork exclusion) ensures only a single branch can be accepted for a given parent.
- **Determinism.** Acceptance predicates depend solely on hash adjacency, inclusion proofs, and signatures (no clocks, no global height). This keeps validation portable and offline-capable.

7.3 Deterministic Limbo Vault (DLV): Purpose and Lifecycle

The DLV is a cryptographic construction for *trustless asset management* under self-executing conditions—without external oracles or a contract VM.It complements the invariants stated earlier with a clear operational lifecycle:

1. **Create and encumber.** A vault configuration (L, C, H) is committed, and assets are placed under the vault's control with a public commitment to the lock L and condition set C.
2. **Accrue proofs.** Parties produce stitched receipts that, when combined, cryptographically attest the satisfaction of C.
3. **Derive unlock key.** The unlocking secret becomes computable only upon fulfillment of C via a stitched proof-of-completion σ:

$$\mathrm{sk}_V = H\Big(L \| C \| \sigma\Big).$$

   Prior to σ, sk_V is infeasible to derive.

Result: autonomous escrow, deferred payments, and contingent releases operate fully offline and remain verifiable under DSM's receipt algebra.

7.4 Security: Bilateral Control Attack Vector

DSM explicitly analyzes the edge case where a single adversary momentarily controls both parties to a relationship ("bilateral control").Even in this strongest per-relationship threat model:

- The adversary can produce *valid* signatures on conflicting candidates, but cannot make both successors *acceptable* because Tripwire forbids consuming the same parent twice.
- Crucially, the *mathematical invariants* (e.g., conservation of balances, uniqueness of parent consumption) remain inviolable: any transition violating these constraints is rejected as invalid.

Implication: bilateral control does not enable double-spend; it only permits the adversary to choose *which* valid successor gets finalized, never to realize an arithmetically impossible state.

8 Token Management and Balance Invariants

Let B_n be the token balance at S_n. Valid updates satisfy

$$B_{n+1} = B_n + \Delta_{n+1}, \qquad B_{n+1} \geq 0. \tag{10}$$

For a transfer α, sender and recipient use $\Delta_{\text{sender}} = -\alpha$ and $\Delta_{\text{recipient}} = +\alpha$. Summing Δ across all parties is zero, preserving total supply without global synchronization. Each state binds $(e'_{n+1}, \text{encapsulated}_{n+1}, B_{n+1}, H(S_n), \text{op}_{n+1})$ under canonical encoding.

Theorem 3 (Double–Spending Impossibility)**.** *There do not exist two distinct accepted successors of S_n that allocate the same spendable balance to different recipients.*

Sketch. Conflicting successors would both consume the same parent but assign identical spend power to different recipients; acceptance of both contradicts Tripwire or hash collision resistance. □

Theorem 4 (Global Supply Conservation)**.** *For any set of bilateral transactions across the entire network, the sum of all Δ values across all parties is zero, preserving total token supply without requiring global synchronization or consensus.*

Proof. Consider a bilateral transaction between parties A and B with transfer amount α. By construction, $\Delta_A = -\alpha$ and $\Delta_B = +\alpha$, so $\Delta_A + \Delta_B = 0$.

For any set of transactions forming a connected graph of bilateral relationships, each transaction contributes zero net supply change when summed across its participants. Since each token movement affects exactly two parties with equal and opposite Δ values, the global sum $\sum_{\text{all parties}} \Delta = 0$.

This conservation holds without global synchronization because each bilateral relationship maintains its own invariant locally, and the global property emerges from the bilateral structure itself. □

8.1 Deterministic Limbo Vault Invariants

Let a vault be $V = (L, C, H)$. The unlocking secret emerges only upon stitched proof–of–completion σ:

$$\mathrm{sk}_V = H(L \| C \| \sigma). \tag{11}$$

Without σ, recovering sk_V is negligible in λ.

9 Context Policy & Token Anchors (CPTA)

This section specifies a *deterministic, immutable* policy object used to create tokens and to constrain subsequent token operations under DSM. A CPTA is a single canonical byte string with a BLAKE3 commitment; clients cache the object locally and may fetch its full bytes from any storage node by commitment digest. Enforcement is entirely device–local via *inclusion proofs* and receipt predicates; no external executor is trusted.

Goals. (1) Deterministic structure for the parts DSM must verify directly (ticker, alias, decimals, emission mode, authority, allowlists). (2) An *expressive* hook for external commitments (eligibility sets, deposit ledgers, registries), referenced by hash and proven via receipts, without importing foreign runtime.

9.1 Identity, Immutability, and Anchoring

Token genesis G_T. Each token has a *token genesis* $G_T \in \{0,1\}^{256}$, derived by the issuer in a collision-resistant way, e.g.

$$G_T \;=\; H\Big(\texttt{"DSM/token-genesis\textbackslash 0"} \,\|\, G_{\text{issuer}} \,\|\, s_T\Big),$$

where G_{issuer} is the issuer's genesis and s_T is issuer-chosen entropy.

CPTA commitment and policy number. The canonical policy bytes (Sec. 9.3) hash to

$$\text{policy\_commit} := H\Big(\texttt{"DSM/cpta\textbackslash 0"} \,\|\, \texttt{canonical\_cpta\_bytes}\Big).$$

The *policy number* is the first 8 or 16 bytes of policy_commit (hex), used for UI/indexing. *CPTAs are immutable.* Any change yields a new policy_commit and a new G_T (new token).

Anchoring and caching. Receipts that create a token MUST include policy_commit and MAY include an opaque *policy anchor pointer* (e.g., content-addressed hash or hint URI). Clients fetch bytes by digest from any storage node and cache locally; the digest binds the content (no trust in the server).

9.2 Object Model (Structured Core + External Commitments)

A CPTA splits into a **structured core** (deterministically enforced by DSM) and an **external section** (pure commitments—hashes the policy *refers to* but never executes).

Structured core (normative fields).

- **identity:** G_T (32B), `version` (u32).
- **display:** `alias` (UTF-8), `ticker` (A–Z, 2–8 chars), `decimals` (u8; 0 for NFTs).
- **kind:** `FUNGIBLE` | `NFT` | `SBT` (non-transferable).
- **supply:** `cap` (u128 or ∞); `initial_mint` (u128, optional).
- **emission:** one of:
  a) `STATIC_NONE` (no emissions),
  b) `STATIC_SCHEDULE` (piecewise deterministic schedule),
  c) `DERIVED` (emission steps unlocked by a *deterministic* iteration budget proof; no clocks).
- **authority (mint/burn only):** threshold t-of-N over a *sorted list of issuer genesis IDs*; emissions do *not* require signatures.
- **allowlists (optional):**
  a) `inline_allowlist`: sorted list of recipient genesis IDs (for small sets), or
  b) `allowlist_root`: 32B Merkle root for large sets; claims present a Merkle *inclusion proof.*

  NFT/SBT claims MUST be one-per-genesis unless policy states otherwise.

External section (commitments only).

- `eligibility_anchors[]`: hashes of external datasets that define eligibility (e.g., a deposit ledger digest, a registrar's roster).
- `metadata_anchors[]`: hashes of off-path documents (syllabi, legal terms, branding), informational only.

DSM never executes external data. Receipts *reference* an anchor by hash and supply whatever proof is required by the policy (e.g., a Merkle proof against an anchored root). Verifiers only check digest equality and proof soundness.

9.3 Canonical Commit Form (CPTA)

CBOR (deterministic, normative).

```
cpta = [
  version: uint,                # e.g., 1
  token_genesis: bstr .size 32, # G_T
  alias: tstr,                  # UTF-8
  ticker: tstr,                 # A..Z (2..8)
  decimals: uint,               # 0..18
  kind: uint,                   # 0=FUNGIBLE, 1=NFT, 2=SBT
  supply_cap: uint|null,        # null means "infinite"
  initial_mint: uint|null,

  emission: [
    mode: uint,                 # 0=STATIC_NONE, 1=STATIC_SCHEDULE, 2=DERIVED
    params: bstr                # mode-specific params (CBOR inside)
  ],

  authority: [
    t: uint,                    # threshold t
    signers: [* bstr .size 32]  # sorted issuer genesis IDs (32B each)
  ],

  allowlist: [
    kind: uint,                 # 0=NONE, 1=INLINE, 2=MERKLE_ROOT
    body: any                   # [] of 32B IDs (sorted) | 32B root | null
  ],

  ext: [
    eligibility_anchors: [* bstr .size 32],
    metadata_anchors:    [* bstr .size 32]
  ]
]
```

Encoding must follow RFC 8949 *deterministic* CBOR: definite lengths, fixed field order, and strict types. The CPTA commitment is the BLAKE3-256 of the domain-separated prefix plus the encoded bytes (above).

9.4 dsm_app.proto Additions (Transport Only)

Listing 1: Transport messages for CPTA; commits/verification are produced and checked by the Rust core, not by transport.

```
message CptaPolicy {
  bytes token_genesis = 1; // 32B G_T
  uint32 version = 2; // must match canonical commit
  string alias = 3;
  string ticker = 4;
  uint32 decimals = 5; // 0..18
  enum Kind { FUNGIBLE = 0; NFT = 1; SBT = 2; }
  Kind kind = 6;

  message Emission {
    enum Mode { STATIC_NONE = 0; STATIC_SCHEDULE = 1; DERIVED = 2; }
    Mode mode = 1;
    bytes params_cbor = 2; // opaque; canonicalized inside commit bytes
  }
  Emission emission = 7;

  message Authority {
    uint32 threshold = 1; // t
    repeated bytes genesis_signers = 2; // 32B each, sorted
  }
  Authority authority = 8;

  message Allowlist {
    enum Kind { NONE = 0; INLINE = 1; MERKLE_ROOT = 2; }
    Kind kind = 1;
    repeated bytes inline_genesis = 2; // 32B each, sorted if present
    bytes merkle_root = 3; // 32B if present
  }
  Allowlist allowlist = 9;

  repeated bytes eligibility_anchors = 10; // 32B digests
  repeated bytes metadata_anchors = 11; // 32B digests

  bytes policy_commit = 12; // 32B; MUST equal canonical commit of CBOR form
  string policy_pointer = 13; // optional: retrieval hint (non-authoritative)
}
```

```
message TokenCreate {
  bytes issuer_genesis = 1; // 32B
  CptaPolicy policy = 2; // full policy (transport view)
}

message TokenOp {
  bytes token_genesis = 1; // 32B G_T
  oneof op {
    Transfer transfer = 2;
    Mint mint = 3;
    Burn burn = 4;
    EmissionStep emission_step = 5;
    NftClaim nft_claim = 6;
  }
  bytes policy_commit = 10; // 32B; binds the op to this immutable CPTA
}

message Transfer {
  bytes from_genesis = 1; // 32B
  bytes to_genesis = 2; // 32B
  uint128 amount = 3; // as bytes/decimal split in practice
}

message Mint {
  uint128 amount = 1;
  repeated bytes signer_genesis = 2; // t-of-N cosigners per CPTA.authority
}

message Burn {
  uint128 amount = 1;
  repeated bytes signer_genesis = 2; // t-of-N cosigners per CPTA.authority
}

message EmissionStep {
  uint64 step_index = 1; // sequential, deterministic
  bytes proof_cbor = 2; // schedule/derived proof; no clocks
}

message NftClaim {
  bytes claimant_genesis = 1; // 32B
  bytes allowlist_merkle_proof = 2; // if allowlist.kind == MERKLE_ROOT
  bytes eligibility_evidence = 3; // CBOR bundle referencing eligibility_anchors
      []
}
```

9.5 Acceptance Predicates (Creation and Ops)

Token creation (normative). A `TokenCreate` succeeds iff:

1. `policy.policy_commit` equals the BLAKE3 of the canonical CPTA CBOR;
2. `policy.token_genesis` $= G_T$ and is unique;
3. `ticker` and `alias` satisfy format constraints; `decimals` matches `kind`;
4. any `initial_mint` is applied atomically under the same receipt and (if > 0) meets `authority` (t-of-N).

Transfers. As in Sec. 8: $\Delta_{\text{sender}} = -\alpha$, $\Delta_{\text{recipient}} = +\alpha$, non-negativity preserved; `SBT` MUST reject any transfer (non-transferable).

Mint/Burn. Require `authority.threshold` distinct cosignatures from `authority.signers` (each a genesis ID). Enforce `supply_cap` and balance non-negativity.

Emissions.

- `STATIC_SCHEDULE`: verify `proof_cbor` matches the schedule parameters (e.g., piece i, amount$_i$). No human signatures.
- `DERIVED`: verify a *deterministic iteration budget* proof (BLAKE3 work units) bound to G_T and the previous emission step; no clocks.

In both modes, the `step_index` must advance by +1 and total supply must not exceed `supply_cap` (if finite).

NFT/SBT claims and allowlists. If `allowlist.kind` = `INLINE`, check the claimant's genesis is in the inline list (binary search over the sorted set). If `MERKLE_ROOT`, verify the Merkle *inclusion proof* against the anchored root; enforce one-per-genesis (device-local counter over stitched receipts). If `eligibility_anchors[]` are present, the receipt MUST carry evidence CBOR that reduces to an equality/containment proof against at least one anchored digest (e.g., a deposit-ledger commitment).

Binding to policy. All `TokenOps` MUST include policy_commit; verifiers reject if it differs from the token's creation policy_commit.

9.6 Worked Examples (Normative Patterns)

9.6.1 A. University Degree (Non-Web3 Credential, SBT)

Intent. Issue a non-transferable credential NFT (SBT) to each graduate.

CPTA (core).

- `kind`=`SBT`, `decimals`=0, `supply_cap`=∞, `emission`=`STATIC_NONE`, `authority`=t=2 of N=3 (Registrar, Provost, Records Office).
- `allowlist.kind`=`MERKLE_ROOT` where the root commits the graduating cohort's *genesis IDs* (sorted).
- `eligibility_anchors[]` includes the digest of the university's *final award roster* (export hash).

Issuance. Each student submits `NftClaim` with (i) a Merkle proof against the cohort root, and optionally (ii) evidence CBOR proving inclusion in the award-roster anchor. The receipt is stitched bilaterally (student$\leftrightarrow$university device), and the SBT is minted without human signatures (authority unused because `emission` = `STATIC_NONE` and claim gates via allowlist).

Transfer. Rejected (non-transferable). Revocation can be modeled as a mutually exclusive *burn* branch requiring authority t-of-N.

9.6.2 B. Community Credit (Fungible, Deterministic Emission)

Intent. Local currency with deterministic, signer-free emissions.

CPTA (core).

- `kind`=`FUNGIBLE`, `decimals`=2, `supply_cap`=10^9 units, `initial_mint`=0.
- `emission`=`STATIC_SCHEDULE` with CBOR params: piecewise map $\{(\mathsf{step}_i, \mathsf{amount}_i)\}$ summing to the cap.
- `authority`=t=0 (no discretionary mint/burn).
- no allowlist (any genesis can hold/transfer).

Emission. Any party may submit `EmissionStep{step,proof}`; the proof deterministically binds G_T, prior `step`, and the schedule params. If valid, the step mints $\texttt{amount}_i$ to the emission sink address defined by the policy (e.g., a community treasury genesis).

9.7 Interplay with External Commitments (Illustrative)

Deposit-gated NFT allowlist. If a project requires "pre-deposit $\rightarrow$ allowlist", publish an anchor digest of the *deposit ledger* (e.g., a Merkle root over tuples $(\text{genesis}, \text{amount})$). Claims include a Merkle proof and a predicate ("amount $\geq$ threshold") encoded in evidence CBOR. Verifiers check: (1) anchor hash matches the CPTA's `eligibility_anchors[]`; (2) the proof reduces to the ledger root; (3) predicate holds. No server is trusted; only digests and proofs.

9.8 Storage, Lookup, and Replication

Storage nodes index CPTAs by policy_commit and (optionally) by `ticker` and `alias` (non-authoritative). Devices cache CPTAs locally; any node can serve bytes (clients verify by digest). For pointer-style retrieval, `policy_pointer` is a hint only—the digest is the source of truth.

Short identifiers. UI and tables may display the *policy number* (first 8–16 bytes of policy_commit, hex) and `ticker/alias` for human legibility; protocol logic *always* uses full 32B digests.

9.9 Security and Determinism

- **Immutability:** the CPTA is frozen by policy_commit. Any change defines a new token (new G_T).
- **No clocks:** emissions and claims rely on deterministic schedules, iteration-budget proofs, and Merkle *inclusion proofs.*
- **Least authority:** routine transfers need no signers; only mint/burn require threshold signatures; emissions require none.
- **Local enforceability:** all predicates (allowlist, caps, authority thresholds, schedules) are checked in the stitched receipt verification path; failure causes rejection without network calls.

10 Storage Node Regulation and Incentives

The decentralized storage layer is governed by cryptography and economics, not discretion. Storage nodes (a) serve object availability (Device Tree, Per–Device SMT aggregates, and `b0x` messages), (b) provide inclusion proofs on demand, and (c) submit to objective audits. Sustainability follows from the subscription model (Sec. 7.1); incentive alignment follows from hardware-bound identity and staking.

10.1 Hardware-Bound Cryptographic Identity

Each node derives a non-forgeable identity from its genesis context and DBRW binding:

$$\text{nodeID} \;=\; H\Big(\text{“DSM/node”} \,\|\, G \,\|\, K_{\text{DBRW}}^{(\text{node})}\Big),$$

where G is the network genesis commitment and $K_{\text{DBRW}}^{(\text{node})}$ is the dual-binding materialized from the node's hardware entropy and execution environment. This makes Sybil creation economically costly: duplicating *distinct* nodeIDs requires distinct hardware and environments.

10.2 Admission, Staking, and Service Commitments

To participate, a node presents:

1. an inclusion proof in the *Node Registry SMT* root R_{nodes},
2. a *non-inclusion* (zero-leaf) proof in the *Node Denylist SMT* root R_{deny}, and
3. a stake commitment $\mathcal{S}$ in the native DSM token, bound to nodeID.

All three are verifiable from canonical digests that advance via stitched receipts. Service obligations are *objective*:

- **Availability:** serve Device Tree snapshots and Per–Device SMT aggregate heads with valid inclusion proofs for requested keys.
- **Delivery:** accept and relay `b0x`-prefixed messages (Sec. 4.2, Implementation Notes) and make them retrievable until consumed.
- **Auditability:** prove storage of randomly sampled items via inclusion proofs and freshness checks tied to the current registry roots.

10.3 Normative Audit Procedures and Evidence Handling

Storage audits in DSM are entirely protocol-native and byte-driven. Storage nodes remain dumb and signature-free; all enforcement flows from mirrored protobuf bytes, deterministic hashes, and device-signed receipts.

We fix the following normative audit structure:

1. **Replica placement and PaidK gate.** For any object address addr, a deterministic placement function maps

$$\mathsf{addr} \longmapsto \text{Replicas}(\mathsf{addr}) \subseteq \{\text{nodeID}\}$$

   using a Fisher–Yates permutation seeded from

$$H\Big(\text{“DSM/place\textbackslash 0”} \| \mathsf{addr}\Big)$$

   and the current Node Registry vector. Reads are subject to a PaidK gate: a client *accepts* the content at addr only if at least K distinct nodes in Replicas(addr) return identical bytes that re-derive addr under the object-domain hash. Any disagreement between replicas for the same addr is objective evidence of misbehavior, and every accepted read contributes to the evidence surface.

2. **Node Storage SMT and ByteCommit mirroring.** Each operator maintains a Node Storage sparse Merkle tree (SMT) over the addresses it serves. After applying all PUT/DELETEs for cycle t, it computes

$$R_t^{\text{node}} = \mathsf{SMT}(\text{NodeStorage}_t), \quad \mathsf{bytes\_used}_t = \sum_{\ell \in \text{leaves}_t} \text{len}(\ell).$$

   It then emits a `ByteCommitV3` message

$$B_t = \Big(\mathsf{node\_id},\ t,\ R_t^{\text{node}},\ \mathsf{bytes\_used}_t,\ \mathsf{parent\_digest}\Big),$$

   encoded via deterministic protobuf under domain tag `"DSM/bytecommit\0"`. The mirrored bytes of B_t are addressed as

$$\mathsf{addr}_B(t) = H\Big(\text{“DSM/obj-bytecommit\textbackslash 0”} \,\|\, \mathsf{node\_id} \,\|\, t \,\|\, H(B_t)\Big).$$

   A verifier accepts B_t if and only if:

   - the protobuf encoding is deterministic and domain-separated;
   - the digest $H(\text{“DSM/bytecommit\textbackslash 0”}\|B_t)$ matches the mirrored address;
   - the parent link is valid ($H(B_{t-1})$ for $t > 0$ or 0^{32} for $t = 0$);
   - R_t^{node} validates as the Node Storage SMT root; and
   - at least q identical copies of B_t are fetched from the replica set determined by the active registry.

No storage-node signatures are ever consulted; all checks are hash- and SMT-based and reconstructable from mirrored bytes.

3. **Capacity signals and registry movement.** Over cycles, the public series

$$u_t = \mathsf{bytes\_used}_t / C$$

(for fixed partition capacity C) is reconstructable from mirrored $\{B_t\}$. Nodes may publish Up/Down capacity signals that reference a window of committed ByteCommits by hash. A signal is accepted if and only if all referenced B_j are valid and the corresponding u_j lie above (Up) or below (Down) configured thresholds. Node position changes in the registry—how many entries to add or remove—are a pure function of these accepted signals and the discovery window; there is no voting, scheduling, or discretionary governance.

4. **Evidence records and Node Denylist SMT.** When a client or auditor detects misbehavior, it constructs a minimal evidence record E summarizing:

   - the offending nodeID and cycle indices;
   - the conflicting storage bytes or ByteCommit bodies; and
   - the expected values implied by the DSM rules (placement, PaidK, and SMT structure).

   The record is summarized by a domain-separated digest

$$h_E = H\Big(\text{“DSM/evidence\textbackslash 0”} \parallel \mathsf{ProtoDet}(E)\Big),$$

which is inserted as a leaf in the Node Denylist SMT with a pointer to the associated stake DLV. Any third party with access to the same bytes can recompute h_E and verify that the node violated deterministic predicates; no storage-node signatures are required.

5. **DrainProof and exit.** A node that wishes to exit publishes ByteCommits whose Node Storage SMT reflects an empty (or near-empty) partition for a configured number of consecutive cycles. This sequence of (B_t, R_t^{node}) pairs constitutes a DrainProof. Stake unlock is then a deterministic predicate over this proof and the Node Registry state, not a governance decision.

All audit evidence is reconstructable from mirrored protobuf bytes and hashes. Storage nodes remain dumb and signature-free; only end devices ever sign stitched receipts.

10.4 Dominant-Strategy Compliance

Let p_d be the probability that a deviation is detected over a discovery window W (measured in cycle indices), given that clients and auditors verify:

- the mirrored `ByteCommitV3` chain for each node,
- replica consistency under the PaidK gate for sampled addresses, and

- the validity of any capacity signals that affect registry position.

Because all of these objects are mirrored deterministically, any sustained deviation (dropping objects, serving inconsistent bytes, lying about capacity) eventually appears as one of:

1. an invalid or missing ByteCommit for some cycle,
2. a PaidK violation (replicas disagreeing on bytes for the same address),
3. an invalid capacity signal whose referenced ByteCommits fail checks.

Let F denote the economic penalty of being placed in the Node Denylist (slashed stake plus foregone future subscription revenue), and let G bound the short-term gain from deviating (e.g., skipping replicas or overstating capacity). DSM chooses parameters

$$(N, K, C, \text{pricing}, \text{stake size}, W)$$

such that

$$p_d \cdot F \;>\; G.$$

Then the expected payoff of deviation is strictly negative:

$$\mathbb{E}[\text{deviate}] \;=\; (1-p_d)\, G \;-\; p_d\, F \;<\; 0.$$

Compliance is thus a dominant strategy. This conclusion does not rely on committees, timestamps, or discretionary governance. All inputs to p_d, F, and G are deterministically derivable from:

- mirrored Node Storage SMT roots and `ByteCommitV3` messages,
- the active Node Registry and Node Denylist SMT roots, and
- stake and subscription pricing recorded in DLVs.

Any verifier with access to the same bytes reaches the same slashing and admission decisions.

11 Post–Quantum Key Evolution and Transport

DSM uses Kyber KEM to derive per–transition step material and SPHINCS+ to authenticate receipts. At each step:

$$(\mathsf{ct}, \mathsf{ss}) = \text{KyberEnc}(\text{pk}_{\text{recipient}}), \quad k_{\text{step}} = H(\mathsf{ss}), \tag{12}$$

where H is BLAKE3–256 with domain separation. Second–preimage resistance of chained commitments prevents forks; SPHINCS+ ensures non–repudiation.

11.1 SPHINCS+ Ephemeral Keys Chained to Parent (Clockless)

Signatures (normative). Parameter set: SPHINCS+ BLAKE3, level = NIST Category 5, variant = ‘f‘ (fast). The DSM implementation uses BLAKE3 for all hash, PRF, and thash operations within SPHINCS+ (not SHAKE). Receipts *MUST* admit a hard maximum serialized size ≤ 128 KiB (including two signatures and three proofs). Submissions exceeding the cap are invalid and *MUST* be rejected prior to proof verification.

Key derivation (normative). Let K_{DBRW} be the DBRW binding (Sec. 12). K_{DBRW} *MUST NEVER be serialized, logged, or included in any commitment.* Derive a master seed using HKDF–BLAKE3:

$$S_{\text{master}} = \text{HKDF-Extract}_{\text{BLAKE3}}(\text{salt} = \texttt{"DSM/dev\textbackslash 0"},\ \text{IKM} = G \parallel \text{DevID} \parallel K_{\text{DBRW}} \parallel s_0)\,, \tag{13}$$

and an attestation key $(\mathsf{AK}_{\text{sk}}, \mathsf{AK}_{\text{pk}}) \leftarrow \text{SPHINCS+.KeyGen}(S_{\text{master}})$.

Given parent h_n and precommit C_{pre}, derive the per–step seed

$$E_{n+1} = \text{HKDF}_{\text{BLAKE3}}(\texttt{"DSM/ek\textbackslash 0"},\ h_n \parallel C_{\text{pre}} \parallel k_{\text{step}} \parallel K_{\text{DBRW}})\,, \tag{14}$$

then generate the ephemeral keypair $(\mathsf{EK}^{\text{sk}}_{n+1}, \mathsf{EK}^{\text{pk}}_{n+1})$.

Ephemeral certification (normative). Define the certification hash with domain separation

$$H_{\text{ek-cert}}(X) := \text{BLAKE3}\Big(\texttt{"DSM/ek-cert\textbackslash 0"} \parallel X\Big).$$

Certify the new key with the previous signer (AK for $n{=}0$, else EK_n):

$$\mathsf{cert}_{n+1} = \text{Sign}_{\mathsf{SK}_n}\Big(H_{\text{ek-cert}}\Big(\mathsf{EK}^{\text{pk}}_{n+1} \parallel h_n\Big)\Big). \tag{15}$$

Sign the receipt body with $\mathsf{EK}^{\text{sk}}_{n+1}$.

Verification (normative). Verification replays the chain of certificates back to AK_{pk} and checks inclusion proofs (Sec. 4.2).

11.2 Identity Pre–commitment

Let P_0 be a provisioning seed; define $P_i = H(P_{i-1})$ with $H =$ BLAKE3–256 under a fixed domain tag. Under collision resistance, adversaries cannot forge a different identity chain consistent with $\{P_i\}$ without breaking P_0. For transport, commitments may be sealed via Kyber and verified upon decryption by checking $H(S_n \| P)$.

12 Dual–Binding Random Walk (DBRW)

Definition 1 (Hardware Entropy). $H(d) \in \{0,1\}^n$ extracts device–specific microarchitectural entropy.

Definition 2 (Environment Fingerprint). $E(e) \in \{0,1\}^m$ fingerprints the execution environment.

Definition 3 (Dual–Binding). $K_{\text{DBRW}} = H\Big(H(d) \| E(e) \| s_{\text{device}}\Big)$,

where here H is BLAKE3–256 with a DBRW-specific domain tag `"DSM/dbrw-bind\0"`; the per-device salt s_{device} ensures uniqueness even for similar hardware or environments.

Theorem 5 (Binding Inseparability). *Given K_{DBRW} and collision resistance of H, it is infeasible to find $(h', e', s') \neq (h, e, s)$ such that $H(h' \| e' \| s') = H(h \| e \| s)$. The per-device salt s_{device} prevents correlation attacks by ensuring unique bindings even when hardware entropy or environment fingerprints are similar across devices.*

DBRW advances without clocks:

$$\rho_i = H(C_{i-1} \| K_{\text{DBRW}} \| \text{“}\rho\text{”}), \qquad C_i = H(C_{i-1} \| K_{\text{DBRW}} \| \rho_i \| N_i), \tag{16}$$

with nonce N_i. Mixing K_{DBRW} into key derivations binds all signatures to the device and environment.

Privacy Rule: DBRW bindings MUST NOT be used for user identification, tracking, or correlation. DBRW exists solely for anti-cloning protection and device binding; all user-facing operations use DevID and genesis-based addressing.

13 Offline Recovery Protocol

After each accepted receipt, the device writes an encrypted capsule

$$\text{Capt} = \text{Enc}_{K_R,\ \text{nonce}=c}\Big(r_t,\ \text{Meta},\ \{(\text{DevID}^{(8)}, h^{A\leftrightarrow\text{Dev}})\},\ \text{Roll}_t,\ \text{challenge}\Big), \tag{17}$$

where c is a monotone counter (local to the capsule stream), r_t is the Per–Device SMT root, $\text{DevID}^{(8)}$ are 8–byte device digests, $h^{A\leftrightarrow\text{Dev}}$ current tips, Roll_t an accumulator over accepted receipts, and challenge is a fresh random nonce bound to the capsule creation context. The ring key K_R derives from a 24–word mnemonic via Argon2id. Update

$$\text{Roll}_{t+1} = H\Big(\text{“RM”} \| \text{Roll}_t \| H(\text{Receipt}) \| \text{DevID}^{(8)} \| h'^{A\leftrightarrow\text{Dev}}\Big). \tag{18}$$

Tombstone and Succession. On loss, decrypt to $(r^\star, \{(\mathrm{DevID}^{(8)}, h)\}, \mathrm{Roll}^\star)$. Broadcast **Tombstone (TR)** to invalidate the old binding at $r^\star$, then **Succession (SR)** to bind the new device; SR references TR by hash and is valid only if TR is active. Resume per–relationship by proposing successors adjacent to the stored parent h; acceptance is by unique parent consumption.

Security. Receipt uniqueness ensures at most one accepted successor per parent. AEAD + mnemonic hardness protect the capsule. Fresh challenge nonces prevent capsule replay attacks by binding each capsule to its creation context. After TR/SR, the old device cannot extend state. The rollup binds the recovered history.

13.1 Modal Synchronization Precedence

If A performs a physical offline transaction with B, then B must synchronize that result (i.e., incorporate the stitched receipt) before initiating a new offline transaction with A; otherwise B would attempt to consume a different parent.

14 Geo–Emissions and Geometry–Bounded Attestation

Geo–emissions in DSM are deterministic, geometry–bounded token revelations. They are created and claimed without clocks, timestamps, or round–trip gossip. Devices operate against a pinned spawn registry, quantized geocells, DBRW hardware/environment binding, and a per–step SPHINCS$^+$ key schedule. Claims are constructed offline and later published as immutable *emission capsules*, which storage nodes verify statelessly.

14.1 Spawn Registry and Emission Capsules

Let $\mathcal{R}$ denote the global spawn registry with Merkle root $\mathsf{Root}_{\text{spawn}}$. Each spawn entry is:

$$\mathsf{spawn} = (\mathsf{spawn\_id},\ \mathsf{region\_id},\ \mathsf{lat},\ \mathsf{lon},\ R,\ \mathsf{capacity},\ \mathsf{policy\_commit})\,,$$

where R is the maximum claim radius and $\mathsf{capacity} \in \mathbb{N}$ is the maximum number of successful claims.

Devices store $\mathsf{Root}_{\text{spawn}}$ locally and obtain an inclusion proof π_{spawn} for any $\mathsf{spawn\_id}$ they wish to target.

An *emission capsule* is a one–shot, geometry–bounded claim:

$$\mathsf{cap}_{\text{emit}} = (\mathsf{spawn\_id},\ \mathsf{device\_id},\ \mathsf{chain\_tip},\ L,\ W,\ C,\ \sigma,\ \pi_{\text{spawn}},\ \mathsf{policy\_commit})\,.$$

Here:

- $\mathsf{device\_id}$ is the DSM device identity.
- $\mathsf{chain\_tip}$ is the current bilateral chain tip for the device–storage relationship.
- L is a geometry commitment derived from a quantized GNSS geocell.
- W is a DBRW witness tied to hardware and environment.
- C is the capsule commitment.
- σ is a per–step SPHINCS$^+$ signature under the ephemeral key for $\mathsf{chain\_tip}$.
- π_{spawn} proves $\mathsf{spawn\_id}$ under $\mathsf{Root}_{\text{spawn}}$.
- $\mathsf{policy\_commit}$ binds to the CPTA/token policy used for minting.

14.2 Quantized Geocells and DBRW Binding

Let ℓ be the device GNSS location. For a fixed network–wide geocell size Δ (e.g. 3–5 m), define:

$$\mathsf{cell} = \mathsf{Quantize}(\ell, \Delta).$$

Encode GNSS quality and fix a geometry commitment:

$$L = \mathsf{H}\Big(\text{“DSM/geo/loc\textbackslash 0”} \parallel \mathsf{cell} \parallel \eta\Big),$$

where η encodes GNSS quality (e.g. HDOP class).

DBRW produces a hardware/environment witness W, and DSM commits to it:

$$W^\star = \mathsf{H}\Big(\text{“DSM/dbrw\textbackslash 0”} \parallel W\Big).$$

14.3 Capsule Commitment and Signature

The emission capsule commitment binds spawn, device, state, geometry, and DBRW witness:

$$C = \mathsf{H}\Big(\text{“DSM/geo/capsule\textbackslash 0”} \parallel \mathsf{spawn\_id} \parallel \mathsf{device\_id} \parallel \mathsf{chain\_tip} \parallel L \parallel W^\star\Big).$$

Let EK_n be the SPHINCS$^+$ ephemeral key corresponding to $\mathsf{chain\_tip}$. The device signs:

$$\sigma = \mathsf{Sign}_{\text{SPHINCS}^+}(\text{EK}_n, C).$$

The complete capsule is then fixed as $\mathsf{cap}_{\text{emit}}$ above. It is immutable: any change to geometry, DBRW state, or chain tip changes C and invalidates σ.

14.4 Local GNSS Integrity Gate (LGIG)

Let $\mathsf{dist}(\cdot,\cdot)$ be the geodesic distance on the Earth model. A device may only target a spawn if

$$\mathsf{dist}\Big(\mathsf{cell}, (\mathsf{lat}, \mathsf{lon})\Big) \le R - \varepsilon,$$

for a fixed safety margin ε determined by GNSS quality class.

Definition 4 (Local GNSS Integrity Gate (LGIG))**.** A capsule $\mathsf{cap}_{\text{emit}}$ satisfies LGIG if:

1. π_{spawn} verifies under $\mathsf{Root}_{\text{spawn}}$ for the advertised $\mathsf{spawn\_id}$.
2. L is correctly derived from a quantized geocell within $R - \varepsilon$ of the spawn center, using the network–wide cell size Δ and the GNSS quality encoding η.
3. W satisfies $\mathsf{Accept}_{\text{DBRW}}(W;\Theta)$ for the fixed DBRW parameter set Θ (hardware/environment binding holds).
4. C recomputes as

   $$C = \mathsf{H}\Big(\text{“DSM/geo/capsule\textbackslash 0”} \parallel \mathsf{spawn\_id} \parallel \mathsf{device\_id} \parallel \mathsf{chain\_tip} \parallel L \parallel W^\star\Big),$$

   where $W^\star = \mathsf{H}\Big(\text{“DSM/dbrw\textbackslash 0”} \parallel W\Big)$.
5. σ verifies as a valid SPHINCS$^+$ signature on C under the ephemeral key EK_n corresponding to the bilateral $\mathsf{chain\_tip}$.

LGIG is purely local: no clocks, no network, and no external oracle are required to construct a valid capsule.

14.5 Asynchronous Publication and CRDT Selection

Capsules are created offline and marked *Pending*. Upon connectivity, the device publishes $\mathsf{cap}_{\text{emit}}$ to one or more storage nodes. Each node verifies LGIG and, if valid, inserts C into its local CRDT state for the given $\mathsf{spawn\_id}$.

Let $\mathcal{C}_{\mathsf{spawn}}$ be the set of valid commitments C for a spawn observed by a storage node. Define a deterministic first–writer–wins (FWW) selector:

$$\mathsf{FWW}_c(\mathcal{C}_{\mathsf{spawn}}) = \text{the } c \text{ lexicographically smallest } C \in \mathcal{C}_{\mathsf{spawn}},$$

where $c = \mathsf{capacity}$.

Only capsules whose C lies in $\mathsf{FWW}_c(\mathcal{C}_{\mathsf{spawn}})$ are eligible for minting. Because FWW is pure set logic on commitments, no timestamps or global ordering are required.

14.6 Replication and Finalization

Minting is a deterministic state transition:

$$\mathsf{DLV}_{\text{root}} \xrightarrow{\mathsf{cap}_{\text{emit}}} \mathsf{DLV}'_{\text{root}},$$

where the emission amount and target CPTA are derived from policy_commit. The capsule is marked *Consumed* once minted. Republishing the same $\mathsf{cap}_{\text{emit}}$ has no effect (idempotence).

Because devices sign only C and storage nodes only perform set operations on commitments, the entire geo–emission pipeline is:

- **Offline–first:** Capsule construction requires no network.
- **Deterministic:** Selection is FWW over C; no block height, time, or consensus is needed.
- **Bounded:** Capacity c and radius R are fixed in the spawn registry.
- **Non–replayable:** chain_tip and device_id bind each capsule to a single relationship.

Theorem 6 (Deterministic Capacity Enforcement)**.** *For any spawn with capacity c, across all honest storage nodes, only capsules with commitments in* $\mathsf{FWW}_c(\mathcal{C}_{spawn})$ *can be minted, independent of message ordering, timing, or network topology.* □

15 Security Analysis and System Properties

15.1 Why the CAP Theorem Does Not Apply

CAP presumes a *single, globally shared* object whose operations must trade off consistency, availability, and partition tolerance. DSM rejects that premise: there is no monolithic global state. Instead, DSM is a collection of independent bilateral relationships, each with its own straight hash chain and Per–Device SMT head, stitched by countersigned receipts.

15.2 Local Predicates (Per Relationship)

Let $R_{i,j}$ denote the relationship domain between devices i and j. We define, *locally*:

$$\begin{aligned}
C_{i,j} &\Leftrightarrow \text{all receipts on } R_{i,j} \text{ verify (signatures + inclusion proofs) and the chain has no forked successor,}\\
A_{i,j} &\Leftrightarrow \text{each valid operation on } R_{i,j} \text{ returns a deterministic non-error response (online or offline),}\\
P_{i,j} &\Leftrightarrow \text{network partitions only transition } R_{i,j} \text{ to offline mode; unrelated } R_{k,\ell} \text{ unaffected.}
\end{aligned}$$

These predicates depend solely on hash adjacency and inclusion proofs; they do not reference clocks or a global height.

15.3 Localized Feasibility (No Global Trade-off)

Theorem 7 (Per-Relationship CAP Feasibility). *For every (i, j), DSM simultaneously satisfies $C_{i,j}$, $A_{i,j}$, and $P_{i,j}$.*

Proof. Consistency: stitched receipts and collision resistance eliminate acceptable double successors for the same parent (Tripwire), so $C_{i,j}$ holds. Availability: each operation either (a) completes online via `b0x` delivery into the counterparty's Per–Device SMT pipeline, or (b) completes offline via live co-signing; in both cases the response is deterministic, establishing $A_{i,j}$. Partition tolerance: a partition only affects the ability of *that* pair to synchronize; all other $R_{k,\ell}$ continue, so $P_{i,j}$ holds. Because DSM does not attempt to maintain a global shared object, the global CAP trade-off never arises. □

15.4 System-Level Consequence

Since the system is the disjoint union of $\{R_{i,j}\}$, the classical CAP impossibility is out of scope. DSM achieves consistency, availability, and partition tolerance *within each relationship domain*—the only domain where the predicates are semantically meaningful in DSM.

15.5 Bifurcation Resistance and Pre–Sign Commitments

Mandatory pre–sign commitments C_{pre} lock parameters; forging conflicting successors requires a hash collision or signature forgery. Offline bilateral exchanges realize the same security via proximity channels.

15.6 Non–Repudiation and Causal Ordering

Countersigned receipts are undeniable; causal ordering emerges from parent embedding and Per–Device/Device Tree *inclusion proofs*.

15.7 Anti–Cloning Guarantees

DBRW binds state to both hardware and environment; without K_{DBRW}, extending state is infeasible.

15.8 Offline Liveness and Recovery

The capsule + TR/SR scheme enables immediate resumption after device loss; per–relationship parents allow constructing successors without replaying history; the rollup ensures integrity and prevents double spending by the old device.

15.9 Additional Guarantees

Auditing can enumerate stitched digests between indices; proofs remain logarithmic. Reputation or rate–limits can be computed from local counters or windows orthogonal to acceptance rules.

16 System Architecture and Implementation

This section is a drop–in replacement that specifies the complete DSM system design for the current mobile–first SDK. Android (NDK/JNI) is the reference target, but the SDK is defined as cross–platform. It ties the cryptographic model to concrete code structure, transport, and mobile integration, while retaining all prior protocol invariants (no global consensus, no wall clocks or heights, bilateral isolation, inclusion proofs only).

16.1 Codebase Layout and Roles

Rust Core (`dsm_core`) Single source of truth for all state transition rules, cryptography, and verification. It is transport–agnostic and exposes a stable C ABI for mobile via NDK and other bindings.

- **`core/`**: Device/Genesis creation, bilateral chain logic, Per–Device SMT maintenance.
- **`crypto/`**: Post–quantum primitives (Kyber KEM, SPHINCS+ signatures), BLAKE3, HKDF–BLAKE3, Argon2id.
- **`bilateral/`**: Offline co–sign flow, stitched receipts, inclusion proofs.
- **`unilateral/`**: Online single–party submits over `b0x[...]` addressing.
- **`recovery/`**: Tombstone/Succession, capsule (AEAD) handling, DLV.
- **`bridge/`**: FFI surface for NDK/JNI and (optionally) WASM/desktop bindings.

16.2 SDK Architecture and Build Targets

Scope and authority (normative). The **Rust protocol core crate** `dsm_core` is the sole execution authority for canonical commit bytes, acceptance predicates, Merkle proof verification, SMT replace semantics, signatures, and KDFs. Platform SDKs and bindings are non-authoritative shims that *MUST* delegate these operations to `dsm_core` and *MUST NOT* re-implement or alter canonical encodings or predicates.

16.2.1 SDK repository (language-agnostic)

The **SDK** refers to the cross–platform repository `DSM_SDK`, which packages bindings, transport schemas, and developer tooling around `dsm_core`. The SDK is not tied to a single platform; Android is one build target alongside iOS, WebAssembly, and desktop.

16.2.2 Android target (NDK/JNI) and app

The Rust core compiles into an Android *native* shared library (NDK). Kotlin/Java call into Rust via JNI. The Android app may embed a React/TypeScript WebView UI and route all requests through a single auditable bridge.

- **NDK:** `cargo-ndk` builds `libdsm.so` for all target ABIs; symbols are C–ABI stable and versioned.
- **JNI wrapper:** `DsmNativeWrapper.kt` exposes a minimal surface (`init_device`, `submit_online`, `co_sign_offline`, `prove_incl`, `get_per_device_root`).
- **Hardware shim:** Kotlin mediates BLE/NFC for offline and capsule flows; `dsm_core` never touches Android SDK objects directly.
- **WebView bridge:** the frontend emits/receives length–prefixed binary Protobuf envelopes (e.g. `Uint8Array`) through a single bridge. Kotlin converts between WebView binary buffers and Rust FFI buffers. No JSON or base64 is used on the protocol path.

16.2.3 React/TypeScript frontend

UI-only. Protobuf types are generated for TypeScript to ensure schema parity with Rust. Commits and signatures are always computed by `dsm_core`, never in the UI. The frontend manipulates opaque binary envelopes and human readable views; it never re-derives protocol hashes.

16.2.4 Build matrix and targets

- **Core:** `dsm_core` (Rust) is the single execution engine.
- **SDK repo:** `DSM_SDK` publishes bindings for Android (NDK/JNI), iOS (`.a`/`.xcframework`), Web (WASM), and desktop (static/shared libs).
- **Android:** one build target in the matrix; the SDK is defined independently of Android specifics.

16.2.5 Binding constraints (normative)

- Bindings *MUST* call `dsm_core` for: canonical commit emission, receipt verification, Merkle proofs, SMT replace, signature creation/verification, and all KDFs (HKDF–BLAKE3).
- Bindings *MUST NOT* hash or sign Protobuf bytes directly; they *MUST* pass canonical commit bytes emitted by `dsm_core` (see Sec. 4.2.1).
- Versioning is pinned: application code depends on `DSM_SDK@v`, which pins `dsm_core@v`; semantic upgrades are coordinated.
- JSON and base64 are forbidden on the protocol path (wire and acceptance); they may be used only for non-normative debugging output.

Storage Nodes Storage nodes are *dumb, signature–free* persistence surfaces. They store Device Tree material, Per–Device SMT mirrors (aggregated), `b0x` messages, ByteCommit chains, and recovery capsules. They maintain state (Node Storage SMT, ByteCommit history) but *do not* evaluate acceptance predicates. All validation is device–side; nodes simply persist and serve bytes, enabling censorship resistance through replication and client–side verification.

16.3 Two Merkle Structures (No Renames)

Device Tree (standard Merkle, normative). A standard Merkle tree whose root is the Device Tree root R_G, constructed from the owner's device identifiers. It binds *every* device of the same owner to R_G and is replicated on all user devices and storage nodes.

Leaves. The leaves are the 32-byte DevID values sorted lexicographically (big-endian byte order).

Internal nodes. For left child L and right child R,

$$H_{\text{dev}}(L, R) := \text{BLAKE3}\Big(\texttt{"DSM/dev-merkle\textbackslash 0"} \parallel L \parallel R\Big).$$

Empty tree root.

$$R_G^{\emptyset} := \text{BLAKE3}\Big(\texttt{"DSM/dev-empty\textbackslash 0"}\Big).$$

Per–Device SMT (sparse). For each device, a Per–Device Sparse Merkle Tree indexes *that device's* bilateral relationships; leaves store the *current chain tip digest* $h_{A\leftrightarrow B}$ per counterparty. Other devices do not mirror this SMT; storage nodes may keep concise aggregated mirrors. Receipts *always* carry inclusion proofs (not "membership proofs") against the relevant SMT roots.

16.4 Addressing and Transport (`b0x[...]`)

Purpose. `b0x` is a *fixed, literal* prefix that unambiguously marks an *online/unilateral* submission. Offline bilateral exchanges *never* use `b0x`; they require live proximity transport (BLE/NFC) and immediate co–signature.

Deterministic online address (rotates with the chain tip). For a unilateral submission from A to B, the delivery address is derived as

$$\text{addr}_{A\to B} := \texttt{b0x[}\,\text{BLAKE3}\Big(\texttt{"DSM/addr-G\textbackslash 0"} \parallel G \parallel \text{salt}_G\Big)_{0..31}\,;\ \text{BLAKE3}\Big(\texttt{"DSM/addr-D\textbackslash 0"} \parallel \text{DevID}_B \parallel \text{sal}$$

where h_n is the current bilateral chain tip digest $\text{tip}_{A\leftrightarrow B}$, and nonce is a sender-chosen value. Each component is exactly 64 hex characters (32 bytes). Per-user salts salt_G and salt_D are derived from the user's identity to prevent correlation attacks while maintaining deterministic addressing.

Rotation and privacy. Each stitched receipt that advances the bilateral chain changes $\text{tip}_{A\leftrightarrow B}$, and thus rotates the final component and the full `b0x[...]` address. Only the two counterparties know the live tip, so only they can derive the next valid address; storage nodes merely store and return bytes keyed by this address and cannot infer relationship metadata from the blinded components.

Retrieval and stitching. Upon sync, B derives the current address from $(G, \text{DevID}_B, \text{tip}_{A\leftrightarrow B})$, fetches any pending submissions under that `b0x[...]` key, verifies inclusion proofs and signatures, and then stitches the receipts to advance the tip. Submissions addressed to *older* tips are either (i) stitched in order if still admissible by the commitment predicate, or (ii) rejected as invalid against the current state.

$$\text{addr}_{A\to B} := \texttt{b0x[}\,\text{BLAKE3}\Big(\texttt{"DSM/addr-G\textbackslash 0"} \parallel G \parallel \text{salt}_G\Big)_{0..31}\,;\ \text{BLAKE3}\Big(\texttt{"DSM/addr-D\textbackslash 0"} \parallel \text{DevID}_B \parallel \text{sal}$$

Modal lock (relationship–local). If any pending online submission exists for (A, B) under the latest derived `b0x[...]` key, *offline* (A, B) transactions are rejected until B retrieves and stitches those submissions. Other relationships (A, C), (B, D), etc. remain unaffected.

No clocks, no heights. Address derivation and admissibility are purely hash–chain driven. There are no wall clocks, epochs, or heights in predicates or encodings for transport or ordering.

Implementation notes.

- **Canonical encoding:** Treat `b0x[...]` as a deterministic envelope key; serialize $(G, \text{DevID}_B, \text{tip}, \text{nonce})$ in fixed field order/lengths (e.g. canonical CBOR or a dedicated canonical byte layout) *before* hashing/signing payloads. Do not hash Protobuf or JSON.
- **Deduplication:** Include a per–submission unique ID inside the signed payload; storage may hold duplicates across nodes, the client dedupes by ID and signature.
- **Forward security:** The signed payload binds the predecessor and successor tips, so even if an old `b0x[...]` key leaks, the resulting receipt cannot be replayed against the current tip.

16.5 Canonical Encoding and Protobuf Pipeline

- **Transport:** Protobuf envelopes only. Schemas are generated via a Prost–based flow: Rust types (`#[derive(Message)]`) define the *transport* schemas used by Android/iOS/Web bindings.
- **Crypto commits:** We *do not hash* on–wire Protobuf bytes. Every signed/hashed object has a *canonical commit form* (fixed field order and length prefixes) emitted by Rust and verified by Rust. The Protobuf payload is a lossy view for transport only.
- **Inclusion proof term:** Use "inclusion proof" everywhere; never "membership proof".
- **No JSON/base64:** Protocol messages and acceptance predicates are defined exclusively over Protobuf and canonical commit bytes. JSON and base64 are forbidden for wire transport and verification logic.

16.6 Ordering and Concurrency (No Clocks, No Heights)

DSM uses the bilateral hash chain itself for strict ordering; no timestamps or heights appear in any predicate. Concurrency is resolved by stitched receipts and SMT root replacement:

1. Each proposed successor at tip h_n carries a pre–commit $C_{\text{pre}} = H(h_n \| \text{op} \| e)$ and an inclusion proof that h_n is the current Per–Device SMT leaf.
2. The successor $h_{n+1} = H(h_n \| \text{op} \| e \| \sigma)$ is accepted *iff* the stitched receipt validates and the Per–Device SMT *replace* $(h_n \to h_{n+1})$ recomputes the advertised new root r'_A with valid inclusion proofs (old and new).
3. Any concurrent attempt with the *same* h_n that is not bit–identical is rejected by the device–local SMT replace rule (Tripwire property).

Deterministic timing without clocks. DSM replaces wall time with a device–calibrated BLAKE3 iteration budget.

$$\kappa := \text{Calibrate once per device (BLAKE3 iterations for a fixed work unit)}, \quad (19)$$
$$\mathsf{Delay}(c) := \text{prove } c \text{ iterations of a fixed BLAKE3 loop}, \quad (20)$$

with c chosen deterministically from policy/HKDF. This yields a cryptographically verifiable, deterministic counter for rate–limits and vault delays without any reliance on wall clocks.

16.7 Key Management, DBRW Binding, and SPHINCS+

Per–step KDF and ephemeral signing keys. Per–step key derivation is fully specified in Sec. 11.1 and Sec. 12. In summary:

- A device–bound secret K_{DBRW} is derived from hardware and environment (DBRW).
- A master seed S_{master} is derived via HKDF–BLAKE3 from $(G, \text{DevID}, K_{\text{DBRW}}, s_0)$.
- For each parent h_n and precommit C_{pre}, a per–step seed E_{n+1} is derived via HKDF–BLAKE3 from $(h_n, C_{\text{pre}}, k_{\text{step}}, K_{\text{DBRW}})$, where k_{step} is derived from a Kyber secret (Sec. 11.1).
- An ephemeral SPHINCS+ keypair $(\mathsf{EK}^{\text{sk}}_{n+1}, \mathsf{EK}^{\text{pk}}_{n+1})$ is deterministically generated from E_{n+1} and certified by the previous key (AK or prior EK).

No long–term signing key is exposed at the protocol layer; all signatures are ephemeral and chained to the hash chain and DBRW binding.

Receipts (Per–Device SMT replace). For $(A \leftrightarrow B)$ at tip h_n, A computes h_{n+1} as above, updates its Per–Device SMT leaf to produce r'_A, and forms

$$\tau_{A\to B} = \text{enc}\Big(\text{“DSM–StitchedReceipt–v1”},\ G,\ \text{DevID}_A,\ \text{DevID}_B,\ h_n,\ h_{n+1},\ r_A,\ r'_A,\ \pi_{\text{rel}}(h_n \in r_A),\ \pi'_{\text{rel}}(h_{n+1} \in$$

Both parties sign the *canonical commit form* of τ with their per–step SPHINCS+ keys. Inclusion proofs must verify; the SMT replace must recompute r'_A; otherwise the receipt is rejected.

16.8 Offline vs. Online Flows

Offline (bilateral, co–sign live). Devices exchange C_{pre}, verify inclusion proofs locally, derive per–step keys, countersign the receipt, and each applies the Per–Device SMT replace. No storage node is required for finality.

Online (unilateral, `b0x[...]`). Sender posts to `b0x[`$G, \text{DevID}_B, \text{BLAKE3}(h_n \| \text{nonce})$`]`. The recipient syncs, verifies inclusion proofs and replace semantics, then applies the new tip. The **modal lock** prevents mixing pending online with new offline for the same pair.

16.9 Storage Nodes and Censorship Resistance

Storage nodes expose a REST API to store/fetch: (i) Device Tree nodes and root R_G, (ii) Per–Device SMT mirrors (aggregated), (iii) `b0x[...]` spools, (iv) recovery capsules, and (v) ByteCommit chains. They neither order nor validate protocol transitions. Censorship resistance follows from client–side verification plus replication: if any node refuses, the same, self–verifying envelope can be relayed to other nodes; acceptance requires only the counterparty's device in the bilateral context.

16.10 Recovery Capsule AEAD and DLV

After each accepted receipt, the device can write a capsule (Sec. 13).

AEAD choice (normative). Use AES-256-GCM with a 256-bit key derived via Argon2id from the mnemonic, and a 96-bit (12-byte) nonce.

Associated data (normative).

$$\text{AD} := \texttt{"DSM/capsule\textbackslash 0"} \parallel r_t \parallel c_{\text{le64}} \parallel \text{BLAKE3}\Big(\texttt{last\_TR\_or\_zero}\Big).$$

Nonce derivation (normative). Derive a fixed-length nonce by hashing the context and truncating:

$$\text{nonce} := \text{BLAKE3}\Big(\texttt{"DSM/capsule-nonce\textbackslash 0"} \parallel r_t \parallel c_{\text{le64}} \parallel \text{DevID}\Big)_{0..11}.$$

Nonce reuse is forbidden. Since c_{le64} is monotone per device, uniqueness holds per K_R.

Encryption (normative).

$$\text{Capt} := \text{AES-256-GCM.Encrypt}_{K_R}\Big(\text{nonce},\ \text{plaintext},\ \text{AD}\Big).$$

where r_t is the current Per–Device SMT root, c is the current counter, and last_TR_or_zero is the hash of the most recent Tombstone Receipt or all-zero bytes if none exists. The plaintext contains the recovery state: current tips, metadata, and receipt rollup. DLV semantics are as in Sec. 7.3.

16.11 Build, Tooling, and Generation Pipeline

- **Rust workspace:** `cargo build -locked -workspace -all-features`.
- **Android NDK:** `cargo-ndk -t armeabi-v7a -t arm64-v8a -t x86_64 -o ./android/app/src/main/jniLibs build -r`.
- **JNI:** Minimal surface in `DsmNativeWrapper.kt`; all parameter validation occurs in Rust.
- **Protobuf:** Prost derives from Rust types generate transport schemas for Kotlin/TS. Canonical commit bytes are emitted *only* by Rust.
- **Frontend:** `pnpm build` bundles React assets copied into Android assets for the WebView.
- **Testing:** Rust unit/integration tests for SMT replace and receipts; Android instrumentation for BLE/NFC shims; end–to–end offline/online parity tests.

16.12 Operational Parameters (Recommended)

- **Hash:** BLAKE3 (256–bit digests) for commits and iteration counters.
- **Signatures:** SPHINCS+ (per–step, deterministic derivation).
- **KEM:** Kyber for step secrets; keys never serialized directly.
- **SMT:** 256–bit key space; inclusion proofs logarithmic; device–local only.
- **Device Tree:** Regular Merkle, replicated on all user devices and storage nodes.
- **Entropy:** s_0 and s_{device} via CSPRNG; per–step seeds via HKDF–BLAKE3 over $(h_n, C_{\text{pre}}, k_{\text{step}}, K_{\text{DBRW}})$.
- **Timing:** No timestamps or heights; use calibrated BLAKE3 iteration budgets.
- **Modal rule:** Pending online for (A, B) blocks offline for (A, B) until sync; others unaffected.

Summary. DSM's implementation is mobile–first: Rust is compiled into an Android *native* library (NDK), invoked through a thin JNI layer, and surfaced to a React UI via a single bridge. Transport uses Protobuf; cryptographic commits are canonical, Rust–defined bytes; ordering is enforced by bilateral hash chains and Per–Device SMT *replace*, not by time or height. SPHINCS+ is per–step and derived deterministically with Kyber and DBRW binding, preserving strong authenticity without long–term signing keys at the protocol layer.

17 Conclusion

DSM is a *clockless* bilateral trust fabric with **two Merkle layers**: a replicated Device Tree that binds device IDs to a single genesis, and a Per–Device SMT that indexes relationships and their linear hash chains. Ordering is enforced solely by hash adjacency. Receipts carry inclusion proofs and post–quantum signatures; ephemeral SPHINCS+ keys are chained to the parent and bound to the device via DBRW. Online delivery is deterministic via `b0x[...]` with a blinded parent tip; offline is bilateral live–sign. The result is robust, scalable, and suitable for large–scale deployment.

terminology (source of truth)

BLAKE3 hash used for commitments and calibrated iteration budgets

Kyber post-quantum KEM used to derive per-step shared secrets

Argon2id memory-hard KDF used to derive the ring key from a mnemonic

genesis root commitment that binds all device identities of a user

DevID stable device identifier (domain-separated hash of a post-quantum attestation key and metadata); leaf in the device tree

device tree standard merkle tree whose leaves are device ids bound to the user's genesis; root R_G

per–device SMT device-local SMT that maps each bilateral relationship to its current chain tip; root r_A

chain tip latest digest h_n of a bilateral straight hash chain

hash adjacency ordering rule: the successor must embed the parent hash under canonical encoding (no clocks/heights)

inclusion proof merkle authentication path proving a key/value is committed in a given root

non–inclusion proof sparse proof that a key resolves to the zero leaf in an SMT

zero leaf canonical empty value for absent keys in an SMT

pre–commit (C_{pre}) deterministic hash at the parent that locks the candidate op and entropy

stitched receipt signed envelope binding $(h_n \to h_{n+1})$, $(r_A \to r'_A)$, and inclusion proofs

smt replace deterministic per-device SMT update $h_n \mapsto h_{n+1}$ recomputing r'_A byte-exactly

canonical commit form byte-exact serialization used for hashing/signing (separate from on-wire protobuf)

protobuf envelope on-wire transport encoding for requests/replies; never hashed for cryptographic commits

smart commitment deterministic, non–turing-complete transition predicate built from pre–commit (C_{pre}) and stitched receipts

pre–commit forking authoring mutually exclusive pre–commit (C_{pre}) candidates at the same parent; only one can be stitched

external commitment hash commitment to external data/state, referenced inside receipts without trusting an external executor

`b0x[...]` fixed prefix marking an online/unilateral delivery key: $(G, DevID, H(\text{tip}\|\text{nonce}))$

nonce single-use salt mixed with the live chain tip to blind the `b0x` address tag

online (unilateral) sender posts a signed candidate to the recipient's rotating `b0x[...]` address; recipient stitches upon sync

offline (bilateral) both devices co-sign the successor live over BLE/NFC; no `b0x`

modal lock if any pending online submission exists for (A, B), new offline (A, B) is rejected until sync

tripwire theorem fork-exclusion: with EUF–CMA signatures and collision-resistant hashing, two accepted successors for the same parent are negligible

causal consistency acceptance requires valid inclusion proofs along per-device/device-tree paths; no global order

recovery capsule encrypted NFC payload recording $(r_t, \text{Meta}, \{\text{PeerID}^{(8)}, h\}, \text{Roll}_t)$ for offline restore

ring key (K_R) key from mnemonic via Argon2id; used to AEAD-encrypt the recovery capsule

tombstone (TR) receipt that invalidates the previous device binding/root $r^\star$

succession (SR) receipt that binds a new device after tombstone (TR); valid only while TR is active

storage node http persistence that replicates device-tree snapshots, aggregated per-device SMT mirrors, `b0x` spools, and capsules; clients verify

subscription model gasless economics: users pay for storage/availability; operators paid for capacity, durability, bandwidth

iteration budget device-calibrated BLAKE3 work units for deterministic delays/rate-limits (no wall clocks)

webview bridge A unidirectional bridge between the native DSM client and a sandboxed WebView, transporting raw protobuf envelopes into the browser environment while keeping the trust boundary anchored in the native client.

Acronyms

SMT sparse merkle tree

KEM key encapsulation mechanism

AEAD authenticated encryption with associated data

HKDF HMAC-based key derivation function

RTT round-trip time

UI user interface

FFI foreign function interface

SPHINCS+ stateless hash-based signature scheme

DBRW dual–binding random walk; binds state to hardware entropy and environment fingerprint

DLV deterministic limbo vault; unlock key derives only upon stitched proof-of-completion

NDK android native toolchain

JNI java/kotlin native bridge

BLE bluetooth low energy

NFC near-field communication

18 Worked Examples (Alice, Bob, Carol)

This section gives concrete, implementation-ready traces that exercise DSM's core flows: (1) offline bilateral (co-sign live), (2) online unilateral via `b0x[...]`, (3) DLV + deterministic smart commitments with an external commitment, and (4) a three-party choreography (Alice–Bob–Carol) realized as composable bilateral updates. Transport is always Protobuf; *all* cryptographic commits use the canonical commit form (Sec. 4.2.1).

Normative Authority (core vs. SDK). The **Rust protocol core crate** `dsm_core` is the *sole* source of truth for: canonical commit bytes, acceptance predicates, Merkle proof verification, SMT replace semantics, and signature/KDF logic. Language SDKs/bindings are *non-authoritative shims* that MUST forward requests to `dsm_core` and MUST NOT re-encode canonical commits or re-implement validation logic.

We use the following fixed notations throughout:

$$G_A, G_B, G_C \in \{0,1\}^{256} \quad \text{(genesis digests)}$$
$$\text{DevID}_A, \text{DevID}_B, \text{DevID}_C \in \{0,1\}^{256}$$
$$H := \text{BLAKE3-256}, \quad \text{SPX} := \text{SPHINCS+ (BLAKE3, Cat-5, f)}$$
$$\mathsf{Key}(A,B) := H(\texttt{"DSM/smt-key\textbackslash 0"} \parallel \min(\text{DevID}_A, \text{DevID}_B) \parallel \max(\text{DevID}_A, \text{DevID}_B))$$
$$k_{A\leftrightarrow B} := \mathsf{Key}(A,B), \quad \texttt{ZERO\_LEAF} := \texttt{0x00}^{32}$$
$$\text{Per-Device SMT roots: } r_A, r_B, r_C$$
$$\text{Relationship tip digests: } h_n^{A\leftrightarrow B} \text{ for step } n$$

18.1 State Snapshot (before any example)

On device A (Alice):

$$\text{Leaf key } k_{A\leftrightarrow B} = \mathsf{Key}(A,B), \quad \text{Leaf value } v_{A\leftrightarrow B} = h_n^{A\leftrightarrow B}$$
$$\text{Inclusion proof } \pi_{\text{rel}}(h_n^{A\leftrightarrow B} \in r_A)$$
$$\pi_{\text{dev}}(\text{DevID}_A \in R_{G_A}) \quad \text{(Device Tree proof)}$$

Bob's device B holds the symmetric view for (A,B) with its own *Per–Device* SMT root r_B and parent $h_n^{A\leftrightarrow B}$.

For brevity in the examples below, we often write h_n when the relationship is clear from context; formally this is $h_n^{A\leftrightarrow B}$ for the (A,B) bilateral domain, and analogously for (A,C) or (B,C).

18.2 Example 1: Offline Bilateral Transfer (Bluetooth/NFC)

Goal: Alice transfers α tokens to Bob *offline*. Both are co-present.

Step 1: Pre-commit. Alice proposes operation op = `Transfer`(α) with entropy e,

$$C_{\text{pre}} = H\Big(h_n^{A\leftrightarrow B} \parallel \text{op} \parallel e\Big).$$

Step 2: Per-step keys (clockless). Each device derives per-step material exactly as in Sec. 16.7, using only *already-committed* inputs; no timestamps or heights. Concretely, both sides invoke the normative per-step KDF with parent tip $h_n^{A\leftrightarrow B}$ and pre-commit C_{pre}, obtaining the SPHINCS$^+$ ephemeral keypair

$$(\mathsf{EK}_{n+1}^{\text{sk}}, \mathsf{EK}_{n+1}^{\text{pk}}).$$

No long-term signing key is exposed at the protocol layer; the per-step key is deterministically bound to $(h_n^{A\leftrightarrow B}, C_{\text{pre}})$ and the device's DBRW binding.

Step 3: Successor state and balance update. Alice builds S_{n+1} embedding $h_n^{A\leftrightarrow B}$ and the balance delta $\Delta_A = -\alpha, \Delta_B = +\alpha$ (Sec. 8), then

$$h_{n+1}^{A\leftrightarrow B} = H(S_{n+1}).$$

Step 4: SMT replace (on both devices). Each device *locally* performs

$$r'_A = \mathsf{SMT\text{-}Replace}(r_A, k_{A\leftrightarrow B} : h_n \mapsto h_{n+1}), \quad r'_B = \mathsf{SMT\text{-}Replace}(r_B, k_{A\leftrightarrow B} : h_n \mapsto h_{n+1}).$$

Step 5: Stitched receipt (co-sign live). Form canonical commit bytes as in Sec. 4.2.1, then both sign:

$$\tau_{A\leftrightarrow B} = \text{enc}\Big(\dots\Big), \quad \sigma_A = \text{SPX.Sign}_{\mathsf{EK}_{n+1,A}^{\text{sk}}}(\mathsf{commit}), \quad \sigma_B = \text{SPX.Sign}_{\mathsf{EK}_{n+1,B}^{\text{sk}}}(\mathsf{commit}).$$

Both devices accept upon verifying signatures and proofs; the tip advances to h_{n+1}. No storage node or `b0x` is involved.

Acceptance (deterministic). Verify: (1) SPX sigs, (2) inclusion proofs old/new leaves, (3) Device Tree proof for DevID$_A$, (4) SMT replace recomputes r'_A/r'_B, (5) token invariant $B_{A,n+1} \geq 0$. If any fails $\Rightarrow$ reject.

18.3 Example 2: Online Unilateral Delivery via `b0x[...]`

Goal: Alice initiates a send while Bob is offline. Alice submits a candidate; Bob later stitches if adjacent.

Step 1: Address derivation (blinded, tip-rotating).

$\mathrm{addr}_{A\to B} = \texttt{b0x[}\, H(\texttt{"DSM/addr-G\textbackslash 0"} \parallel G_B \parallel \mathrm{salt}_G)_{0..31}\ ;\ H(\texttt{"DSM/addr-D\textbackslash 0"} \parallel \mathrm{DevID}_B \parallel \mathrm{salt}_D)_{0..31}\ ;\ H(\texttt{"}$

Only Alice and Bob know the live tip; storage nodes cannot correlate relationships.

Step 2: Submission. Alice posts a Protobuf envelope $\mathcal{E}$ keyed under $\mathrm{addr}_{A\to B}$ containing the candidate successor, proofs, and Alice's SPX signature over the canonical commit bytes. Storage nodes persist; they do *not* validate.

Step 3: Stitching on sync (Bob). When Bob comes online, he derives $\mathrm{addr}_{A\to B}$ from his G_B, DevID_B, and current $h_n^{A\leftrightarrow B}$; fetches pending $\mathcal{E}$; verifies proofs and signatures; recomputes $r'_B = \mathsf{SMT\text{-}Replace}(\dots)$. If valid, Bob countersigns to produce the stitched receipt and advances to h_{n+1}.

Modal lock (relationship-local). If any pending online submission exists for (A, B), a new *offline* transaction for (A, B) is invalid until stitched; other pairs are unaffected.

18.4 Example 3: DLV + Smart Commitments + External Commitment

Goal: Alice escrows tokens in a Deterministic Limbo Vault (DLV) to Bob, to be released only if an external condition X is met (e.g., hash of a delivery attestation). No Turing-complete VM; purely deterministic commitments.

DLV configuration (commit-only).

$V = (L, C, H), \quad C = \{\texttt{require\_X},\ \texttt{deadline\_proof}\}, \quad X = H(\texttt{"DSM/ext\textbackslash 0"} \parallel \mathsf{attestation\text{-}bytes}).$

Alice publishes a receipt that commits to V and moves the escrowed amount from her free balance to the vault balance. This is a normal stitched receipt with a *smart commitment* clause referencing X (no oracle execution).

Satisfying C. Bob produces stitched receipts carrying *inclusion* of X (as a pure hash value—the external data itself is not trusted) and the required co-signature from Alice acknowledging satisfaction. The *proof-of-completion* σ is the minimal stitched evidence set that references X and the DLV commit.

Unlock key derivation (deterministic).

$$\mathrm{sk}_V = H(L \parallel C \parallel \sigma).$$

No clocks. If C is never satisfied, the DLV remains locked; recovery/refund can be expressed as a mutually exclusive pre-commit branch (below).

Pre-commit forking (mutually exclusive outcomes). Alice prepares two branches at the same parent:

$$C_{\text{pre}}^{\texttt{release}} = H(h_n \parallel \texttt{release} \parallel e_1), \quad C_{\text{pre}}^{\texttt{refund}} = H(h_n \parallel \texttt{refund} \parallel e_2).$$

Tripwire guarantees only one successor can be accepted. If X is presented and co-signed, `release` stitches; otherwise the mutually exclusive `refund` branch may stitch after a *deterministic* iteration budget window (Sec. 16.6)—still clockless.

18.5 Example 4: Three-Party Choreography (Alice, Bob, Carol)

DSM remains bilateral at the protocol layer; multi-party logic is composed via *coordinated* bilateral receipts and shared external commitments.

Scenario. Carol is the beneficiary if both Alice and Bob attest to condition Y (e.g., Carol delivered service to each). Each pair runs its own bilateral chain: $(A \leftrightarrow C)$ and $(B \leftrightarrow C)$, with a shared external commitment

$$Y = H(\texttt{"DSM/ext\textbackslash 0"} \parallel \textsf{delivery-proof}).$$

Phase 1: Lock by Alice and Bob. Independently, Alice and Bob each create a DLV transfer to Carol with conditions that reference the *same* Y:

$$V_A : C_A = \{\texttt{require\_Y},\ \texttt{A-ack}\}, \quad V_B : C_B = \{\texttt{require\_Y},\ \texttt{B-ack}\}.$$

These are stitched on $(A \leftrightarrow C)$ and $(B \leftrightarrow C)$, respectively. No 3-party signature is required; only bilateral receipts exist.

Phase 2: Satisfaction and release. Carol (or Alice/Bob) presents Y in each bilateral relationship. If Carol correctly performed, Alice co-signs `A-ack` with Y and Bob co-signs `B-ack` with Y. Each bilateral domain produces its σ, so the two DLVs independently derive their unlock keys:

$$\text{sk}_{V_A} = H(L_A \parallel C_A \parallel \sigma_A), \quad \text{sk}_{V_B} = H(L_B \parallel C_B \parallel \sigma_B).$$

Funds release to Carol occur in *two* stitched receipts, one per pair; there is no global consensus or 3-party ledger. If one side fails (e.g., Bob disagrees), Alice's path can still independently release or refund under her own mutually exclusive pre-commit branches.

Consistency and safety. At no point can conflicting successors consume the same parent in any bilateral domain (Tripwire). No clocks are used. External data never executes; only commitments to its digest are referenced.

18.6 Canonical Protobuf Transport (Illustrative Snippet)

Transport messages are Protobuf; cryptographic commits are canonical commit bytes emitted by the **Rust core crate** `dsm_core` (e.g., a fixed-length, length-prefixed CBOR layout).

Listing 2: Protobuf envelope (illustrative transport fields only; commits are canonical bytes from the Rust core)

```
message StitchedReceipt {
  bytes genesis = 1; // 32B
  bytes dev_id_a = 2; // 32B
  bytes dev_id_b = 3; // 32B
  bytes parent_tip = 4; // 32B
  bytes child_tip = 5; // 32B
  bytes parent_root = 6; // 32B
  bytes child_root = 7; // 32B
  bytes rel_proof_prev = 8; // opaque proof bytes (canonical encoding inside)
  bytes rel_proof_next = 9; // opaque proof bytes (canonical encoding inside)
  bytes dev_proof = 10; // opaque proof bytes (canonical encoding inside)
  bytes sig_a = 11; // SPHINCS+ signature (over canonical commit bytes)
  bytes sig_b = 12; // SPHINCS+ signature (over canonical commit bytes)
}
```

Normative reminder. *Never* hash/sign the Protobuf bytes. Always hash/sign the canonical commit bytes (Sec. 4.2.1) produced by `dsm_core`; Protobuf is *transport only*. SDKs/bindings MUST treat `dsm_core` as the authority and MUST NOT alter commit encodings or predicates. JSON and base64 are forbidden on the protocol path.

18.7 What Acceptors Must Check (All Examples)

Given any claimed successor, an acceptor MUST:

1. Verify SPX signatures against the canonical commit bytes.
2. Verify inclusion proofs for $(h_n \in r)$ and $(h_{n+1} \in r')$.
3. Verify DevID inclusion in the Device Tree (R_G).
4. Recompute Per–Device SMT replace and match r' byte-exactly.
5. Enforce token invariants $(B_{n+1} \geq 0$; supply conservation).
6. Enforce modal lock for (A, B) if pending online exists.
7. Enforce receipt size $\leq 128\,\text{KiB}$ (signatures + proofs).
8. Reject any indefinite-length or non-canonical encoding for proofs; determinism is required.

Result. These examples cover: offline bilateral, online unilateral, DLV lifecycle with mutually exclusive pre-commit branches, an external commitment, and a composed three-party outcome—*all* realized by stitched, clockless, bilateral receipts with Merkle inclusion proofs and SPX signatures, with `dsm_core` as the single execution authority.