The Deterministic State Machine Primitive
Boundary, Definition, and Composition

Brandon "Cryptskii" Ramsay

March 10, 2026

Abstract

This note defines the Deterministic State Machine (DSM) as a primitive rather than as a wallet, storage network, or generalized blockchain replacement. The primitive is the smallest cryptographic object that DSM needs in order to produce deterministic, locally verifiable, non-forking state transitions between devices without validators, consensus, or trusted servers. The paper draws a hard boundary around what is part of the primitive, what merely transports or stores primitive outputs, and what higher-order protocols are composed on top of it.

1 Why This Boundary Matters

The repository contains several layers: the pure Rust core, JNI and bridge adapters, storage node infrastructure, Bluetooth and online delivery paths, higher-order constructions such as DLVs and dBTC, and end-user wallet code. Without a clean boundary, contributors tend to blur three different questions:

1. What is the irreducible cryptographic primitive?
2. What infrastructure merely carries or persists primitive artifacts?
3. What application semantics are built from the primitive but are not the primitive itself?

The answer matters for architecture, proofs, threat modeling, open-source onboarding, and review. If a change modifies the primitive, it changes the system's security model. If a change only modifies transport, storage, or application semantics, it should not be allowed to silently mutate the primitive's guarantees.

2 Thesis

DSM is a decentralized, relationship-local state transition primitive.

More precisely, DSM provides a way for a pair of devices to:

1. bind device identity to a user genesis,
2. commit the current state of each bilateral relationship under device-local authenticated structures,
3. advance one relationship by producing a deterministic successor that embeds its parent,

4. attach inclusion proofs and signatures to that successor in a stitched receipt, and
5. reject every conflicting successor to the same parent except the unique accepted one.

Everything else in the repository either supports this primitive or composes on top of it.

3 Minimal Objects of the Primitive

Definition 1 (Genesis)**.** *A genesis digest $G \in \{0,1\}^{256}$ is the root commitment that binds the owner's device set and anchors subsequent device identity and state.*

Definition 2 (Device Identifier)**.** *Each device A has a stable identifier DevID_A derived from a post-quantum attestation key and device-specific binding material. Device identifiers are leaves in the owner's Device Tree.*

Definition 3 (Device Tree)**.** *The Device Tree is a standard Merkle tree with root R_G whose leaves are the device identifiers bound to genesis G. It proves device-to-genesis membership.*

Definition 4 (Per-Device SMT)**.** *Each device A maintains a Sparse Merkle Tree with root r_A. Its leaves map bilateral relationship keys (A, B) to the current relationship tip digest $h_{A\leftrightarrow B}$.*

Definition 5 (Relationship Domain)**.** *For devices A and B, the relationship domain $R_{A,B}$ is the straight hash chain plus the associated inclusion proofs needed to show the current tip under the relevant per-device roots. DSM has no monolithic global application state; it is the disjoint union of these relationship domains.*

Definition 6 (Stitched Receipt)**.** *A stitched receipt is the canonical proof-carrying artifact that binds:*

1. *the parent tip h_n and child tip h_{n+1},*
2. *the per-device root transition $r_A \to r'_A$,*
3. *inclusion proofs for the old and new tips,*
4. *device membership proof under R_G, and*
5. *the required post-quantum signatures.*

4 Acceptance Predicate

The primitive is not defined by UI flow or by how bytes travel over the network. It is defined by the acceptance predicate over canonical bytes, proofs, and signatures.

For a candidate transition $\tau_{A,B}$ from parent tip h_n to child tip h_{n+1}, the primitive accepts only if:

$$\mathsf{Accept}(\tau_{A,B}) = 1$$

when all of the following hold:

1. **Adjacency.** h_{n+1} is the canonical successor of h_n and embeds the parent under the required deterministic encoding.

2. **Current-state proof.** The old tip h_n verifies as the current leaf under the advertised per-device root.

3. **Replace proof.** Replacing h_n with h_{n+1} recomputes the advertised new per-device root byte-exactly.

4. **Device binding.** The acting device proves inclusion in the Device Tree for genesis G.

5. **Signature validity.** The required device-side signatures verify over the canonical commit bytes.

6. **Fork exclusion.** No distinct conflicting successor to h_n can also satisfy the same acceptance predicate except with negligible probability under the security assumptions.

This is the primitive. A byte stream that does not satisfy this predicate is not a valid DSM state transition no matter where it came from. A byte stream that does satisfy this predicate remains valid no matter which untrusted storage node, relay, or transport path delivered it.

5 What Is Inside the Primitive

The following elements are part of the primitive itself:

1. **Canonical deterministic encoding.** The primitive is byte-sensitive. Canonical protobuf commit bytes, domain-separated hashing, and strict serialization rules are inside the primitive because acceptance depends on them.

2. **Device-to-genesis binding.** Genesis, device identifiers, the Device Tree, and device membership proofs are inside the primitive because they define who is authorized to advance state.

3. **Relationship-local authenticated state.** Per-device SMT roots and relationship tip mappings are inside the primitive because they define the authenticated current state.

4. **Hash-chain ordering.** Parent embedding, straight-hash-chain progression, and logical ordering by adjacency are inside the primitive because they define what it means for one state to succeed another.

5. **Proof-carrying transitions.** Stitched receipts are inside the primitive because they are the object a verifier actually checks.

6. **Tripwire fork exclusion.** The unique-successor property is inside the primitive because without it DSM would not be a safe state machine.

7. **Clocklessness.** Rejection of wall-clock time, global heights, and validator rounds from acceptance predicates is inside the primitive because it is part of the definition of valid progression.

8. **Infrastructure blindness.** Storage nodes and relays must be unable to create, validate, or authorize state transitions. That exclusion is part of the primitive's boundary, not a deployment preference.

6 What Is Outside the Primitive

The following are not part of the primitive, even though they may be necessary in a full system:

1. **Transport.** BLE, NFC, WebView bridge plumbing, JNI, HTTP, MessagePort, and b0x delivery are delivery mechanisms. They move bytes but do not define validity.
2. **Storage and replication.** Storage nodes, ByteCommit mirroring, replica placement, capacity accounting, and recovery of stored objects are availability infrastructure, not acceptance authority.
3. **Application UX.** Wallet screens, onboarding, routing, toasts, explorers, and presentation logic are outside the primitive.
4. **Economic policy.** Pricing, subscriptions, operator compensation, and market-facing token economics are outside the primitive.
5. **Higher-order protocols.** DLVs, CPTA policies, dBTC, DJTE, recovery capsules, and other protocol constructions are built from the primitive's receipt model. They are important protocol systems, but they are not the irreducible primitive.

Remark 1. *The last point is the most important boundary in the repository. A DLV is not the primitive. A Bitcoin bridge is not the primitive. Emissions are not the primitive. They are all deterministic programs written in the language of stitched receipts, authenticated roots, and relationship-local state progression.*

7 Why DSM Is Decentralized

DSM is decentralized in a stronger sense than merely "many nodes exist." The primitive has no global validator set, no sequencer, no leader, no block producer, and no globally shared mutable object that every participant must agree on before a transition is meaningful.

Authority is distributed to the endpoints of each relationship domain and to any verifier who can check the corresponding proofs. A transition is valid because it satisfies the acceptance predicate, not because an institution, quorum, or server says so.

Proposition 1 (Relationship-local decentralization)**.** *DSM decentralizes acceptance by confining validity to bilateral proofs over local authenticated state rather than to votes over a global shared state.*

This is why the statelessness argument in the companion research notes matters: DSM exits the global-succinct-commitment model entirely. It does not attempt to compress one world-state into a single validator commitment. Instead, it partitions state into independent relationship domains and verifies each domain locally.

8 The Primitive Boundary as a Composition Stack

Layer	Role	Examples in this repository
Primitive	Defines validity and non-forking state progression	Genesis binding, DevID, Device Tree, Per-Device SMT, canonical commit bytes, stitched receipts, Tripwire
Delivery	Carries primitive artifacts between parties	BLE, NFC, b0x spool keys, bridge RPC, JNI plumbing
Availability	Stores and mirrors primitive artifacts	Storage nodes, ByteCommit, replication, recovery object persistence
Composed protocol semantics	Expresses domain-specific state machines using the primitive	Tokens, CPTA, DLVs, dBTC, DJTE, recovery workflows
Application	Surfaces the system to developers and end users	Android app, React UI, onboarding, explorer links, operational tooling

The composition rule is simple: only the first layer decides what a valid transition is. Every higher layer may package, store, relay, or use valid transitions, but it cannot redefine validity without changing the primitive.

9 Repository Boundary

In this repository, the implementation boundary is anchored in the pure Rust core and enforced by the single authoritative path:

$$\text{UI} \rightarrow \text{MessagePort} \rightarrow \text{Kotlin bridge} \rightarrow \text{JNI} \rightarrow \text{SDK} \rightarrow \text{Core}$$

The primitive lives primarily in the core crate: deterministic encoding, domain-separated hashing, Merkle proofs, relationship-local state progression, and verification logic. The SDK and bridge mediate I/O. Storage nodes are deliberately outside the trust boundary. The frontend is entirely outside the trust boundary.

This boundary gives contributors a practical rule:

1. If a change alters acceptance, ordering, proofs, identity binding, or fork exclusion, it is a primitive change.
2. If a change only alters transport, storage, UI, or operator workflow, it is not a primitive change.
3. If a higher-order feature needs a new security guarantee, the first question is whether it can be expressed as a deterministic predicate over primitive artifacts rather than by adding trusted infrastructure.

10 Closure Rule

The primitive should be treated as closed by default.

This is not a style preference. It follows from the role of a primitive. A primitive is the narrow object whose guarantees other systems depend on. Once it becomes a rolling feature surface, every "extension" becomes a security-model change, a review burden, and a proof burden.

Accordingly, DSM should reject the idea of primitive expansion for convenience, product growth, or future optionality. There should be no speculative extension hooks in the primitive, no spare acceptance branches waiting for later use, and no "just in case" encoding flexibility. If a new capability can be expressed by composing stitched receipts, authenticated roots, and deterministic predicates above the primitive, then that is where it belongs.

More strongly: DSM should treat the primitive as something that must be nailed down early. The point of the primitive is not to serve as a future feature host. The point is to define a fixed acceptance predicate that higher-order systems can rely on without renegotiating its meaning every time a new idea appears.

The only defensible reasons to modify the primitive are:

1. to repair a soundness, verification, or ambiguity bug,
2. to simplify the primitive while preserving its acceptance surface, or
3. to replace a broken cryptographic or formal assumption while retiring the old path rather than extending it.

11 Why Feature Accretion Breaks the Primitive

The failure mode is not merely that the codebase becomes larger. The real failure is that the meaning of validity becomes less fixed.

Once features are added at the primitive layer, the acceptance surface widens. A wider acceptance surface means more valid states, more branches, more interactions, and more room for ambiguity. Invariants that were once crisp become conditional on feature combinations. Review shifts from checking a narrow predicate to checking a moving platform. At that point the system has stopped behaving like a primitive and started behaving like an execution environment.

Two different families illustrate the point:

Ethereum and Turing-complete expansion. Ethereum demonstrates one form of primitive drift: once the validation layer executes broadly programmable user logic, validity depends on interpreter semantics, resource metering, call structure, and cross-contract interactions. This creates a very powerful system, but it is no longer a narrow primitive in the sense used here. The set of possible behaviors is too large to analyze with the crispness expected of a primitive.

Bitcoin and bounded script expansion. Bitcoin illustrates a different, narrower case. Even without Turing completeness, adding more validation-layer script surface still has consequences. New opcodes, new script versions, covenant-like semantics, and script-based protocol proposals can all enlarge the space of valid constructions and the burden of review, coordination, and interpretation. The issue is therefore not only Turing completeness. The issue is that the validation layer itself becomes the place where features keep accumulating.

Implication for DSM. DSM should learn from both cases. It should avoid the Turing-complete path, but it should also avoid the softer trap of validation-layer feature accretion through bounded

scripting or interpreter-like extensions. A DSM scripting language, exchange sub-language, or programmable acceptance engine would shift the protocol away from a fixed acceptance predicate and toward interpreter semantics. Even if bounded, that would still widen the acceptance surface and weaken the claim that the primitive is nailed down.

The decisive point is that non-Turing-complete is not, by itself, enough. A bounded language can still become the vehicle for endless validation-layer accretion. Once the system becomes the place where every useful idea asks for "just one more" accepted branch, opcode, or programmable rule, the primitive has already been lost. It may still be useful software, but it is no longer the kind of narrow, analyzable primitive this document is trying to define.

12 Checklist for Primitive Changes

A proposal belongs inside the primitive only if at least one of the following is true:

1. It changes the canonical bytes that verifiers hash or sign.
2. It changes device-to-genesis authorization.
3. It changes the authenticated representation of current relationship state.
4. It changes the acceptance predicate for a state transition.
5. It changes the unique-successor or no-fork guarantee.

If none of those are true, the proposal is almost certainly not primitive. It may still be useful, but it should be modeled as transport, storage, tooling, UX, or a higher-order protocol.

Even if one of those conditions is true, that still does not justify expansion. The next question must be whether the change is a repair, a clarification, or a forced replacement of a broken assumption. If the answer is "no," the change should not enter the primitive.

13 Conclusion

The Deterministic State Machine primitive is the narrow cryptographic core that makes DSM what it is: a clockless, proof-carrying, relationship-local, non-forking state transition system with device-bound authority and no global validator dependence.

That boundary should remain explicit. Contributors should be able to say, for any feature or code path, whether it belongs to:

1. the primitive,
2. the infrastructure that carries the primitive, or
3. the higher-order systems built from it.

Once that distinction is kept clean, the repository becomes easier to reason about, easier to document, easier to secure, and much easier to open to outside contributors.