# Deterministic State Machine: A Concise, Post–Quantum Specification

Brandon "Cryptskii" Ramsay

December 15, 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 Deterministic 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 Deterministic Join–Triggered Emissions (DJTE): a capped, halving–scheduled distribution mechanism sourced from a CPTA–bound Deterministic Limbo Vault (DLV) and activated only by a spend–gate unlock. DJTE is proof–carrying and offline–verifiable: eligibility is committed via shard activation accumulators, global population counts are committed via a sparse Merkle root, winner selection is exact–uniform over the activated set using deterministic index descent, and double consumption is structurally prevented by a spent–proof SMT that transitions a Join Activation Proof (JAP) from *unspent* to *spent* under strict–fail validation. 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.

**Deterministic emissions without consensus or time.** DSM treats emissions as a deterministic state transition, not a "network event." Under DJTE, new distribution is triggered only when a device unlocks its spend–gate and produces a Join Activation Proof (JAP). The activated population is committed in shard–local append–only accumulators, while a global sparse Merkle commitment binds the shard counts into a single root that defines the exact eligible population size $N$. Given a public seed derived from hash–adjacent DSM state (not time), any verifier can compute an exact–uniform winner index $k \in [0, N)$, deterministically map $k$ into a shard and local position, and verify the winner by inclusion proof. Supply is enforced by debiting a CPTA–bound source DLV with an explicit remaining balance and a deterministic halving schedule; emissions never exceed the cap because over–issuance fails verification.

**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

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

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.

**Hash function (normative).** Unless explicitly stated otherwise, $H(\cdot)$ denotes

$$H(X) := \mathrm{BLAKE3\text{-}256}\Big(\texttt{"DSM/hash\0"} \parallel X\Big),$$

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

**Canonical encoding (normative).** Whenever the specification requires hashing or signing a structured object (state, receipt commit, proofs), the object is first serialized using DSM Envelope wire v3 deterministic Protobuf rules (Sec. 4.2.1). No JSON, CBOR, base64, hex text encodings, or non-deterministic serializers are permitted in acceptance predicates.

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.

**Node hashing (normative).** For any binary Merkle node with left child digest $L$ and right child digest $R$,

$$\mathsf{Node}(L, R) := \text{BLAKE3-256}\Big(\texttt{"DSM/merkle-node\0"} \parallel L \parallel R\Big),$$

and leaves are hashed with an explicit leaf domain:

$$\mathsf{Leaf}(X) := \text{BLAKE3-256}\Big(\texttt{"DSM/merkle-leaf\0"} \parallel X\Big).$$

**SMT Parameters:** The zero leaf value is `ZERO_LEAF` (exactly 32 zero bytes). Leaf keys for relationship indexing are derived as

$$k_{A\leftrightarrow B} := \text{BLAKE3-256}\Big(\texttt{"DSM/smt-key\0"} \parallel \min(\text{DevID}_A, \text{DevID}_B) \parallel \max(\text{DevID}_A, \text{DevID}_B)\Big),$$

using lexicographic ordering on the 32-byte identifiers to ensure canonicality.

**Zero leaf.**

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

**Default nodes.**

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

with

$$\mathsf{SMTNode}(L, R) := \text{BLAKE3-256}\Big(\texttt{"DSM/smt-node\0"} \parallel L \parallel R\Big).$$

**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 $\left(k,\ v_{\text{path}},\ \texttt{siblings[]}\right)$ where:

- $k$ is the 32-byte key queried.
- $v_{\text{path}}$ is either
  1. the existing leaf $\left(k', v'\right)$ 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 hash upward to a candidate root.
2. Case (a): if $v_{\text{path}}$ is a concrete leaf $(k', v')$, require $k' \neq k$ and that $(k', v')$ is consistent with the proof's divergence position; the reconstructed root must equal *root*.
3. Case (b): if $v_{\text{path}}$ is `absent`, require that the leaf position for $k$ is `ZERO_LEAF` under the provided siblings and defaults; the reconstructed root must equal *root*.

**Protobuf encoding (normative).** All SMT proofs are serialized as DSM Envelope wire v3 Protobuf messages with deterministic serialization (Sec. 4.2.1). For non-inclusion proofs, the message MUST contain:

- `bytes key` (size 32) for $k$
- `oneof path_value`:
  1. `message ExistingLeaf { bytes key_prime (size 32); bytes value_prime (size 32); }`
  2. `bool absent = true`
- `repeated bytes siblings` where each element is size 32, ordered from root-to-leaf.

No CBOR, JSON, base64, or hex-text encodings are permitted for proofs in acceptance predicates.

### 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 $\pi_{\text{rel}}$ and $\pi_{\text{dev}}$ 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 $G \in \{0,1\}^{256}$. Each device holds a long–term post–quantum attestation keypair $(\text{sk}_A, \text{pk}_A)$ (SPHINCS+), a stable device identifier $\text{DevID}_A$ (a domain–separated digest bound to $\text{pk}_A$ and device attestation), and a Per–Device SMT over relationships. All hashes and signatures are domain–separated (e.g. labels like `"DSM/receipt\0"`). Domain tags are part of the commitment and are not optional.

**Device identifier (normative).** Let $\text{Att}_A$ be the stable device attestation digest (platform-specific, but deterministic). Then

$$\text{DevID}_A := \text{BLAKE3-256}\Big(\texttt{"DSM/devid\0"} \parallel \text{pk}_A \parallel \text{Att}_A\Big).$$

### 2.5 Genesis State Creation

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

$$G = \text{BLAKE3-256}\Big(\texttt{"DSM/genesis\0"} \parallel b_1 \| \cdots \| b_t \| A\Big). \tag{3}$$

## 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; storage nodes are dumb by design and do not validate transitions.

# 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 $e$:

$$C_{\text{pre}} = \text{BLAKE3-256}\Big(\texttt{"DSM/precommit\0"} \parallel h_n \| \text{payload} \| e\Big). \tag{4}$$

Counterparty verifies and co–signs $C_{\text{pre}}$. 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

$$\begin{aligned}\tau_{A\to B} = \ \text{enc}\Big(&\texttt{"DSM/stitched-receipt/v3\0"},\ G,\ \text{DevID}_A,\ \text{DevID}_B,\\ &h_n,\ h_{n+1},\ r_A,\ r'_A,\\ &\pi_{\text{rel}}\Big(h_n \in r_A\Big),\ \pi'_{\text{rel}}\Big(h_{n+1} \in r'_A\Big),\ \pi_{\text{dev}}\Big(\text{DevID}_A \in R_G\Big)\Big),\end{aligned}$$

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.

**Protobuf-Deterministic Definition (normative)** The canonical commit form is the deterministic Protobuf serialization (DSM Envelope wire v3) of the *ReceiptCommit* message *only*. Transport envelopes MAY wrap this message for routing, fragmentation, or b0x delivery, but envelopes are excluded from all cryptographic commitments.

**ReceiptCommit fields (normative)** *receipt-commit* serializes the following fields in fixed semantic order (the Protobuf tags are defined in the DSM schema v2.4.0 / Envelope wire v3):

```
ReceiptCommit {
  genesis: bytes (size 32)          ; G
  devid_a: bytes (size 32)          ; DevID_A
  devid_b: bytes (size 32)          ; DevID_B
  parent_tip: bytes (size 32)       ; h_n
  child_tip: bytes (size 32)        ; h_{n+1}
  parent_root: bytes (size 32)      ; r_A
  child_root: bytes (size 32)       ; r_A'
  rel_proof_parent: bytes           ; pi_rel(h_n in r_A)
  rel_proof_child: bytes            ; pi'_rel(h_{n+1} in r_A')
  dev_proof: bytes                  ; pi_dev(DevID_A in R_G)
}
```

**Deterministic serialization rules (normative)** All implementations MUST produce identical bytes for the same logical ReceiptCommit:

1. Protobuf encoding is DSM Envelope wire v3, *deterministic*: fields are serialized in strictly increasing tag order; unknown fields are forbidden.
2. All `bytes` fields use definite length (length-delimited) encoding.
3. The first seven `bytes` fields are exactly 32 bytes; any other length is invalid.
4. Proof fields are raw proof bytes of their respective proof messages (also Protobuf-deterministic). Nested proof messages MUST themselves be deterministically serialized with no unknown fields.
5. `map` fields are forbidden in canonical commit forms. `repeated` fields (if any appear inside proof messages) are order-significant and MUST be serialized in the given order.
6. No optional fields, extensions, or forward-compatible padding are permitted in canonical commit forms. Versioning occurs at the envelope/message-type level, not via extra fields.

**Hashing (normative)**

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

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

**Test vectors (normative)** Canonical test vectors are distributed as Protobuf fixtures (byte-exact files) alongside expected BLAKE3-256 digests. Implementations MUST reproduce the digests exactly for conformance. Test vectors are intentionally not embedded as hex strings to avoid non-Protobuf representations in normative acceptance paths.

### 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. $\pi_{\text{rel}}$ proves $h_n \in r_A$ and $\pi'_{\text{rel}}$ proves $h_{n+1} \in r'_A$ (Per–Device SMT inclusion).
3. $\pi_{\text{dev}}$ proves $\text{DevID}_A$ 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:

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

### 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. The address is an *opaque deterministic routing token* derived from genesis, recipient device, and the sender's current parent tip for the relationship:

$$\texttt{b0x[}\,\text{BLAKE3-256}(\texttt{"DSM/addr-G\0"} \,\|\, G \,\|\, \text{salt}_G)\ \text{BLAKE3-256}(\texttt{"DSM/addr-D\0"} \,\|\, \text{DevID}_B \,\|\, \text{salt}_D)\ \text{BL} \tag{9}$$

where $G$ is the recipient's genesis, $\text{DevID}_B$ the recipient device, $h_n$ the sender's current parent tip for $(A \leftrightarrow B)$, nonce an ephemeral per-send value, and $\text{salt}_G$, $\text{salt}_D$ are per-user blinding salts. All three components are blinded to prevent correlation attacks while maintaining deterministic addressing. The address is treated as raw bytes in DSM Envelope wire v3 (no hex/base64/JSON encodings in acceptance predicates). 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 Online "b0x check" semantics (no global sync)

DSM does not "sync" global state. When online, devices *check their b0x* for waiting items. Acceptance remains strictly local: a received item is either (i) accepted because it is adjacent and proofs verify, (ii) queued because it is not yet adjacent, or (iii) rejected as invalid. Storage nodes are dumb by design and may deliver arbitrary bytes; devices are the sole validators.

## 5.3 Offline Bilateral

Offline requires both parties live (e.g. Bluetooth/NFC). The parties exchange precommitments, finalize, and countersign the receipt locally (no `b0x`). Offline finality does not require going online; the resulting receipt may later be placed into the recipient's `b0x` (or exchanged out-of-band) as a normal deliverable object.

## 5.4 Modal Synchronization Lock

Let $\mathsf{Pending}_{A\leftrightarrow B}$ hold if an accepted but not-yet-adjacent 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 the pending items are resolved (accepted or rejected). 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 $\text{DevID}_A$ 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 policy anchors (CPTA) and minted-asset creation incur a one-time fee that covers indexing, replication commitments, and archival integrity.
- **Operator sustainability.** Storage nodes are paid for capacity, durability, and retrieval bandwidth rather than transient compute. This removes incentives to throttle usage via gas and aligns incentives with availability.

*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 $\sigma$:

$$\mathrm{sk}_V = H\Big(\texttt{"DSM/dlv-unlock\0"} \parallel L \| C \| \sigma\Big).$$

   Prior to $\sigma$, $\mathrm{sk}_V$ 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 $\alpha$, sender and recipient use $\Delta_{\text{sender}} = -\alpha$ and $\Delta_{\text{recipient}} = +\alpha$. Summing $\Delta$ 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.

**Atomicity (normative).** Any token-affecting operation MUST be represented as a DSM state transition and MUST be coupled to the same adjacency and receipt predicates as any other state update. Token deltas cannot be applied "out of band"; a balance change without a valid adjacent receipt is invalid.

**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 $\Delta$ 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 $\alpha$. 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 $\Delta$ 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. □

**Capped emissions (normative).** Global conservation applies to transfers. Emissions are modeled as deterministic reveals from a CPTA-bound source DLV (Sec. 9) and are therefore conservation-preserving with respect to the fixed total supply: an emission allocates $+\alpha$ to the recipient while debiting $-\alpha$ from the source DLV balance in the same adjacent transition. Any transition that would cause the source DLV to underflow is invalid.

### 8.1 Deterministic Limbo Vault Invariants

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

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

Without $\sigma$, recovering $\text{sk}_V$ is negligible in $\lambda$.

**DLV domain separation (normative).**

$$\text{sk}_V = \text{BLAKE3-256}\Big(\texttt{"DSM/dlv-unlock\0"} \parallel L \| C \| \sigma\Big).$$

This construction is deterministic, clockless, and purely receipt-derived.

## 9 Context Policy & Token Anchors (CPTA)

This section specifies a *deterministic, immutable* policy object used to define token behavior and to constrain subsequent token operations under DSM. A CPTA is a single canonical Protobuf object 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, caps, transferability constraints, emission source binding). (2) An *expressive* hook for external commitments (eligibility sets, deposit ledgers, registries), referenced by hash and proven via receipts, without importing any 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.

$$G_T \;=\; \text{BLAKE3-256}\Big(\texttt{"DSM/token-genesis\0"} \,\|\, G_{\text{issuer}} \,\|\, s_T\Big),$$

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

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

$$\text{policy\_commit} := \text{BLAKE3-256}\Big(\texttt{"DSM/cpta\0"} \,\|\, \texttt{canonical\_cpta\_bytes}\Big).$$

*CPTAs are immutable.* Any change yields a new policy_commit and a new $G_T$ (new token). UI may display the first 8 or 16 bytes of policy_commit for legibility, but protocol logic always uses the full 32-byte digest.

**Anchoring and caching.** Receipts that create or reference a token MUST include policy_commit and MAY include an opaque *policy anchor pointer* (e.g., content-addressed hash or retrieval hint). Devices 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/SBTs).
- **kind:** `FUNGIBLE` | `NFT` | `SBT` (non-transferable).
- **supply:** `cap` (u128, fixed); `initial_alloc` (u128, optional, debited from source DLV if nonzero).
- **emission:** `DJTE` parameters, binding deterministic join-triggered reveals to this token:

  - `source_dlv`: 32B vault identifier (CPTA-bound DLV holding the pre-existing supply).
  - `halving_interval`: u64 (number of emissions per halving epoch).
  - `base_amount`: u128 (epoch-0 emission amount before halving).
  - `recipient_mode`: `WINNER_UNIFORM` (deterministic exact-uniform selection over activated set; Sec. **??**).

- **authority (discretionary operations):** threshold $t$-of-$N$ over a sorted list of issuer genesis IDs. Discretionary mint/burn is OPTIONAL and, if enabled, MUST still respect `cap`. DJTE emissions never require human 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 (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)

**Protobuf (deterministic, normative).** The canonical CPTA bytes are the deterministic Protobuf serialization (DSM Envelope wire v3) of the *CptaPolicy* message as defined in the DSM schema. No CBOR, JSON, base64, or hex-text encodings are permitted in normative commit paths. The CPTA commitment is the BLAKE3-256 of the domain-separated prefix plus the encoded bytes:

$$\text{policy\_commit} := \text{BLAKE3-256}\big(\texttt{"DSM/cpta\0"} \,\|\, \texttt{canonical\_cpta\_bytes}\big).$$

Deterministic Protobuf rules match Sec. 4.2.1: increasing tag order, no unknown fields, no maps in canonical objects, and order-significant repeated fields.

### 9.4 dsm_app.proto Additions (Transport Only)

Listing 1: Transport messages for CPTA and token ops; 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 Supply {
    bytes cap_u128 = 1; // u128 as 16-byte big-endian
    bytes initial_alloc_u128 = 2; // optional; debited from source DLV if nonzero
  }
  Supply supply = 7;

  message Djte {
    bytes source_dlv = 1; // 32B vault id / anchor
    uint64 halving_interval = 2; // emissions per epoch
    bytes base_amount_u128 = 3; // u128 as 16-byte big-endian
    enum RecipientMode { WINNER_UNIFORM = 0; }
    RecipientMode recipient_mode = 4;
  }
  Djte djte = 8;

  message Authority {
    uint32 threshold = 1; // t
    repeated bytes genesis_signers = 2; // 32B each, sorted
    bool enable_discretionary_mint_burn = 3; // if false: mint/burn ops invalid
  }
  Authority authority = 9;

  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 = 10;

  repeated bytes eligibility_anchors = 11; // 32B digests
  repeated bytes metadata_anchors = 12; // 32B digests

  bytes policy_commit = 13; // 32B; MUST equal canonical commit of this message
```

```
  string policy_pointer = 14; // 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;
    DjteEmission djte_emission = 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
  bytes amount_u128 = 3; // u128 as 16-byte big-endian
}

message Mint {
  bytes amount_u128 = 1;
  repeated bytes signer_genesis = 2; // t-of-N cosigners per CPTA.authority
}

message Burn {
  bytes amount_u128 = 1;
  repeated bytes signer_genesis = 2; // t-of-N cosigners per CPTA.authority
}

message DjteEmission {
  bytes jap_hash = 1; // 32B: Join Activation Proof digest
  uint64 emission_index = 2; // deterministic counter (see Sec. DJTE), not time
  bytes selection_proof = 3; // Protobuf proof bundle (ShardCountSMT + SAA)
  bytes amount_u128 = 4; // emitted amount (must match halving schedule)
  bytes recipient_genesis = 5; // 32B winner genesis
}

message NftClaim {
  bytes claimant_genesis = 1; // 32B
  bytes allowlist_merkle_proof = 2; // if allowlist.kind == MERKLE_ROOT
```

```
  bytes eligibility_evidence = 3; // Protobuf 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 Protobuf bytes;
2. `policy.token_genesis` $= G_T$ and is unique under issuer-defined namespace;
3. `ticker` and `alias` satisfy format constraints; `decimals` matches `kind`;
4. the `cap` is finite and nonzero; the `source_dlv` is present for DJTE-enabled tokens;
5. any `initial_alloc` is applied atomically under the same receipt and (if $> 0$) is debited from `source_dlv`.

**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 (optional).** If `enable_discretionary_mint_burn=false`, any `Mint/Burn` operation is invalid. Otherwise, require `authority.threshold` distinct cosignatures from `authority.signers`. Enforce `cap` and balance non-negativity. Discretionary mint/burn MUST NOT be used to bypass DJTE or exceed the fixed cap.

**Emissions (DJTE).** DJTE emissions are deterministic reveals from the CPTA-bound `source_dlv`, triggered only by spend-gate unlock events (JAPs). A `DjteEmission` is valid iff:

1. `jap_hash` is valid and unspent under the global SpentProofSMT transition for this emission index (Sec. **??**);
2. `selection_proof` proves the exact eligible population size $N$ (ShardCountSMT root) and inclusion of the selected winner in the shard activation accumulator (SAA);
3. the winner derivation is reproduced deterministically from the public seed and maps to `recipient_genesis` by exact-uniform index descent;
4. `amount_u128` equals the policy's halving schedule at `emission_index` and does not exceed remaining balance in the `source_dlv` (no underflow);

5. the state transition debits `source_dlv` by $-\alpha$ and credits the recipient by $+\alpha$ atomically in the same adjacent receipt.

No human signatures are required for DJTE emission validity.

**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 that reduces to an equality/containment proof against at least one anchored digest (e.g., a Merkle proof against a deposit-ledger root). Evidence is a Protobuf bundle; DSM does not execute external code.

**Binding to policy.** All `TokenOp`s 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, `cap`=$2^{128}-1$ (effectively unbounded within u128), `DJTE` disabled, and discretionary mint/burn enabled with `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* commitment.

**Issuance.** Each student submits `NftClaim` with (i) a Merkle proof against the cohort root, and optionally (ii) eligibility evidence proving inclusion in the award-roster anchor. The receipt is stitched bilaterally (student↔university device), and the SBT is created via a deterministic branch that requires the authority threshold.

**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, DJTE Emission)

**Intent.** Local currency whose distribution is deterministic, capped, and triggered by new spend-gated genesis activations.

#### CPTA (core).

- `kind`=`FUNGIBLE`, `decimals`=2, `cap`=$10^9$ units, `initial_alloc`=0.
- `DJTE` enabled with `source_dlv` bound to the token's pre-existing supply, `base_amount` and `halving_interval` defining a Bitcoin-style halving schedule.
- `authority.threshold`=0 and discretionary mint/burn disabled.
- no allowlist (any genesis can hold/transfer).

**Emission.** Each spend-gate unlock produces a JAP, which deterministically triggers a DJTE emission. The emission selects a recipient by exact-uniform winner selection over the activated population (ShardCountSMT + SAA proofs), debits the source DLV, and credits the winner in an adjacent receipt. No signatures are required for the emission itself; only the usual receipt signatures apply to the transition.

### 9.7 Interplay with External Commitments (Illustrative)

**Deposit-gated NFT allowlist.** If a project requires "pre-deposit → 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 a Protobuf evidence bundle. 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 a short policy number (first 8–16 bytes of policy_commit) 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:** operations are adjacency-verified; DJTE emissions are driven by spend-gate unlock events and deterministic emission indices.
- **Least authority:** routine transfers need no signers; only optional discretionary mint/burn require threshold signatures; DJTE emissions require none.
- **Local enforceability:** all predicates (allowlists, caps, authority thresholds, DJTE proofs) 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) serve Merkle/SMT proof material derived from stored byte-indexes 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. Nodes are dumb by design and signature-free; all enforcement is derived from mirrored Protobuf bytes, deterministic hashes, and device-signed stitched receipts.

### 10.1 Hardware-Bound Cryptographic Identity

Each node derives a non-forgeable identity from a network genesis anchor and DBRW binding:

$$\text{nodeID} \;=\; \text{BLAKE3-256}\Big(\texttt{"DSM/node-id\0"} \,\|\, G_{\text{net}} \,\|\, K_{\text{DBRW}}^{(\text{node})}\Big),$$

where $G_{\text{net}}$ 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. *Privacy:* $K_{\text{DBRW}}^{(\text{node})}$ MUST NEVER be serialized, logged, or included in any commitment or envelope; it is used only as internal key material.

### 10.2 Admission, Staking, and Service Commitments

To participate, a node presents:

1. an inclusion proof in the *Node Registry SMT* root $R_{\text{nodes}}$,

2. a *non-inclusion* (zero-leaf) proof in the *Node Denylist SMT* root $R_{\text{deny}}$, and

3. a stake commitment $\mathcal{S}$ in the native DSM token, bound to nodeID (implemented as a stake DLV reference plus its stitched proof-of-funding).

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 message blobs and make them retrievable until consumed.
- **Auditability:** prove storage of randomly sampled items via inclusion proofs and cycle-index continuity 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 $\mathsf{addr}$, a deterministic placement function maps

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

using a Fisher–Yates permutation seeded from

$$\text{BLAKE3-256}\big(\texttt{"DSM/place\0"} \parallel \mathsf{addr}\big)$$

and the current Node Registry vector. Reads are subject to a PaidK gate: a client *accepts* the content at $\mathsf{addr}$ only if at least $K$ distinct nodes in Replicas($\mathsf{addr}$) return identical bytes that (i) re-derive $\mathsf{addr}$ under the object-domain hash and (ii) satisfy any associated proof predicates (SMT/Merkle) for the requested key. Any disagreement between replicas for the same $\mathsf{addr}$ is objective evidence of misbehavior.

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 index $t$, 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"`. Define

$$h_t := \text{BLAKE3-256}\Big(\texttt{"DSM/bytecommit\0"} \,\|\, \mathsf{ProtoDet}(B_t)\Big),$$

and address the mirrored bytes as

$$\mathsf{addr}_B(t) = \text{BLAKE3-256}\Big(\texttt{"DSM/obj-bytecommit\0"} \,\|\, \mathsf{node\_id} \,\|\, \mathsf{u64le}(t) \,\|\, h_t\Big).$$

A verifier accepts $B_t$ if and only if:

- $\mathsf{ProtoDet}(B_t)$ is deterministic and domain-separated;
- the digest $h_t$ matches the mirrored addressing rule above;
- the parent link is valid ($h_{t-1}$ for $t > 0$ or $0^{32}$ for $t = 0$);
- $R_t^{\text{node}}$ 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 digest. 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 DSM rules (placement, PaidK, addressing, and SMT structure).

   The record is summarized by a domain-separated digest

$$h_E = \text{BLAKE3-256}\Big(\texttt{"DSM/evidence\0"} \,\|\, \mathsf{ProtoDet}(E)\Big),$$

   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 cycle indices. This sequence of $(B_t, R_t^{\text{node}})$ 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 index,
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 a Kyber KEM to derive per–transition step material and SPHINCS+ to authenticate receipts. All derivations are clockless and deterministically bound to adjacency inputs.

**Deterministic Kyber encapsulation (normative).** Let coins be deterministically derived from public and local secret inputs:

$$\mathsf{coins} := \text{BLAKE3-256}\Big(\texttt{"DSM/kyber-coins\0"} \parallel h_n \parallel C_{\text{pre}} \parallel \text{DevID}_{\text{sender}} \parallel K_{\text{DBRW}}\Big).$$

Encapsulation uses a deterministic coins interface (equivalently, a seeded KEM):

$$(\mathsf{ct}, \mathsf{ss}) = \text{KyberEncDet}(\text{pk}_{\text{recipient}}, \mathsf{coins}), \quad k_{\text{step}} = \text{BLAKE3-256}\Big(\texttt{"DSM/kyber-ss\0"} \parallel \mathsf{ss}\Big), \tag{12}$$

where all hashing uses 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 $\leq 128$ KiB (including two signatures and included proof material). Submissions exceeding the cap are invalid and *MUST* be rejected prior to proof verification.

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

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

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 $C_{\text{pre}}$, derive the per–step seed

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

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

$$H_{\text{ek-cert}}(X) := \text{BLAKE3-256}\Big(\texttt{"DSM/ek-cert\0"} \parallel X\Big).$$

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

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

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

**Verification (normative).** Verification replays the chain of certificates back to $\mathsf{AK}_{\text{pk}}$ and checks inclusion proofs (Sec. 4.2).

### 11.2 Identity Pre–commitment

Let $P_0$ be a provisioning seed; define $P_i = \text{BLAKE3-256}(\texttt{"DSM/provision\0"} \parallel P_{i-1})$. 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 $\text{BLAKE3-256}(\texttt{"DSM/commit\0"} \parallel S_n \parallel P) = \mathsf{expected}$.

## 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)**.**

$$K_{\text{DBRW}} = \text{BLAKE3-256}\Big(\texttt{"DSM/dbrw-bind\0"} \parallel H(d) \parallel E(e) \parallel s_{\text{device}}\Big),$$

where $s_{\text{device}}$ is a per-device salt ensuring uniqueness even for similar hardware or environments.

**Theorem 5** (Binding Inseparability)**.** *Given $K_{\text{DBRW}}$ and collision resistance of BLAKE3–256 under domain separation, it is infeasible to find $(h', e', s') \neq (h, e, s)$ such that* $\text{BLAKE3-256}(\texttt{"DSM/dbrw-bind}$ $h' \parallel e' \parallel s') = \text{BLAKE3-256}(\texttt{"DSM/dbrw-bind\0"} \parallel h \parallel e \parallel s)$*. The per-device salt $s_{\text{device}}$ prevents correlation attacks by ensuring unique bindings even when hardware entropy or environment fingerprints are similar across devices.*

DBRW advances without clocks:

$$\rho_i = \text{BLAKE3-256}\Big(\texttt{"DSM/dbrw-rho\0"} \parallel C_{i-1} \parallel K_{\text{DBRW}}\Big), \qquad C_i = \text{BLAKE3-256}\Big(\texttt{"DSM/dbrw-step\0"} \parallel C \tag{16}$$

with nonce $N_i$ (deterministically derived from adjacency inputs or obtained from a local entropy source; it MUST NOT be time-based). Mixing $K_{\text{DBRW}}$ into key derivations binds all signatures to the device and environment without introducing any external authority.

**Privacy Rule (normative):** 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 stitched receipt, the device writes an encrypted recovery capsule to offline media (e.g., NFC, printed QR, removable storage) as an *append-only stream*. The capsule is *transport-agnostic*; only its canonical plaintext and AEAD construction are normative.

**Canonical plaintext (normative).** Let $\mathsf{ProtoDet}(\cdot)$ denote deterministic Protobuf encoding (fixed field order, no unknown fields, no map iteration nondeterminism). Define the capsule plaintext as:

$$\text{Plain}_t := \mathsf{ProtoDet}\Big(r_t,\ \text{Meta},\ \{(\text{DevID}^{(8)}, h^{A\leftrightarrow\text{Dev}})\},\ \text{Roll}_t,\ \text{challenge}_t,\ c_t\Big), \tag{17}$$

where:

- $c_t$ is a monotone capsule index (local to the capsule stream; *not* a clock).
- $r_t$ is the Per–Device SMT root after accepting receipt $t$.
- $\text{DevID}^{(8)}$ are 8–byte truncated device digests used only for compact indexing; protocol verification binds to full 32–byte DevIDs elsewhere.
- $h^{A\leftrightarrow\text{Dev}}$ are the current relationship chain tips for each listed device relationship.
- $\text{Roll}_t$ is an accumulator over accepted receipts (defined below).
- $\text{challenge}_t$ binds the capsule to its creation context and prevents replay between streams.

**Key derivation (normative; mnemonic ring).** Let the user supply a 24–word mnemonic. Derive a fixed-length seed using a memory-hard KDF with *fixed parameters* (no time-based tuning):

$$S_{\text{mn}} := \text{Argon2id}\Big(\texttt{"DSM/recovery-ring\0"},\ \text{mnemonic\_bytes}\Big),$$

then derive the AEAD key using HKDF–BLAKE3:

$$K_R := \text{HKDF}_{\text{BLAKE3}}\Big(\texttt{"DSM/recovery-aead\0"},\ S_{\text{mn}}\Big).$$

(mnemonic_bytes is the canonical wordlist encoding; implementations MUST NOT treat locale or whitespace as significant.)

**Nonce derivation (normative; clockless).** AEAD nonces are derived deterministically from the capsule index and the current roll:

$$\text{nonce}_t := \text{BLAKE3-256}\Big(\texttt{"DSM/recovery-nonce\0"} \,\|\, \mathsf{u64le}(c_t) \,\|\, \text{Roll}_t\Big)_{0..23},$$

i.e., the first 24 bytes for XChaCha20–Poly1305. Nonce reuse is prevented by the monotone $c_t$.

**Capsule encryption (normative).** Use XChaCha20–Poly1305:

$$\text{Capt}_t = \text{XChaCha20-Poly1305.Enc}_{K_R,\ \text{nonce}=\text{nonce}_t}\Big(\text{Plain}_t;\ \text{AD} = \texttt{"DSM/recovery-capsule-v3\0"}\Big). \tag{18}$$

**Roll accumulator (normative).** Let $\text{Receipt}_t$ be the accepted stitched receipt bytes (transport envelope excluded; hash the canonical commit form). Update:

$$\text{Roll}_{t+1} = \text{BLAKE3-256}\Big(\texttt{"DSM/recovery-roll\0"} \,\|\, \text{Roll}_t \,\|\, \text{BLAKE3-256}\Big(\texttt{"DSM/receipt\0"} \,\|\, \text{Receipt}_t\Big) \,\| \tag{19}$$

where $\text{DevID}^{(8)}$ and $h'^{A\leftrightarrow\text{Dev}}$ correspond to the relationship tip(s) touched by $\text{Receipt}_t$. This binds capsule order to accepted receipt history without clocks.

**Challenge binding (normative).** Derive:

$$\text{challenge}_t := \text{BLAKE3-256}\Big(\texttt{"DSM/recovery-challenge\0"} \,\|\, \text{Roll}_t \,\|\, r_t \,\|\, \mathsf{u64le}(c_t)\Big).$$

This prevents transplanting a capsule into a different stream or context without detection.

**Tombstone and Succession.** On loss, decrypt the latest capsule to recover $(r^\star, \{(\text{DevID}^{(8)}, h)\}, \text{Roll}^\star, c^\star)$. Recovery proceeds in two deterministic operations under genesis $G$:

- **Tombstone (TR).** Publish a *Device Tree update receipt* that marks the lost device leaf $\text{DevID}_{\text{old}}$ as `TOMBSTONED` (a canonical leaf marker), producing a new Device Tree root $R'_G$. TR is valid iff it is authorized by the recovery authority defined for $G$ (e.g., threshold over designated recovery devices) and is adjacent in the Device Tree update chain.
- **Succession (SR).** Publish a second Device Tree update receipt that inserts the new device $\text{DevID}_{\text{new}}$, producing $R''_G$. SR references TR by digest and is valid only if TR is active (i.e., the tombstone marker is present in $R'_G$ and TR has not been superseded).

**Resumption of bilateral relationships.** For each stored parent tip $h$ in the decrypted capsule, the new device resumes by proposing successors *adjacent* to $h$. Acceptance remains *unique parent consumption* (Tripwire); no replay of full history is required. The recovered $\text{Roll}^\star$ provides an integrity anchor for the recovered stream.

**Security.** Receipt uniqueness ensures at most one accepted successor per parent. AEAD plus mnemonic hardness protect the capsule. Challenge binding prevents capsule replay between contexts. After TR/SR, the old device cannot extend state: any attempt fails because its DevID is tombstoned in the current Device Tree root and thus fails device-to-genesis binding checks. The roll accumulator binds recovered state to the accepted receipt stream.

### 13.1 Modal Synchronization Precedence

If $A$ performs a physical offline transaction with $B$, then $B$ must incorporate that stitched receipt before initiating a new offline transaction with $A$; otherwise $B$ would attempt to consume a different parent.

## 14 Genesis-Gated Emission Reveal and Deterministic Faucet Claims

This section replaces legacy geo-emissions and geometry attestation. DSM emissions are now specified as *genesis-gated, CPTA-bound reveals* sourced from a protocol DLV, with optional faucet-style discovery implemented via a pinned spawn registry and deterministic claim selection. There are no clocks, timestamps, or global ordering requirements.

### 14.1 Source Vault (CPTA-Bound DLV) and Fixed Supply

Let policy_commit$_{\text{ROOT}}$ be the CPTA commitment for the native token. Let $V_{\text{ROOT}}$ be a special CPTA-bound DLV that holds the full fixed supply and is *directly unspendable*; value can only leave via the emission-reveal transition predicate below.

The vault state includes:

$$\big(\text{supply\_remaining},\ i,\ \text{halving\_epoch},\ \text{vault\_chain\_tip}\big),$$

where $i$ is the emission index (monotone, adjacency-updated) and vault_chain_tip is the straight-hash-chain tip for the vault's own relationship domain (vault $\leftrightarrow$ storage anchoring relationship).

### 14.2 Spend-Gate Trigger (PaidK) and Emission Indexing

Emission is triggered only when a new genesis $G_{\text{new}}$ passes the Spend Gate: the device proves that at least $K$ distinct assigned storage nodes have accepted the paid subscription state required to store $G_{\text{new}}$'s genesis objects (the *PaidK gate*). The proof is a stitched receipt bundle over the subscription DLV(s) and node assignment roots.

Formally, define $\mathsf{PaidK}(G_{\text{new}}) = 1$ iff the verifier can validate $K$ distinct, valid, adjacent receipts showing the required paid storage commitments for $G_{\text{new}}$.

On $\mathsf{PaidK}(G_{\text{new}}) = 1$, the protocol increments $i := i + 1$ as part of the vault emission transition.

### 14.3 Halving Schedule (Clockless)

Let $H$ be the fixed halving interval in number of emissions (not time). Let base be the initial per-emission amount. Define:

$$\text{epoch}(i) := \left\lfloor \frac{i}{H} \right\rfloor, \qquad \text{amt}(i) := \left\lfloor \frac{\text{base}}{2^{\text{epoch}(i)}} \right\rfloor.$$

The vault transition MUST reject if $\text{amt}(i) = 0$ while supply_remaining $> 0$ (parameter mismatch), or if $\text{amt}(i) >$ supply_remaining. Supply is reduced deterministically:

$$\text{supply\_remaining}' = \text{supply\_remaining} - \text{amt}(i).$$

### 14.4 Regional Mapping and Pinned Spawn Registry (Optional Discovery Layer)

Each emission is mapped to a coarse region identifier region_id associated with $G_{\text{new}}$ (e.g., state/province class). The mapping is deterministic and coarse-grained; it is not a proof-of-location system.

Let $\mathcal{R}$ be a pinned spawn registry with Merkle root $\mathsf{Root}_{\text{spawn}}$. Each spawn entry is:

$$\mathsf{spawn} = (\mathsf{spawn\_id},\ \mathsf{region\_id},\ \mathsf{cooldown\_class},\ \mathsf{capacity},\ \text{policy\_commit})\,.$$

The registry is an application-visible distribution map; protocol enforcement only requires inclusion proofs under $\mathsf{Root}_{\text{spawn}}$.

For emission index $i$, define a deterministic spawn selection inside the region:

$$\mathsf{spawn\_id}(i) := \mathsf{SelectSpawn}\Big(\text{BLAKE3-256}\big(\texttt{"DSM/emit-spawn\0"} \,\|\, \mathsf{u64le}(i) \,\|\, \text{region\_id} \,\|\, G_{\text{new}}\big),\ \text{region\_i}$$

where $\mathsf{SelectSpawn}(\cdot)$ is a fixed, published selection function over the ordered spawn list for region_id.

### 14.5 Faucet Claim Capsule (Deterministic, No Geo-Attestation)

A claim is a one-shot capsule that binds the claimant to the spawn and to its current adjacency state. Define:

$$\mathsf{cap}_{\text{claim}} = (\mathsf{spawn\_id},\ G_{\text{claimant}},\ \text{DevID}_{\text{claimant}},\ \mathsf{chain\_tip},\ C,\ \sigma,\ \pi_{\text{spawn}},\ \text{policy\_commit})\,,$$

with:

$$C = \text{BLAKE3-256}\big(\texttt{"DSM/faucet-claim\0"} \,\|\, \mathsf{spawn\_id} \,\|\, G_{\text{claimant}} \,\|\, \text{DevID}_{\text{claimant}} \,\|\, \mathsf{chain\_tip} \,\|\, \text{policy\_con}$$

and $\sigma$ a SPHINCS$^+$ signature under the claimant's step key for $\mathsf{chain\_tip}$ (Sec. 11.1). The spawn inclusion proof $\pi_{\text{spawn}}$ proves $\mathsf{spawn\_id}$ under $\mathsf{Root}_{\text{spawn}}$.

*Local-only gates:* cooldown, proximity, and sensor checks (walk-and-claim UI behavior) are device-local policy gates. They MUST NOT be treated as globally verified predicates in this version of the spec. The protocol-level invariants are: (1) spawn authenticity via $\pi_{\text{spawn}}$, (2) device binding via signatures and DevID inclusion, and (3) capacity enforcement via deterministic selection below.

### 14.6 Deterministic Capacity Enforcement (No Timestamps)

Let $\mathcal{C}_{\mathsf{spawn}}$ be the set of valid claim commitments $C$ for a spawn observed by verifiers. Define a deterministic first-writer-wins selector:

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

where $c = \mathsf{capacity}$ from the spawn entry. Only claims whose $C$ lies in $\mathsf{FWW}_c(\mathcal{C}_{\mathsf{spawn}})$ are eligible to be applied as emission recipients. This rule is purely set-based; no timestamps or global ordering are referenced.

### 14.7 Emission Application (Vault Transition)

Minting/reveal is an ordinary DSM token transition sourced from $V_{\text{ROOT}}$:

$$V_{\text{ROOT}} \xrightarrow{\mathsf{cap}_{\text{claim}},\ \mathsf{PaidK}(G_{\text{new}})} V'_{\text{ROOT}}$$

that transfers amt($i$) from the vault to $G_{\text{claimant}}$ under policy_commit$_{\text{ROOT}}$, updates $i := i + 1$, and reduces supply_remaining. Replaying an already-consumed claim has no effect (idempotence) because it would require consuming an already-consumed parent in the vault transition domain (Tripwire).

**Theorem 6** (Deterministic Capacity Enforcement)**.** *For any spawn with capacity $c$, across all honest verifiers applying the same pinned spawn registry root and the same set $\mathcal{C}_{\mathsf{spawn}}$, only claims with commitments in $\mathsf{FWW}_c(\mathcal{C}_{\mathsf{spawn}})$ can be applied as emission recipients, 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*:

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

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 $C_{\text{pre}}$ 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 $K_{\text{DBRW}}$, 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 roll accumulator anchors recovered state integrity.

### 15.9 Additional Guarantees

Auditing can enumerate stitched digests between indices; proofs remain logarithmic. Reputation or rate–limits can be computed from local deterministic counters or bounded 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 stable ABI surfaces for mobile and other bindings.

- **`core/`**: Genesis/device creation, relationship (pair) straight-hash-chains, Per–Device SMT maintenance, Device Tree verification.
- **`crypto/`**: Post–quantum primitives (Kyber KEM, SPHINCS+ signatures), BLAKE3, HKDF–BLAKE3, Argon2id, AEAD.
- **`receipt/`**: Stitched receipts, inclusion proofs, canonical commit bytes, Tripwire enforcement.
- **`bilateral/`**: Offline co–sign flow (BLE/NFC transport-agnostic), conflict detection, local apply.
- **`unilateral/`**: Online unilateral submit/retrieve using `b0x[...]` spool keys.
- **`recovery/`**: Tombstone/Succession, recovery capsule AEAD stream, DLV primitives relevant to recovery predicates.
- **`bridge/`**: FFI/JNI surface; protobuf-in/protobuf-out, no re-encoding of commits.

## 16.2 SDK Architecture and Build Targets

**Scope and authority (normative).** The Rust protocol core crate `dsm_core` is the **sole execution authority** for: (1) canonical commit bytes, (2) acceptance predicates, (3) Merkle and SMT inclusion proof verification, (4) Per–Device SMT *replace* semantics, (5) signature creation/verification, and (6) all KDFs (HKDF–BLAKE3, Argon2id). Platform SDKs and bindings are non-authoritative shims that *MUST* delegate these operations to `dsm_core` and *MUST NOT* re-implement 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 (`.proto`), and developer tooling around `dsm_core`. The SDK is defined independently of any one platform.

### 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; exported symbols are C/JNI ABI-stable and versioned.
- **JNI wrapper:** `DsmNativeWrapper.kt` exposes a minimal surface: `init_device`, `submit_online`, `co_sign_offline`, `verify_receipt`, `get_per_device_root`, `prove_incl`.
- **Hardware shim:** Kotlin mediates BLE/NFC, GNSS/UI sensors, and storage I/O; `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, no base64, and no hex encodings are 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/Kotlin. 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; the SDK remains platform-agnostic.

### 16.2.5 Binding constraints (normative)

- Bindings *MUST* call `dsm_core` for canonical commit emission, receipt verification, inclusion proof verification, SMT replace, signatures, and all KDFs.
- Bindings *MUST NOT* hash/sign re-encoded payload bytes; they *MUST* pass canonical commit bytes emitted by `dsm_core` (Sec. 16.5).
- Versioning is pinned: application code depends on `DSM_SDK@v`, which pins `dsm_core@v`; semantic upgrades are coordinated and explicit.
- **Encoding ban:** JSON, base64, hex, and Serde-derived canonicalization are forbidden on the protocol path (wire, storage keys used for protocol addressing, and acceptance predicates).

**Storage Nodes** Storage nodes are *dumb, signature–free* persistence surfaces. They store Device Tree material, Per–Device SMT mirrors (aggregated), `b0x` spools, ByteCommit chains, and recovery capsules. They do *not* evaluate acceptance predicates. Validation is device-side; nodes persist and serve bytes. Censorship resistance derives from replication plus 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.* Leaves are 32-byte DevID values sorted lexicographically (big-endian byte order).

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

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

*Empty tree root.*

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

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

### 16.4 Addressing and Online Unilateral Transport (`b0x[...]`)

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

**Key format (normative; no hex/base64).** The `b0x` key is a *string key for storage indexing only*; it is not a verification primitive. It is formed by base32 encoding (Crockford base32, uppercase, no padding) of three 32-byte digests:

$$\text{addr}_{A\to B} := \texttt{b0x[}\,\mathsf{B32}\Big(\text{BLAKE3}\Big(\texttt{"DSM/addr-G\0"} \,\|\, G \,\|\, \text{salt}_G\Big)\Big)\,.\,\mathsf{B32}\Big(\text{BLAKE3}\Big(\texttt{"DSM/addr-D\0"} \,\|\, \text{DevID}$$

Here $h_n$ is the current relationship chain tip digest for $(A \leftrightarrow B)$, and nonce is sender-chosen per submission. $\mathsf{B32}(\cdot)$ is a pure text encoding; it carries no security assumptions.

**Salt derivation (normative).** Per-user blinding salts are derived inside `dsm_core` and never exposed:

$$\text{salt}_G := \text{HKDF}_{\text{BLAKE3}}\Big(\texttt{"DSM/b0x-salt-G\0"},\, S_{\text{master}}\Big), \quad \text{salt}_D := \text{HKDF}_{\text{BLAKE3}}\Big(\texttt{"DSM/b0x-salt-D\0"},\, S_{\text{m}}$$

where $S_{\text{master}}$ is the device's master secret seed (Sec. 11.1).

**Rotation and privacy.** Each stitched receipt advancing $(A \leftrightarrow B)$ changes $h_n$, rotating the final component and thus the full `b0x[...]` key. Only counterparties tracking the live tip can derive the current spool key; storage nodes store opaque bytes keyed by `b0x[...]` and learn no relationship metadata from blinded components.

**Retrieval and stitching.** Upon sync, $B$ derives the current key from $(G, \text{DevID}_B, h_n)$, fetches pending submissions stored under that key, verifies receipts (inclusion proofs, certificates, signatures), and stitches them to advance the tip. Submissions addressed to older tips are queued (waiting predecessors) or rejected as invalid against the current state.

**Modal lock (relationship-local).** If any pending online submission exists for $(A, B)$ under the latest derived `b0x[...]` key (including the local queue), offline $(A, B)$ transactions are invalid until those pending items are synchronized and either stitched or rejected. Other relationships are unaffected.

**No clocks, no heights.** Address derivation and admissibility are purely hash-chain driven. No wall clocks, timestamps, epochs, or heights appear in any predicate.

### 16.5 Canonical Encoding and Protobuf Pipeline

DSM is protobuf-only on all authoritative paths. The system distinguishes: (i) *transport envelopes* (protobuf messages for network/app plumbing), and (ii) *canonical commit bytes* (protobuf messages with strict determinism rules emitted only by `dsm_core`).

- **Transport:** Protobuf envelopes only (Envelope wire v3). Schemas are defined in canonical `.proto` and code-generated for Rust/Kotlin/TypeScript.
- **Canonical commit bytes (normative):** Every hashed/signed object has a *CommitV3* protobuf form with: (a) no `map` fields, (b) all repeated fields either preserved in protocol-defined order or explicitly lexicographically sorted, (c) no unknown fields, (d) fixed domain tags, and (e) deterministic encoding produced by `dsm_core`. All hashing/signing is over these commit bytes, never over ad-hoc re-encodings.
- **Bindings:** Platform code forwards opaque commit bytes produced by `dsm_core`; it does not re-encode commits.
- **Terminology:** Use "inclusion proof" everywhere (never "membership proof").
- **Encoding ban:** JSON/base64/hex are forbidden on the protocol path (wire verification, commit hashing, receipt signing, storage addressing keys).

### 16.6 Ordering and Concurrency (No Clocks, No Heights)

DSM uses the bilateral straight hash chain itself for strict ordering; no timestamps or heights appear in any predicate. Concurrency is resolved by stitched receipts and Per–Device SMT *replace*:

1. Each proposed successor at tip $h_n$ carries a pre-commit $C_{\text{pre}} = \text{BLAKE3}\big(\texttt{"DSM/pre\0"} \,\|\, h_n \,\|\, \text{op} \,\|\, e\big)$ and an inclusion proof that $h_n$ is the current Per–Device SMT leaf.
2. The successor $h_{n+1} = \text{BLAKE3}\big(\texttt{"DSM/tip\0"} \,\|\, h_n \,\|\, \text{op} \,\|\, e \,\|\, \sigma\big)$ 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 consuming the same $h_n$ that is not bit-identical is rejected by the device-local SMT replace rule (Tripwire).

**Deterministic rate limits without time (optional, non-consensus).** Policies may impose *counter-based* limits (e.g., per-relationship step counters) and/or *work-unit* gates defined as a fixed number of BLAKE3 iterations. These are strictly local predicates or vault predicates parameterized by integers, never by wall time. No calibration against time is permitted.

### 16.7 Key Management, DBRW Binding, and SPHINCS+

Per-step key derivation is specified in Sec. 11.1 and Sec. 12. In summary:

- Device-bound secret $K_{\mathrm{DBRW}}$ derives from hardware and environment (DBRW) and is never serialized, logged, or committed.
- Master seed $S_{\mathrm{master}}$ derives via HKDF–BLAKE3 from $(G, \mathrm{DevID}, K_{\mathrm{DBRW}}, s_0)$.
- For each parent $h_n$ and pre-commit $C_{\mathrm{pre}}$, a per-step seed $E_{n+1}$ derives via HKDF–BLAKE3 from $(h_n, C_{\mathrm{pre}}, k_{\mathrm{step}}, K_{\mathrm{DBRW}})$, where $k_{\mathrm{step}}$ is derived from Kyber shared secret material.
- Ephemeral SPHINCS+ keys are 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 parent and DBRW binding.

**Receipts (Per–Device SMT replace).** For $(A \leftrightarrow B)$ at tip $h_n$, a stitched receipt carries: (i) old/new tips $(h_n, h_{n+1})$, (ii) old/new Per–Device SMT roots $(r_A, r'_A)$ (and symmetrically $r_B, r'_B$ when required), (iii) inclusion proofs for the old and new leaves, (iv) Device Tree inclusion for signing DevIDs under $R_G$, (v) the EK certificate chain data, and (vi) two SPHINCS+ signatures over the canonical commit bytes of the receipt body. If any inclusion proof fails or SMT replace does not recompute the advertised root, the receipt is invalid.

### 16.8 Offline vs. Online Flows

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

**Online (unilateral, `b0x[...]` spool).** Sender posts a unilateral submission to the derived `b0x[...]` key. The recipient syncs, verifies proofs and signatures, then stitches and applies. The relationship-local modal lock forbids starting an offline transaction for $(A, B)$ while pending online projections exist for $(A, B)$.

### 16.9 Storage Nodes and Censorship Resistance

Storage nodes expose protobuf-only endpoints to store/fetch: (i) Device Tree material and roots, (ii) Per–Device SMT mirrors (aggregated), (iii) `b0x[...]` spool items, (iv) recovery capsules, and (v) ByteCommit chains.

Nodes never validate acceptance predicates. Censorship resistance follows from: (1) client-side verification of all fetched bytes and proofs, (2) deterministic replica placement and PaidK

gates (Sec. 10), and (3) multi-node replication: if one node refuses, the same self-verifying protobuf object is relayed to other nodes.

### 16.10 Recovery Capsule AEAD and DLV

Recovery capsules are defined in Sec. 13. This subsection pins implementation choices for the SDK.

**AEAD choice (normative).** Use XChaCha20–Poly1305 with a 256-bit key $K_R$ derived from the mnemonic ring key derivation, and a 24-byte nonce derived deterministically from the capsule counter and roll accumulator (Sec. 13).

**Associated data (normative).**

$$\text{AD} := \texttt{"DSM/recovery-capsule-v3\0"} \parallel r_t \parallel \mathsf{u64le}(c_t).$$

Associated data is authenticated but not encrypted; it binds the capsule to the current Per–Device SMT root and capsule index without clocks.

**Nonce uniqueness (normative).** Nonce reuse is forbidden. Uniqueness is enforced by monotone counter $c_t$ per capsule stream and deterministic nonce derivation under $K_R$.

### 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/mai build -r`.
- **JNI:** Minimal surface in `DsmNativeWrapper.kt`; all parameter validation and all cryptographic operations occur in Rust.
- **Protobuf:** Generate Rust/Kotlin/TypeScript types from the canonical `.proto`. Envelope wire is pinned to v3 only.
- **Frontend:** `pnpm build`; assets bundled into Android assets for WebView.
- **Testing:** Rust unit/integration tests for SMT replace and receipt verification; Android instrumentation tests for BLE/NFC shims; end-to-end tests proving offline/online parity for the same relationship domain.

### 16.12 Operational Parameters (Recommended)

- **Hash:** BLAKE3 (256-bit digests) for commits and deterministic counters.
- **Signatures:** SPHINCS+ (per-step, deterministic derivation; size capped as specified in Sec. 11.1).
- **KEM:** Kyber for step secrets; secrets never serialized.
- **SMT:** 256-bit key space; inclusion proofs logarithmic; device-local authoritative.
- **Device Tree:** Standard Merkle; replicated to storage nodes and user devices.
- **Entropy:** $s_0$ and $s_{\text{device}}$ from CSPRNG; per-step seeds via HKDF–BLAKE3 over $(h_n, C_{\text{pre}}, k_{\text{step}}, K_{\text{DBRW}})$.
- **Time:** No timestamps, epochs, or heights in predicates or encodings.
- **Modal rule:** Pending online for $(A, B)$ blocks offline for $(A, B)$ until synchronized; other relationships commute.

**Summary.** DSM's implementation is mobile-first: Rust compiles 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 (Envelope v3). All cryptographic commits are *canonical protobuf commit bytes* emitted and verified solely by `dsm_core`. Ordering is enforced by bilateral hash adjacency and Per–Device SMT replace, not by time or height. SPHINCS+ is per-step and deterministically derived with Kyber and DBRW binding.

## 17 Conclusion

DSM is a clockless bilateral trust fabric with two Merkle layers: a replicated Device Tree that binds DevIDs to a single genesis, and a Per–Device SMT that indexes relationship domains and their linear straight 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[...]` spool keys; 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 ($C_{\text{pre}}$)** 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 ($C_{\text{pre}}$) and stitched receipts

**pre–commit forking** authoring mutually exclusive pre–commit ($C_{\text{pre}}$) 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:

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

## 18.1 State Snapshot (before any example)

On device $A$ (Alice):

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

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 $\alpha$ tokens to Bob *offline*. Both are co-present.

**Step 1: Pre-commit.** Alice proposes operation $\text{op} = \texttt{Transfer}(\alpha)$ 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 $C_{\text{pre}}$, 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 $\text{DevID}_A$, (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).**

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

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 $\text{addr}_{A\to B}$ 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 $\text{addr}_{A\to B}$ from his $G_B$, $\text{DevID}_B$, 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).**

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

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* $\sigma$ is the minimal stitched evidence set that references $X$ and the DLV commit.

**Unlock key derivation (deterministic).**

$$\text{sk}_V = H(L \,\|\, C \,\|\, \sigma).$$

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

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

**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 $\sigma$, 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.