# Deterministic State Machines Explained Mathematically

Bilateral State, Sparse Merkle Trees, Determinism, Precommitment,
Linear Resources, Canonical State Chaining, and Conflict-Local Finality

Brandon Ramsay
Irrefutable Labs Inc.

**Abstract**

The Deterministic State Machine (DSM) is a guarded, linear, forward-only, constraint-based state architecture. Its purpose is to make state validity derivable from canonical cryptographically committed state rather than from a global ordering authority.

DSM does not require a blockchain, validator set, sequencer, mempool, gas market, clock-based transaction ordering, or global consensus mechanism for ordinary state evolution. Instead, validity is established locally from canonical bytes, bilateral relationship chains, Sparse Merkle Tree (SMT) commitments, cryptographic precommitment of candidate futures, deterministic guards, explicit linear-resource consumption keys, signatures, policies, and proofs.

The important architectural change is not merely that DSM orders fewer things. DSM removes global ordering from the state-validity mechanism. Where an application itself requires an ordering relation, that relation can be represented explicitly and deterministically inside committed state or policy rather than being supplied by a universal external transaction-ordering system.

Multiple possible futures may be precommitted from one parent state. However, conflicting futures over the same linear resource are forced to derive the same authoritative resource-consumption key. Once one such resource is consumed in a realized state lineage, a second conflicting consumption contradicts the validity rules.

The mathematics needed to understand the construction is intentionally elementary. Functions, sets, Boolean logic, hashes, trees, algebra, and proof by contradiction are sufficient to understand the principal safety theorem.

Throughout the paper, shaded comparison boxes explain how Bitcoin solves the corresponding problem and where DSM differs architecturally. These comparisons do not rely on claiming that Bitcoin is defective. Bitcoin deliberately solves global decentralized monetary ordering with proof of work and UTXOs. DSM solves the state-transition problem differently: by making conflict, linearity, authority, and admissible futures properties of authenticated state itself.

## Contents

# Part I
# What Is Claimed, and Who Decides

The reader's objections, answered in the order they occur. The mathematical claims used here are constructed and proved in Part II. Operational claims about storage, position binding, threshold evidence, and offline identity are stated explicitly as protocol assumptions and implementation obligations, and are revisited in Parts IV and V.

## 1 The Central Question

A distributed system must answer a basic question:

> If several machines can propose changes to state, how does a verifier know which changes are valid?

A common answer is to construct a universally ordered history.
For example,

$$T_1 < T_2 < T_3 < T_4 < T_5.$$

If everybody accepts the same ordering, then a conflict can be resolved by asking which transaction came first in the accepted history.

DSM takes a fundamentally different approach.
It asks:

> Can the validity and conflict rules be represented directly inside cryptographically committed state so that a verifier does not need a global ordering service at all?

DSM's answer is yes.
The core safety objective is:

| For one committed linear resource, two conflicting consumptions cannot both occur in one valid realized history. |
|---|

This does not require every operation in the entire system to be placed into one total order.

Instead, each transition carries enough canonical structure for a verifier to determine whether it is a valid continuation of the state it claims to extend.

### 1.1 Candidate Futures Versus Realized Futures

DSM permits multiple candidate futures.

Suppose state $s$ has candidates

$$P(s) = \{c_1, c_2, c_3\}.$$

All three candidates may be legitimate possibilities.
But possibility is not realization.

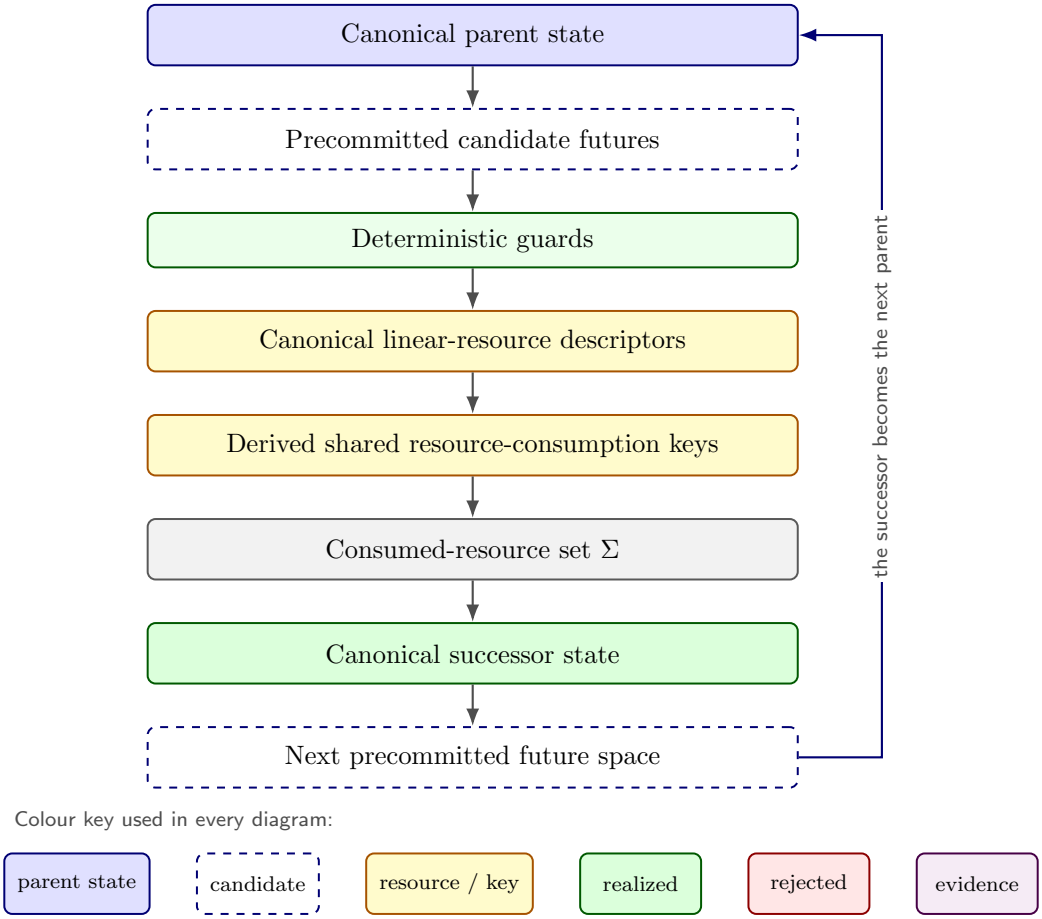
| candidate future $\neq$ realized future |
|---|

The safety property concerns which candidates can become part of one valid realized state lineage.

**Bitcoin Comparison**

Bitcoin solves double-spend prevention by maintaining a shared proof-of-work blockchain from which the accepted UTXO state is derived.
Conflicting transactions may exist, but the accepted chain history determines which spend survives.
DSM does not use a global chain-selection rule as the state-validity mechanism.
Instead, a conflicting set of DSM branches is tied to a common linear resource. The shared resource is consumed once, and the canonical successor state records that fact.
**Architectural consequence:**
DSM removes the requirement that unrelated state transitions participate in one common ordering contest.
The conflict rule travels with the state resource itself.

## 2 The Entire DSM Idea in One Diagram




The system therefore forms a repeating cycle:

$$\boxed{\text{state} \rightarrow \text{candidate space} \rightarrow \text{guarded realization} \rightarrow \text{resource consumption} \rightarrow \text{new state}}$$

## 3 Six Answers Before the Mathematics

A reader coming from Bitcoin or Ethereum will not read this document in the order its mathematics is built. They will read it with a list of objections, and each time an objection is not answered on the

page where it occurs they will fill the gap themselves, usually with "validator set," "hidden consensus," "payment channel," or "trusted database." The rest of the document would then be spent undoing a misconception it allowed to form.

So the answers come first, in one table, with the section that earns each one. Everything after this section is explanation, not suspense.

| The question the reader has | The answer, and where it is earned |
|---|---|
| What replaces global chain ordering? | Deterministic derivability from committed state. A verifier recomputes validity from the parent, the candidate, and proofs; it never asks which transaction came first in a shared history. (Sections 5, 51) |
| What stops two branches consuming one resource? | They derive the same consumption key $K$ from the same committed parent and the same resource descriptor. Realization adds $K$ to a monotonic consumed set $\Sigma$; the second branch fails the check $K \notin \Sigma$. This is a theorem, not a policy. (Sections 6–7, 51) |
| What if Bob and Charlie see different branches? | Alice's balance is one resource in Alice's own device root, and that root carries a position. Each position is bound to exactly one root in a write-once cell that only Alice's device can write and nobody can rewrite. Charlie reads the next cell before accepting. Two phones do not help her. (Section 8) |
| Who determines whether a transition is valid? | The recipient. Bob verifies what Alice presents against his own copy of the frontier and the committed policies. No third party is consulted for validity, and no third party could be. (Sections 4, 10, 11) |
| What does storage determine? | Persistence, availability, attribution, and per-key non-equivocation. A storage node never signs, never evaluates a guard, never computes a balance, and does not interpret transaction or economic payload semantics. It stores bytes and, for a few write-once registers, refuses to hold two values for one key. (Section 10) |
| What does "quorum" mean here? | The client contacts the storage members that the resource's owner committed to, verifies each response, and counts locally whether enough distinct members hold the same fact. The members do not participate in quorum formation: they do not know or evaluate $q$, do not communicate to vote, and produce no collective decision. The general primitive is client-side threshold verification of independently held write-once facts. In SoFi that primitive is used to form a consume-once binding over one vault-parent key. (Sections 8, 10, 62) |

The reader who holds those six answers can read the rest of this document as a proof that they are true, rather than as a search for whether they will be.

## Just enough notation to read Part I

Part I uses the following symbols before they are formally constructed in Part II. The constructions are exact there; here they are only named.

**State $s$, root $\rho$.** A device's core state is committed by $\rho_{\text{core}}$, the root of a Sparse Merkle Tree over the device's leaves; the full state commitment $\rho$ additionally binds the digests of the precommitted candidate space and the guard family. Changing any of them changes $\rho$.

**Position $u$.** In online mode, $u$ is a committed integer position that increments on every value-moving advance. More generally, $u$ is the mode's monotonic progression object. A root is always presented together with its position, $(\rho, u)$.

**Candidate, realized.** A candidate is a successor a device is *allowed* to construct from $s$; many candidates may exist at once. A successor is realized when a counterparty accepts it and it becomes part of the lineage. At most one conflicting consumption of a linear resource can occur in one valid realized lineage; showing why is the job of Part II.

**Guard $g$.** A deterministic predicate a candidate must satisfy to be accepted: a signature, a hash preimage, a threshold, an arithmetic bound. Some guards are static (their family is written $G_5$); some depend on what has already been realized ($G_7$).

**Resource descriptor $x$, key $K$.** A descriptor names the thing being consumed: a balance, a vault generation, a recovery generation. The consumption key is derived as $K = \kappa_{\text{res}}(s, x)$ from the parent state and the descriptor, and *only* from those, so two candidates that consume the same thing from the same parent derive the same $K$ whatever else they differ in.

**Consumed set $\Sigma$.** The set of keys already consumed in a lineage. It only grows. The realization rule includes $K \notin \Sigma$.

**Relationship chain.** Every pair of parties $A \leftrightarrow B$ has its own hash chain with its own leaf in each device's tree. Chains never share a parent, key, or counterparty.

**Mirror, register.** Storage nodes hold copies of committed roots and receipts (the mirror) and a set of write-once cells keyed by (genesis, device, position) (the register). Neither validates anything.

That is all Part I needs. Where a section in Part I relies on a claim that Part II proves, it says so and gives the reference.

# 4 The Six Questions Every Transition Must Answer

Given parent state $s$, candidate successor $s'$, and witness $w$, DSM asks:

1. Was $s'$ actually precommitted by $s$?
2. Has the correct deterministic guard been fulfilled?
3. Does $s'$ preserve the required state structure?
4. Are the linear resources required by the branch still unconsumed?
5. Does the transition obey committed deterministic policy?
6. Does it satisfy any mode-specific requirements?

The full acceptance predicate is

$$\begin{aligned}\text{Accept}(s, s', w) = \ & \text{CandidateOK}(s, s') \\ & \wedge \text{GuardOK}(s, s', w) \\ & \wedge \text{StructuralOK}(s, s') \\ & \wedge \text{LinearityOK}(s, s') \\ & \wedge \text{PolicyOK}(s, s') \\ & \wedge \text{ModeOK}(s, s').\end{aligned}$$

Every term is Boolean.
Therefore,

$$\text{Accept}(s, s', w) \in \{\text{True}, \text{False}\}.$$

## 5 DSM Does Not Need Global Ordering

This point deserves explicit wording.

DSM does not merely reduce global ordering.
It removes global ordering from the validity mechanism.
The system's validity rules are expressed through:

- canonical parent state;
- explicit dependencies;
- bilateral chain adjacency;
- authenticated roots;
- precommitted successor space;
- deterministic guards;
- linear-resource consumption;
- policy.

If an application requires a sequence relation, that relation can be encoded as part of state.
For example, if an application needs:

$$x_1 \prec x_2 \prec x_3,$$

that dependency can be represented explicitly in the canonical state machine.
It does not require the entire network to globally order unrelated operations alongside it.

**Bitcoin Comparison**

Bitcoin intentionally constructs a global proof-of-work ordering for monetary transactions.
DSM does not use a universal transaction-ordering system.
**Architectural consequence:**
Ordering becomes an explicit state constraint only when the application semantics require it.
It is not an infrastructure requirement imposed on all state.
Therefore DSM sacrifices neither deterministic validity nor linear-resource safety by removing global ordering from the architecture.

## 6 Concrete Double-Spend Example

Suppose Alice controls one 10-unit resource $x_A$.

Two candidates are precommitted:

$$c_B : 10 \to Bob,$$

$$c_C : 10 \to Charlie.$$

Both consume the same source generation.
Therefore:

$$K_B = \kappa_{\text{res}}(s, x_A)$$

and:

$$K_C = \kappa_{\text{res}}(s, x_A).$$

Hence:

$$K_B = K_C = K.$$

Initially:

$$K \notin \Sigma_s.$$

Suppose Bob's branch realizes.
Then:

$$\Sigma_B = \Sigma_s \cup \{K\}.$$

Charlie's branch would now require:

$$K \notin \Sigma_B.$$

But:

$$K \in \Sigma_B.$$

Therefore the second conflicting realization is invalid.

# 7 Both Candidates Can Look Valid

A careful reader will notice something about the example above.

Charlie's branch is rejected *after* Bob's branch has been folded into the lineage. Before that, against $\Sigma_s$ alone, both branches pass every check: both are precommitted, both guards can be satisfied, both derive $K$, and $K \notin \Sigma_s$.

That is not a flaw. It is the same situation as Bitcoin, and it is worth saying so plainly.

Two Bitcoin transactions spending the same UTXO are each individually valid. The scripts verify, the signatures verify, the input exists. What decides between them is not validity; it is which one the chain includes.

DSM has the same two-valid-candidates moment. What differs is what settles it.

### So who picks?

In Bitcoin, the chain picks, and the chain is a global object that many parties contend to extend.

In DSM, nobody picks, because there is nothing to pick between at the point where it matters. A resource has one acceptor, and that acceptor's accept step is atomic: the successor root and the updated $\Sigma$ are adopted together or not at all. Whichever candidate the acceptor folds in first has consumed $K$ by the time the second is evaluated. There is no window in which two candidates are both accepted and awaiting resolution.

$$\text{Bitcoin: two valid candidates} \rightarrow \text{ordering contest} \rightarrow \text{one survives}$$

$$\text{DSM: two valid candidates} \rightarrow \text{one atomic accept} \rightarrow \text{the other is no longer valid}$$

**Bitcoin Comparison**

Bitcoin readers already know that "valid" and "confirmed" are different words. DSM keeps that distinction and renames the second half: a candidate is *valid against a parent*, and it is *realized* once the one acceptor has folded it into the lineage.

**Architectural consequence:**

The step from valid to realized is a local, atomic accept at one party, not a global contest. There is no interval during which two conflicting candidates are both "accepted but unresolved."

**Tradeoff / boundary:**

The static form of the theorem, where at most one candidate even passes the predicate, is a property of selector families only. The explainer's uniqueness theorem in Section 51 is the realized-history form, which is the one that covers every family.

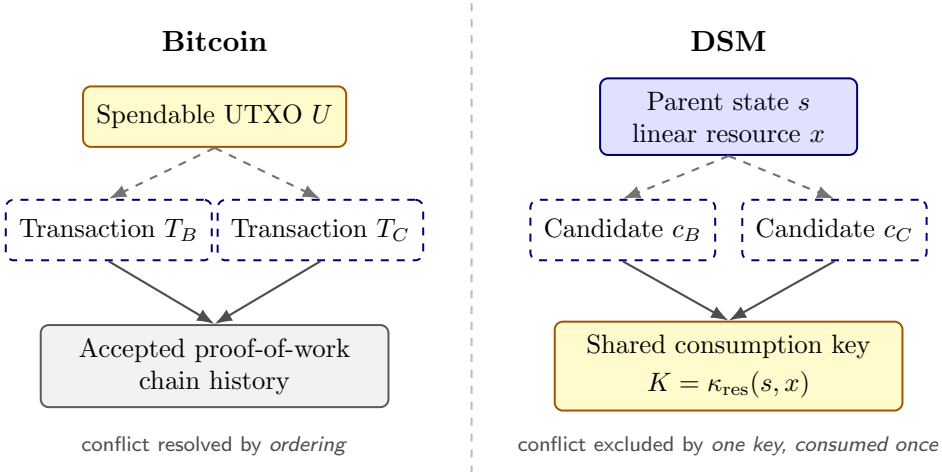



**Figure 1:** Different double-spend exclusion structures. Bitcoin resolves conflicting spends through the accepted proof-of-work chain. DSM binds conflicting candidates to the same canonical linear-resource consumption key.

**Bitcoin Comparison**

**Architectural consequence:**

DSM does not need a globally ordered chain to establish the exclusivity of the resource.
The exclusion condition is already committed into the resource state.

**Tradeoff / boundary:**

This is a state-safety theorem about one parent and one verifier. The harder case, one payer and two verifiers who never speak, is the subject of the next section.

## 8 Same Balance, Two Counterparties

The sections above showed two candidates from one parent state.

A Bitcoin reader will immediately ask a harder question.

Suppose Alice does not present two candidates to one verifier. Suppose she presents one candidate to Bob and a different candidate to Charlie, and Bob and Charlie never speak to each other.

This is the unconfirmed double spend: pay two merchants with the same coin and hope neither learns about the other before the goods change hands.

Set it up in DSM terms.

Alice holds a balance of 10. She has a relationship with Bob and a relationship with Charlie:

$$r_{AB}, \qquad r_{AC}.$$

Step one. Alice constructs and presents a transfer of 10 to Bob over $r_{AB}$. Her device state advances:

$$s_0 \to s_1, \qquad \rho_0 \to \rho_1.$$

Two words matter here and will be kept apart throughout: the transfer is *constructed* and *presented* by Alice; it is *accepted* and *realized* only when Bob's verifier says so. What Bob requires before saying so is the subject of this section.

Step two. Alice constructs a second transfer of 10, this time to Charlie over $r_{AC}$, and builds it from $s_0$ as if step one had never happened.

### Why the relationship chains alone do not catch it

The two relationship parents are different resources:

$$x_{AB} = (\text{relationship}, k_{AB}, h_{AB,n}), \qquad x_{AC} = (\text{relationship}, k_{AC}, h_{AC,m}).$$

They derive different keys. Neither chain knows about the other. Relationship linearity is not what stops this case, and neither is the non-crossability of Section 9: Alice is not replaying a Bob transition to Charlie, she is building a fresh, correctly addressed $A \leftrightarrow C$ transition from a stale root.

### Where the balance lives

Alice's balance is not stored inside either relationship chain. It is a leaf in Alice's device SMT, committed by her core root $\rho_{\text{core}}$.

Its resource descriptor is holder-scoped:

$$x_{\text{bal}} = (\text{cpta-balance}, G_T, P_T, \text{Alice}).$$

Both transfers consume the balance. Both therefore derive the same key from the same parent:

$$K_{\text{bal}} = \kappa_{\text{res}}(s_0, x_{\text{bal}}).$$

After step one:

$$K_{\text{bal}} \in \Sigma_1.$$

The transfer to Charlie is a second child of $s_0$ consuming a key that $s_1$ already records as consumed. By the uniqueness theorem it cannot join the lineage that contains $s_1$.

But Charlie has never seen $s_1$. Against $s_0$ alone, the transfer to Charlie looks perfect: the $r_{AC}$ leaf is present, the balance leaf reads 10, and $K_{\text{bal}} \notin \Sigma_0$.

So the remaining question is exactly the Bitcoin reader's question:

*How does Charlie know that $s_0$ already has a child?*

### The committed position counter

Every DSM device root carries a committed progression object $u$ (see the state object in Section 28). In online mode it is a position counter. Every root advance increments it:

$$(\rho_0,\, n) \;\rightarrow\; (\rho_1,\, n+1).$$

The counter is inside the root, so the root cannot advance without the counter and the counter cannot advance without the root.

### What Charlie holds

A relationship is not created by first contact. It is created by both sides taking the counterparty's committed device state from the storage-node mirror and verifying it. From that moment Charlie holds Alice's committed position:

$$(\rho_A,\, u_A).$$

Charlie does not trust the mirror. He verifies the signed root chain that produced $(\rho_A, u_A)$. The mirror supplies bytes; Charlie supplies the verdict.

### The check at acceptance

When Alice presents a transfer to Charlie, the presented parent carries a position. Before evaluating anything else, Charlie asks one question: what root does Alice's device hold at the *next* position?

He can ask it because every value-moving advance of a device root has to be registered. The storage set keeps an *economic-root register*: a write-once cell for each

$$(G,\ \text{DevID},\ \text{position}),$$

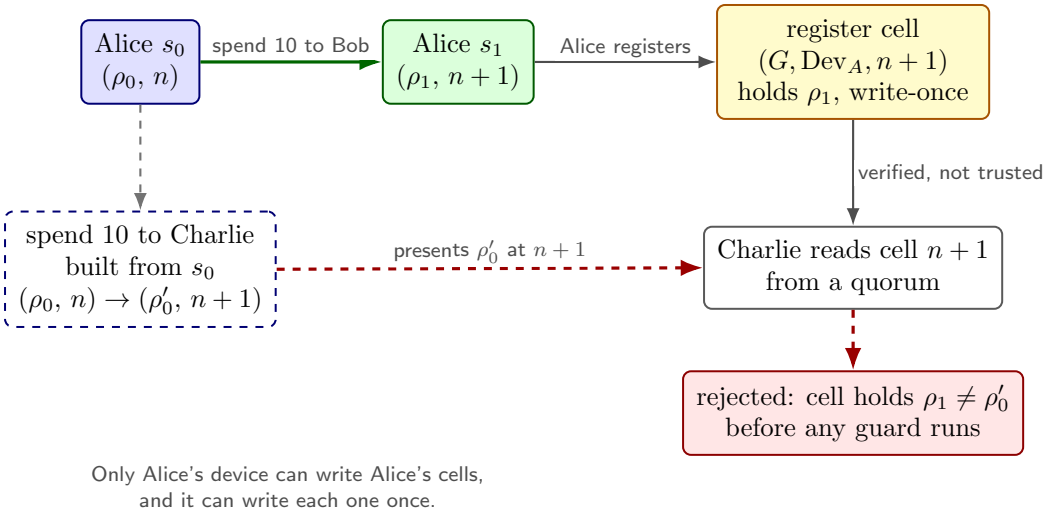
where $G$ is the genesis and DevID the device. The cell's key is derived from those three values, so anyone can compute it. The cell can be written only by the device it names: the write is device-authenticated and the node checks that the signing key and device inside the claim are the caller's before storing anything. First bytes for the cell win, in one atomic write; identical bytes re-acknowledge; different bytes are refused and the held digest is returned. There is no update and no delete (Section 10).

So the register holds, for Alice's device, at most one root per position, and only Alice could have put it there. Charlie reads the cell for $n+1$ from a quorum of the set. That quorum is evaluated by Charlie: he queries the owner-committed member set, authenticates each response, and locally tests whether the committed threshold is satisfied; the members do not vote with one another and do not know what he is counting. There are exactly three outcomes:

- The cell holds a root that is not the one Alice is presenting. Alice's device has already moved past $n$; the presented parent is superseded. Reject.
- The cell is empty. Charlie will accept only once Alice has registered the Charlie-transfer root at $n+1$ and the quorum confirms it. If Alice will not register, Charlie does not accept.
- The cell holds exactly the presented root. Accept.

In the story above, Alice presented a transfer to Bob at $n+1$. Bob, being an honest acceptor, applied the same rule: he treated the transfer as realized only once the cell at $n+1$ held $\rho_1$. So either Alice registered $\rho_1$, in which case Bob accepted and Charlie's read returns $\rho_1 \neq \rho_0'$ and he rejects; or she never registered it, in which case Bob never accepted, holds candidate and receipt material rather than a realized transfer, and Charlie's acceptance is what forces $\rho_0'$ into the cell. She cannot have both.

Bob's own copy of the receipt is still on the mirror, and it is still useful: it is a second witness to the same fact. But it is not what the argument rests on.




## Two wallets, one owner

The obvious way to try to defeat a check that relies on a counterparty is to be the counterparty. Alice runs two wallets, $W_1$ and $W_2$. She pays $W_2$ from $W_1$ at position $n+1$ and tells nobody: no push to any mirror, no receipt anywhere. Then $W_1$ pays Bob from the same balance, also at $n+1$.

If the position check were "ask the previous counterparty," this would work, because the previous counterparty is Alice. Bob would see $W_1$ at $n$, accept, and Alice would hold the value in $W_2$ while Bob believed he held it too.

Against the register it fails in the same way as before, because the cell for $(G, \mathrm{Dev}_{W_1}, n+1)$ belongs to $W_1$, not to whoever $W_1$ paid. Bob's acceptance requires that cell to hold the Bob-transfer root. Either it already holds the $W_2$-transfer root, and Bob rejects; or it is empty, Bob requires $W_1$ to fill it with $\rho_{\mathrm{Bob}}$, and from then on $W_2$'s incoming receipt names a root at $n+1$ that the register contradicts. The next time $W_2$ tries to spend that value to an honest acceptor who walks provenance back one hop, the equivocation is a comparison of two leaves.

The register does not care who the counterparty was. It cares that a device can name one root per position, and that the device is the only party who can do the naming.

## Two roots at one position

Suppose instead that Alice tries to present a root at position $n+1$ that differs from $\rho_1$. Then two distinct roots claim the same position under the same device. That is a fork, and it is decidable on sight: any party holding both compares two leaves and is done. The register exists so that the second leaf can never be *acknowledged* anywhere honest; the fork is refused at the cell, not discovered afterward.

## What this does and does not depend on

It depends on the register's members not acknowledging two values for one cell, and on that holding across restarts and restores. That is a durable non-equivocation assumption about storage, stated as such in Section 10 and listed in Section 78. It does not depend on the storage node validating anything, ranking anything, or signing anything; a node will store a nonsense root perfectly consistently, and whether the root is the result of a valid transition is Charlie's question. Storage remains outside the authority path.

If the quorum is unreachable, Charlie is not a weaker online verifier. He is, by definition, an offline receiver. That case has its own machinery: the committed anchor counter, commit-time identity witnesses, and the measurement gate of Sections 64–69.

**Key Idea**

Online, a device can name one root per position, in a cell only it can write and no one can rewrite. The double spend is not resolved later by a chain. It is refused at acceptance by reading one cell.

**Bitcoin Comparison**

A Bitcoin merchant who accepts an unconfirmed transaction is exposed until the transaction is mined, because a conflicting transaction may be mined first. The merchant cannot know, at the counter, whether the coin has already been spent elsewhere. Confirmation depth is the remedy, and it is a waiting period.

DSM requires no confirmation-depth waiting period for this case; acceptance still requires the configured storage threshold to respond. The payer's committed position is part of the payer's state, the payer's root at each position is registered write-once by the payer's own device, and the receiver reads the next cell at the moment of acceptance.

**Architectural consequence:**

The information a Bitcoin merchant is waiting for is exactly the information a DSM receiver already has: whether the payer's state has moved past the parent being presented. DSM commits that fact as a counter inside the payer's root and binds each position to one root in a cell the payer cannot overwrite and nobody else can write.

**Tradeoff / boundary:**

This is an online property. It relies on reaching a quorum of the storage set and on the durable non-equivocation of the register, which is an availability and storage-integrity question, not a validity one. A payer who withholds does not gain a second spend; the receiver simply cannot accept, which is a liveness failure. When the quorum is unavailable at the moment of exchange, the receiver is in offline mode and the offline machinery applies.

## 9 A Transition Cannot Cross Relationships

There is a second exclusion in DSM that is easy to miss because it never needs a theorem. It is built into the addressing.

Every relationship is its own hash chain. $A \leftrightarrow B$ and $A \leftrightarrow C$ do not share a parent, a leaf, a key, or a counterparty. A transition that happened between Alice and Bob is not an object that can be picked up and replayed between Alice and Charlie.

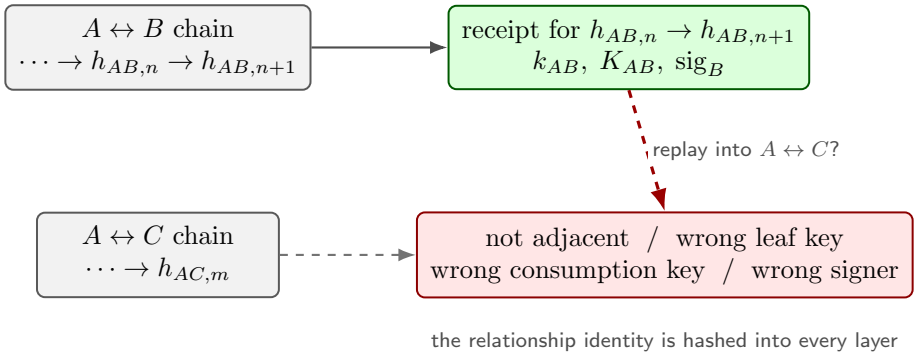
Look at what a transition on $A \leftrightarrow B$ actually carries:

- its parent $h_{AB,n}$, which is a node of the $A \leftrightarrow B$ chain and of no other chain;
- its SMT leaf key $k_{A \leftrightarrow B}$, which embeds both device identifiers;
- its resource descriptor $x_{AB} = (\text{relationship}, k_{AB}, h_{AB,n})$, and therefore its consumption key $K_{AB} = \kappa_{\text{res}}(s, x_{AB})$;
- Bob's signature, over canonical bytes that name Bob's device.

Now try to present that transition to Charlie as a step on $A \leftrightarrow C$. It fails four separate times.

| | |
|---|---|
| parent $h_{AB,n}$ | is not adjacent to $h_{AC,m}$ |
| leaf key $k_{AB}$ | $\neq k_{AC}$ |
| consumption key $K_{AB}$ | $\neq K_{AC}$ |
| signature by $B$ | is not a signature by $C$ |

Any one of these is fatal. There is no reframing, re-keying, or re-signing that turns a $A \leftrightarrow B$ object into a valid $A \leftrightarrow C$ object, because the relationship identity is hashed into every layer.




This gives DSM three orthogonal exclusions, and it helps to keep them apart:

1. **Within a relationship:** the parent is a linear resource. Two successors of $h_{AB,n}$ cannot both be realized (Section 51).
2. **Across relationships:** nothing transfers. A step on $A \leftrightarrow B$ is not a candidate on $A \leftrightarrow C$ at all. This section.
3. **Across relationships, same balance:** the balance is a holder-scoped resource in Alice's device root, and the write-once root register is how the second counterparty learns the root has moved (Section 8).

The first is a theorem. The second is addressing. The third is the register.

**Bitcoin Comparison**

In Bitcoin a signed transaction is a free-standing object. Anyone who holds the bytes can relay them anywhere, and the network will accept them from any source, because the transaction names a UTXO rather than a counterparty.

A DSM transition is not free-standing. It names its relationship in its parent, its leaf key, its consumption key, and its co-signature, and it is valid only against that relationship's frontier.

**Architectural consequence:**

There is no "relay it somewhere else" attack surface. A transition that exists for one relationship is meaningless bytes to every other relationship, and the verifier does not need a rule to reject it; it simply fails to verify.

**Tradeoff / boundary:**

This is scoping, not secrecy. Charlie can be shown the $A \leftrightarrow B$ receipt as evidence of something (for example, as a stitched proof in a multi-party workflow). What he cannot do is accept it as a step on his own chain.

# 10 Storage Nodes

This document will say "storage is not authority" many times. A Bitcoin reader will want to know what a storage node actually *is*, because in their world the thing that stores the chain is also the thing that validates it, and separating the two sounds like a trick.

The description below is of the shipping node: a Rust service in front of a PostgreSQL database, speaking protobuf over TLS. The production set is six nodes across three cloud regions at roughly ninety dollars a month. That number is worth holding in mind against what Bitcoin's storage-and-ordering layer costs.

### The rule the code enforces

A storage node never signs, never validates a protocol rule, never evaluates a guard, never computes a balance, and never decides whether a transition is valid. Storage nodes do not evaluate DSM semantic state: economic and protocol payloads are opaque to them, apart from the narrow content-addressing, admission, attribution, and write-once rules described below. The node does not vary acceptance by what a payload would parse as; the previous store did, and it was replaced because a generic substrate cannot. Nothing in any protocol-relevant path reads a clock; ordering inside the node uses logical ticks.

Inter-node gossip exists, and it is state synchronisation only: no leader election, no Raft, no Paxos between nodes, no vote.

### What it stores

**Immutable objects.** A payload $P$ in namespace $N$ is stored at

$$\text{addr} = H(\texttt{DSM/storage-object} \parallel N \parallel H(N \| P)),$$

computed by the node from the input. A caller-supplied address is checked, never used as the key. There is no update path and no overwrite path in the code, not an update path that refuses. Replaying identical bytes re-acknowledges; different bytes at the same address are reported as corruption. On read the node recomputes the address before serving. The client re-hashes anyway, because a check performed by the party being verified is not a check.

**Per-device tip mirror.** A public head and encrypted per-relationship leaves, keyed by device and relationship. This is where a counterparty's copy of a receipt lives: a second witness to a device's frontier, alongside the register below.

**Inbox spool.** Unilateral delivery for the offline counterparty. Envelopes are strictly versioned, ordered by insertion, and acknowledged per routing key. Admission is a set of cheap deterministic gates: canonical encoding, device authentication, a replay-protected message id, and a recipient key. The node never opens the envelope.

**Identity and recovery.** Genesis anchoring, device-tree indexing, and recovery capsules, all stored as bytes under derived keys.

**The node's own commitments.** Each cycle a node emits an unsigned ByteCommit: a Sparse Merkle root over what it holds, linked to the previous cycle's digest, stored as an ordinary object under a deterministic address and mirrored by peers. Capacity is regulated against it, and an operator exits by proving two consecutive empty cycles. Verifiers check the chain link and the root themselves; the node's word is not part of it.

### The spend-gate

Writes are not free. A device may write only once it has paid a flat rate to $K = 3$ distinct operators; the node stores the payment receipts and counts distinct operators, and once the threshold is met the device is enabled permanently. This is the join event that drives DJTE (Section 61) and it is the floor under the spam argument: an identity costs three receipts before it can store a byte.

### The one place the node is not dumb

The honest version of this section has an exception, and the code says so in its own comments: for the write-once registers, "the dumb indexer description of a storage node is not accurate."

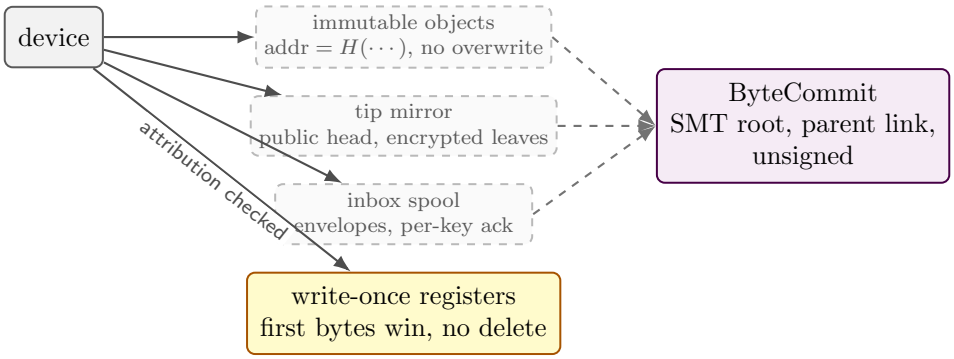
Three registers share one discipline. A settlement slot keyed by $(\text{vault\_id}, \text{parent\_sequence})$ is the lock of Section 62. An economic-root cell keyed by $(G, \text{DevID}, \text{position})$ is where a device names its root at a position and can never name a second; it is the mechanism behind Section 8. A faucet-ticket register is the third. For all of them:

- first bytes for the key win, in one atomic write over the unique key, never check-then-insert;
- identical bytes re-acknowledge, different bytes are refused with the held digest;
- there is no update path and no delete path;
- before storage the node checks *attribution* only: the claimant key and device inside the signed body must be the authenticated caller, and the storage-set id must be this node's own set;
- every response echoes the node's configured member id, so a client counting toward a quorum counts a response only when the answering node is the member its catalogue named.

Registered is not validated. A malicious trader can register an arbitrary root perfectly consistently and the node will store it; whether that root is the result of a valid transition is the verifier's question and nothing the node says bears on it. What the node contributes is *non-equivocation*: it will never hold two values for one key, and that fact must survive restart and storage lifecycle. Restoring the register from a snapshot that predates a held claim is a safety violation, not an availability event.

That is a real assumption and it should be read as one, and it is narrower than a crash-versus-Byzantine label. Immutable-object misresponses are detectable by hash and affect availability only. The safety-critical assumption is durable per-key non-equivocation across restart and restore.

Per-node non-equivocation becomes at-most-one-winner through intersection. The configured $(S, q)$ must guarantee that any two response sets satisfying the acceptance threshold intersect in enough non-equivocating members to prevent two incompatible threshold certificates: with $|S| = n$, any two sets of size $q$ share at least $2q - n$ members, so $q > n/2$ is the uniqueness condition. The shipping profile uses $n = 5$, $q = 4$: any two certificates share at least three members, which leaves the intersection intact even if one previously acknowledging member later becomes unavailable.




**Bitcoin Comparison**

A Bitcoin full node stores the chain and validates every block and transaction in it; storage and validation are one program. It cannot be trusted for anything, and it does not need to be, because every other node re-validates.

A DSM storage node stores bytes and does not validate anything; validation happens on the two devices at the ends of a relationship. It cannot be trusted for content either, and it does not need to be, because the recipient re-hashes and re-verifies. What it is trusted for is narrower: to keep bytes available, and, in the write-once registers, not to say two things about one key.

**Architectural consequence:**

The entire storage tier for the network is six small instances, none of which runs a consensus protocol, holds a DSM validation or consensus authority key, or knows what a transaction is. There is no miner or validator role to capture because there is no role that semantically validates or globally orders transitions.

**Tradeoff / boundary:**

The write-once registers carry a durable per-key non-equivocation assumption: a member never acknowledges two values for one key, and that survives restarts and restores. Beta accepts a further griefing exposure, that any authenticated device can claim a settlement slot and vanish, locking that vault parent. The code marks that acceptable for a controlled fleet and a launch blocker for a public market.

# 11 Online DSM

Ordinary online DSM does not require hardware to determine transition uniqueness.

A verifier may receive:

- canonical state bytes;
- relationship proof;
- signatures;
- Merkle proofs;
- precommitment information;
- guard witness;
- resource-consumption proof;

- policy information.

Then it performs local verification.

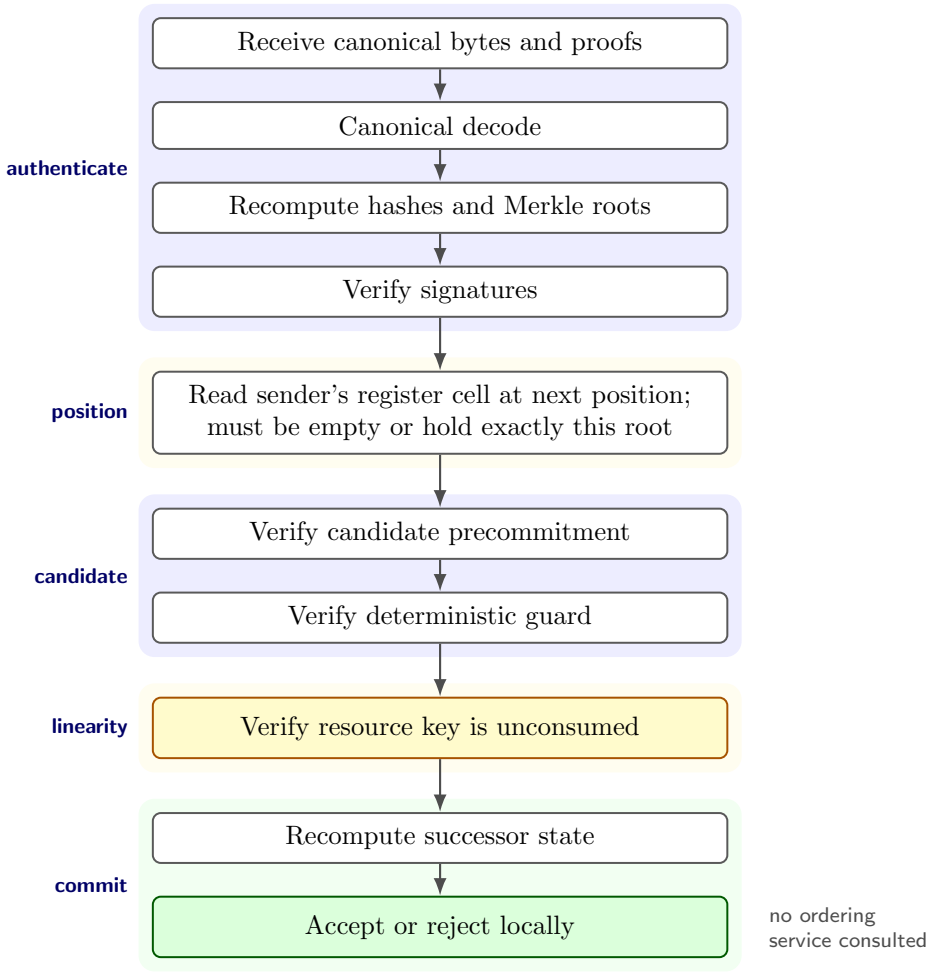



**Figure 2:** Online Verification Path

## 12 DSM Is Not a Payment Channel

A Bitcoin reader who has followed this far will have a name ready: Lightning.

Two parties, a private chain of signed states, no on-chain transaction for the ordinary case. The resemblance is real and it stops early.

A payment channel is bilateral *off* chain, but its safety is rooted *on* chain. The channel exists because a funding transaction was mined. It closes by broadcasting the latest signed state. If a counterparty broadcasts a stale state, the honest party must notice within a timeout and publish a penalty transaction, which is why watchtowers exist. Every dispute reduces to one question: which state does the global ledger accept?

DSM has no ledger to reduce to.

There is no funding transaction, because there is no chain to fund on. There is no channel close, because the bilateral chain is not a temporary detour from a canonical record; it *is* the canonical record for that relationship. And there is no stale-state attack, because a stale state is not a thing that can be broadcast anywhere. The only party who accepts a step on $A \leftrightarrow B$ is the counterparty, and the

counterparty holds the frontier. A stale parent presented to the party who already advanced past it is simply not adjacent.

| | Lightning channel | DSM bilateral chain |
|---|---|---|
| Opened by | on-chain funding transaction | mutual pre-add from committed state |
| Safety rooted in | the global ledger | the consumed resource itself |
| Stale state | broadcastable, penalised on chain | not adjacent, never accepted |
| Dispute window | timeout, must be watched | none |
| Watchtower | needed if offline | no role |
| Hostile counterparty | can force a close | can only refuse to advance |
| Court of last resort | the chain | none on the common path |

The last two rows are the honest trade.

In Lightning a hostile counterparty can hurt you, but the chain will eventually settle what you are owed, provided the honest party or its watchtower acts within the relevant protocol window. In DSM a hostile counterparty cannot take anything from you and cannot forge a state you did not sign, but they can decline to continue, and no third party will continue on their behalf. DSM classifies that as a liveness failure and keeps it out of the safety theorem entirely.

**Bitcoin Comparison**

Lightning moves the common case off chain and keeps the chain as the arbiter for the uncommon case.

DSM removes the arbiter. Conflicting successors of one consumed resource are excluded by resource linearity at the one acceptor who holds the frontier, not selected among by whichever history a network eventually agrees on.

**Architectural consequence:**

No timeouts, no penalty transactions, no watchtowers, and no requirement that the counterparty be live within a deadline. Safety does not depend on anyone broadcasting anything in time.

**Tradeoff / boundary:**

Lightning buys something DSM does not offer: a stranger who has never held your frontier can still be made whole by the chain. In DSM, the party who holds your frontier is the party who can accept your next step. That is the design, not a gap, and the position check of Section 8 is how the frontier reaches a counterparty who was not present for your last step.

# Part II
# The Construction

The primitives, built in order, ending in the uniqueness theorem that Part I relied on.

## 13 Determinism

The word *deterministic* is not decorative. It is the foundation of the architecture.

**Mathematical primer** — optional for technical readers

Consider the elementary function $f(x) = 2x + 3$. For $x = 5$ the answer is $f(5) = 13$. Alice obtains 13, Bob obtains 13, Charlie obtains 13. The answer does not depend on which person evaluates the

function. DSM imposes exactly that property on state verification.

Let

$$V(s, c, w)$$

be the verifier.
Then for identical canonical inputs,

$$V_A(s, c, w) = V_B(s, c, w).$$

No verifier gets to interpret the bytes according to preference.
No storage node gets to declare an invalid object valid.
No ordering service gets to decide which mathematical rule applies.
The rule is already fixed.

**Key Idea**

Determinism allows authority to reside in the transition predicate rather than in the identity of whoever announces the result.
A conforming verifier should be able to reproduce the decision from the canonical inputs alone.

### 13.1 Determinism Is Broader Than Arithmetic

DSM requires deterministic:

- serialization;
- hashing;
- relationship identifiers;
- candidate digests;
- guard evaluation;
- resource descriptors;
- resource-consumption keys;
- SMT updates;
- policy evaluation;
- successor construction;
- root recomputation.

Thus determinism is distributed throughout the architecture.

**Bitcoin Comparison**

Bitcoin is deterministic at the level of consensus validation as well. Conforming nodes should agree whether the same block or transaction is valid.
The architectural difference is that Bitcoin still uses a shared proof-of-work chain to establish the

accepted global transaction history from which the current UTXO state follows.
DSM removes global transaction ordering from the validity model.
Given a committed parent and a proposed transition, DSM attempts to make the answer derive entirely from canonical state, precommitments, guards, consumption state, policy, and proofs.
**Architectural consequence:**
DSM does not require another system to decide which state-transition rule wins. The same canonical state produces the same validity answer locally.

# 14 Canonical Encoding

Deterministic logic is useless if machines encode the same object differently.

Suppose two programs represent the same balances as:

$$\texttt{Alice=10,Bob=20}$$

and

$$\texttt{Bob=20,Alice=10}.$$

A human may say they mean the same thing.
Cryptographically, they are different byte strings.
Normally,

$$H(\texttt{Alice=10,Bob=20}) \neq H(\texttt{Bob=20,Alice=10}).$$

DSM therefore requires canonical encoding.
For logical object $x$,

$$\mathrm{enc}(x)$$

means its unique protocol-defined byte representation.
Then

$$d_x = H(\mathrm{enc}(x)).$$

Every correct implementation must generate exactly the same bytes.

## 14.1 Canonical Equality

DSM attempts to convert logical equality into byte equality.

If

$$x = y$$

as protocol objects, then a canonical encoder must satisfy

$$\mathrm{enc}(x) = \mathrm{enc}(y).$$

If two implementations produce different encodings for the same logical state, they have not implemented the same protocol object.

### 14.2 Why This Matters Everywhere

Canonical encoding affects:

- signatures;
- hashes;
- state roots;
- relationship heads;
- candidate commitments;
- guards;
- SMT leaves;
- resource keys;
- policy;
- proofs.

A serialization disagreement becomes a state disagreement.

**Bitcoin Comparison**

Bitcoin also uses consensus-critical canonical structures and exact byte interpretation.
The difference is how widely DSM relies on committed canonical state.
In DSM, not only transactions but also current relationship heads, precommitted successor spaces, guards, resource descriptors, consumed-state sets, and policies participate in the deterministic commitment structure.
**Architectural consequence:**
A verifier does not need to ask a database operator what an object was supposed to mean. It reproduces the canonical object byte-for-byte.

## 15 Domain Separation

A cryptographic hash should also know what kind of object it is hashing.
For example,

$$H(\texttt{DSM/consume resource/v1} \parallel x)$$

should not be confused with

$$H(\texttt{DSM/smt-key/v1} \parallel x).$$

The prefixes are called domain separators.
They make semantically different hash operations live in cryptographically different namespaces.
Thus the same raw field cannot accidentally be interpreted as two different protocol object types.

## 16 Cryptographic Hashes

A hash function maps arbitrary input to a fixed-size digest.

$$H(x) = h.$$

For a 256-bit digest,

$$h \in \{0,1\}^{256}.$$

A useful intuitive analogy is a cryptographic fingerprint.
Changing the input changes the fingerprint.
If

$$x \neq y,$$

then, under the collision-resistance assumption,

$$H(x) \neq H(y)$$

except with negligible probability.
Hashes allow DSM to commit to large state objects using compact fixed-size identifiers.

## 17 Forward-Only Hash Chaining

Suppose states evolve as

$$S_0, S_1, S_2, S_3.$$

Define

$$h_n = H(\text{enc}(S_n)).$$

A successor binds its predecessor:

$$S_{n+1} \supset h_n.$$

Thus:

$$S_0 \to S_1 \to S_2 \to S_3.$$

If somebody changes $S_1$, then

$$h_1 \to h_1'.$$

But $S_2$ was committed to the original $h_1$.
The chain no longer verifies.

## 18 Bilateral Relationships

A DSM bilateral relationship is an independently evolving state chain between two identified participants or devices.

Let the devices be

$$A, B, C, D.$$

The relationship between $A$ and $B$ is

$$r_{AB}.$$

The relationship between $A$ and $C$ is

$$r_{AC}.$$

These are not the same state resource:

$$r_{AB} \neq r_{AC}.$$

Each relationship has its own forward-only chain.
For $A \leftrightarrow B$:

$$h_{AB,0} \rightarrow h_{AB,1} \rightarrow h_{AB,2} \rightarrow h_{AB,3}.$$

For $A \leftrightarrow C$:

$$h_{AC,0} \rightarrow h_{AC,1}.$$

For $A \leftrightarrow D$:

$$h_{AD,0} \rightarrow h_{AD,1} \rightarrow h_{AD,2}.$$

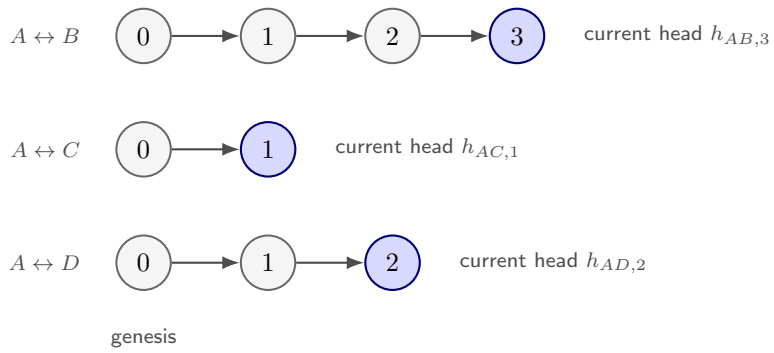



**Figure 3:** Independent Bilateral Chains

The important point is that these are distinct chains.
An update to one does not imply an update to the others.

## 19 Relationship State as a Vector

Alice's current relationship state can be written as a vector:

$$R_A = (h_{AB,3}, h_{AC,1}, h_{AD,2}).$$

Suppose Alice and Bob advance.
Then

$$R'_A = (h_{AB,4}, h_{AC,1}, h_{AD,2}).$$

Only one coordinate changed.
This is mathematically similar to changing one component of a vector while leaving the others fixed.

## 20 Relationship Projections

Let the total state space be

$$S.$$

For relationship $r$, define projection

$$\pi_r : S \to S_r.$$

The projection extracts the state belonging to $r$.
If

$$s = (x, y, z),$$

then

$$\pi_1(s) = x, \qquad \pi_2(s) = y, \qquad \pi_3(s) = z.$$

Suppose transition $f_{AB}$ touches only $A \leftrightarrow B$.
Then:

$$\pi_{AB}(s') = f_{AB}(\pi_{AB}(s)).$$

For an unrelated relationship:

$$\pi_{AC}(s') = \pi_{AC}(s).$$

This means that any valid DSM state transition involving a relationship must agree exactly with the deterministic transition defined for that relationship, while leaving unrelated relationship projections unchanged.

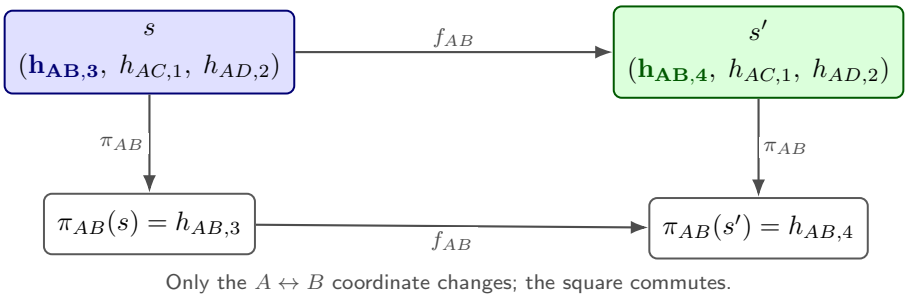



**Figure 4:** Projection-Preserving Update

## 21 Why Bilateral State Matters

Bilateral relationships allow local state to advance independently.

DSM does not need a global statement such as

$$h_{AB,4} < h_{CD,7}$$

unless some explicit state rule actually creates a dependency between those operations.

The validity of $A \leftrightarrow B$ is determined from the state relevant to $A \leftrightarrow B$, together with any named shared resources or policy.

**Bitcoin Comparison**

Bitcoin does not maintain a separate canonical state chain for every pair of participants.
Bitcoin transactions consume globally recognizable UTXOs and eventually participate in the same proof-of-work blockchain history.
DSM uses explicit bilateral chains and authenticated projections.
**Architectural consequence:**
A relationship can progress without requiring unrelated relationships to be placed before or after it in one global history.
DSM replaces global ordering entirely as a validity mechanism rather than merely trying to reduce how often it is used.
If an application needs an ordering relation, that relation can itself be represented deterministically in committed state.

## 22 Why a Device Needs a Compact Commitment

A device may have a large number of relationships:

$$r_1, r_2, \ldots, r_n.$$

One could store a giant ordered list:

$$(h_1, h_2, \ldots, h_n).$$

But proving one relationship would be inefficient if the verifier had to receive the entire list.
DSM therefore uses authenticated tree commitments.
The important structure is the Sparse Merkle Tree.

## 23 Ordinary Merkle Trees

Suppose there are four leaves:

$$L_1, L_2, L_3, L_4.$$

Hash each leaf:

$$a = H(L_1), \quad b = H(L_2), \quad c = H(L_3), \quad d = H(L_4).$$

Then:

$$u = H(a \parallel b),$$

$$v = H(c \parallel d),$$

and finally:

$$\rho = H(u \parallel v).$$

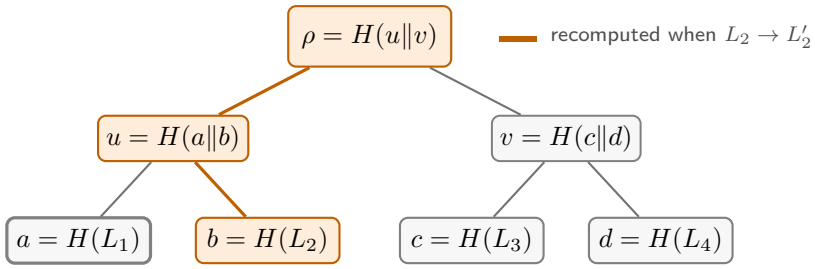



**Figure 5:** Ordinary Merkle Tree

Change one leaf:

$$L_2 \to L_2'.$$

Then:

$$H(L_2) \to H(L_2'),$$

which changes $u$, which changes $\rho$.
Therefore one small change creates a new root commitment.

## 24 Sparse Merkle Trees

A Sparse Merkle Tree uses a fixed logical key space.
For a 256-bit key:

$$k \in \{0,1\}^{256}.$$

The logical tree has

$$2^{256}$$

possible leaf positions.

That does *not* mean the implementation stores $2^{256}$ leaves.

Almost all positions are empty.

Empty subtrees have deterministic default hashes.

Only populated paths and necessary nodes need to be represented.

Thus a sparse tree can logically address an enormous namespace while physically storing a tiny subset of it.

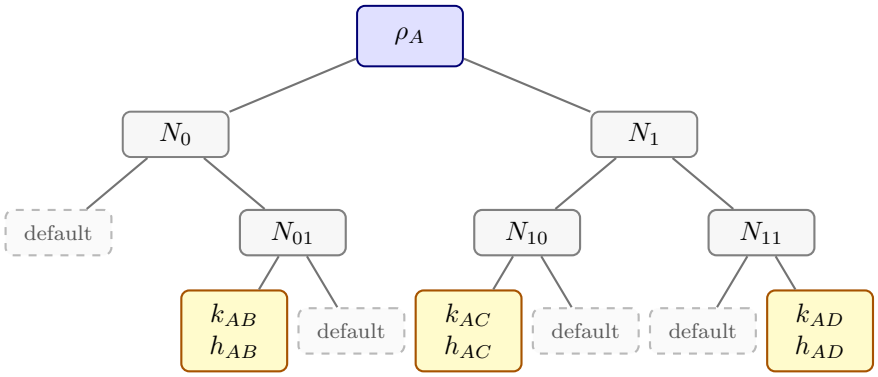



**Figure 6:** Sparse Merkle Tree Concept

This miniature diagram shows only a few levels.

A real 256-bit SMT has a logical depth of 256.

## 25 Relationship Keys

DSM can derive a bilateral relationship key deterministically.

For devices $A$ and $B$:

$$\boxed{k_{A\leftrightarrow B} = H(\texttt{DSM/smt-key/v1} \,\|\, \min(DevID_A, DevID_B) \,\|\, \max(DevID_A, DevID_B)).}$$

Because the device IDs are canonically sorted:

$$k_{A\leftrightarrow B} = k_{B\leftrightarrow A}.$$

The relationship leaf stores its current head:

$$\mathrm{SMT}_A[k_{AB}] = h_{AB,n}.$$

Likewise:

$$\mathrm{SMT}_A[k_{AC}] = h_{AC,m},$$

$$\mathrm{SMT}_A[k_{AD}] = h_{AD,q}.$$

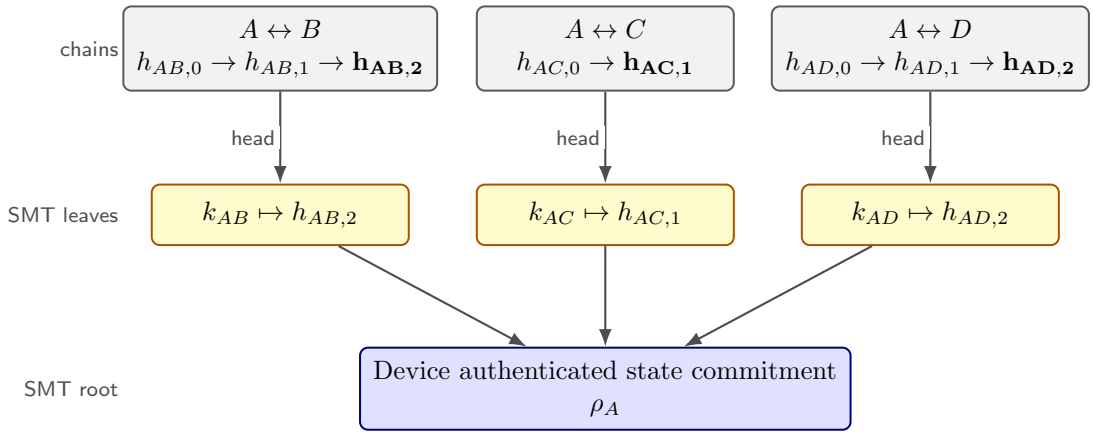



**Figure 7:** Bilateral Chains Into One Device SMT

The relationship chains remain independent.
The SMT commits their current heads.

## 26 Updating One SMT Leaf

Suppose Alice and Bob advance:

$$h_{AB,n} \to h_{AB,n+1}.$$

Before:

$$\mathrm{SMT}_A[k_{AB}] = h_{AB,n}.$$

After:

$$\mathrm{SMT}'_A[k_{AB}] = h_{AB,n+1}.$$

All unrelated leaves remain unchanged.
The root changes:

$$\rho_A \to \rho'_A.$$

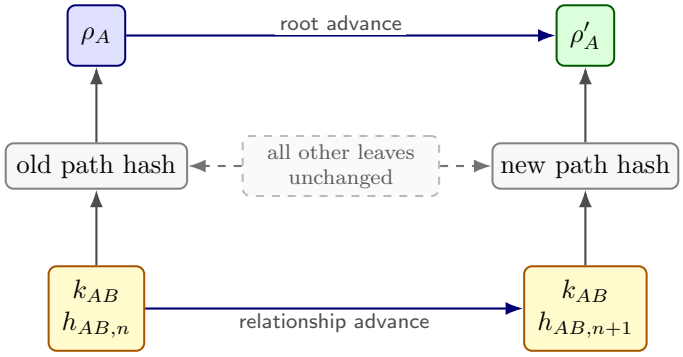



**Figure 8:** One Leaf Changes the Root

# 27 Merkle Proofs

A Merkle proof proves that one leaf belongs to one committed root without sending every leaf.

Suppose the verifier wants to prove:

$$k_{AB} \mapsto h_{AB,n}.$$

The proof supplies the sibling hashes needed to recompute the path.
At each level, the verifier performs:

$$p_{i+1} = H(p_i \parallel sibling_i)$$

or

$$p_{i+1} = H(sibling_i \parallel p_i)$$

depending on whether the path goes left or right.
After the final level, the verifier obtains:

$$p_{256}.$$

The proof is valid if

$$p_{256} = \rho_A.$$

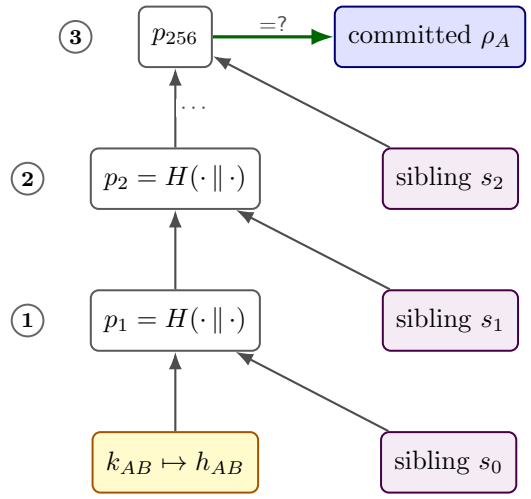



**Figure 9:** Merkle Inclusion Proof

In a 256-bit SMT, the logical proof path has depth 256, although default subtrees may be represented compactly by implementation-specific proof compression.

**Bitcoin Comparison**

Bitcoin also uses Merkle trees.
A Bitcoin block's transaction Merkle root commits the transactions contained in that block.
DSM uses authenticated tree structures as part of current state itself.
For example:

$$k_{AB} \mapsto h_{AB}$$

represents the presently committed bilateral relationship head.

**Architectural consequence:**

The tree is not merely a proof that an event occurred in an historical block. It participates directly in authenticated state evolution.

**Tradeoff / boundary:**

The important distinction is not "Bitcoin has no Merkle trees." Both use Merkle commitments. The difference is what the tree commits and how the root participates in the state model.

## 28 The DSM State Object

A complete abstract DSM state can be represented as:

$$\boxed{s = (R, u, P, \Gamma, \Sigma, \Pi, \Omega, \rho).}$$

Each component has a distinct function.

### $R$: Relationship Map

$$R : \mathcal{R} \to \mathcal{H}.$$

For relationship identifier $r$,

$$R(r) = h_r.$$

It records the current authenticated relationship head.

### $u$: Progression Object

The object $u$ represents monotonic progression.

Depending on the protocol mode, it may be:

- a sequence digest;
- a local progression coordinate;
- a receipt frontier;
- a committed offline anchor counter.

### $P$: Candidate Precommitment Space

$$P = \{c_1, c_2, \ldots, c_n\}.$$

It commits the currently permitted future candidates.

### $\Gamma$: Guard Family

$$\Gamma = \{(b_i, g_i, K_i)\}_{i=1}^{n}.$$

It associates candidate branches with deterministic fulfillment rules and resource sets.

### $\Sigma$: Consumed Resource Set

$$\Sigma$$

contains resource-consumption keys already consumed in the realized lineage.

### $\Pi$: Policy

$$\Pi$$

contains deterministic policy and authority data.

### $\Omega$: Mode Evidence

$$\Omega$$

contains mode-specific evidence. In an ordinary online state it may be empty.

### $\rho$: State Commitment

$$\rho$$

is the canonical state root.

## 29 The Layered Root

There is a subtle dependency problem.

Candidates need to bind the parent state.

But if the parent root includes candidate commitments, then naively defining the candidates in terms of the full root could produce a cycle.

DSM avoids this using a core root.

First:

$$\boxed{\rho_{\text{core}} = \text{SMT}(R, u, \Sigma, \Pi, \Omega).}$$

Candidates and guards may safely bind:

$$\rho_{\text{core}}.$$

Then the full root is:

$$\boxed{\rho = H(\rho_{\text{core}} \parallel \text{digest}(P) \parallel \text{digest}(\Gamma)).}$$

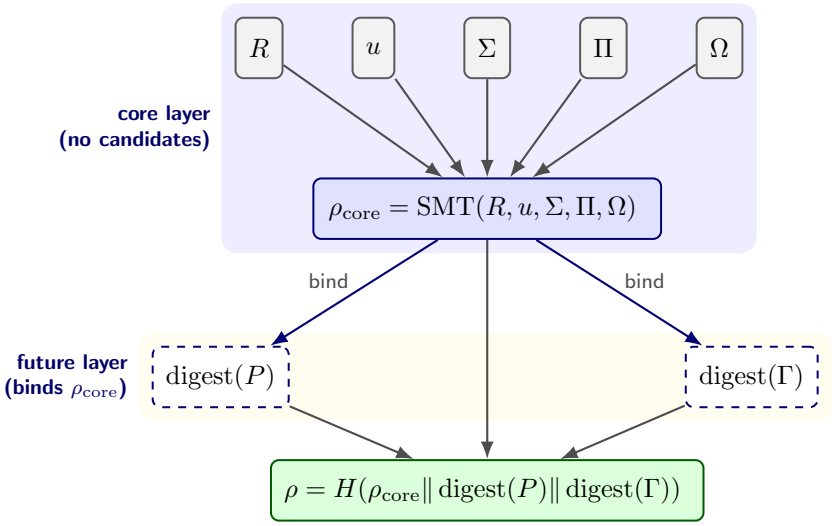



**Figure 10:** Layered Canonical Commitment

The dependency graph is acyclic.

$$(R, u, \Sigma, \Pi, \Omega) \to \rho_{\text{core}} \to (P, \Gamma) \to \rho.$$

## 30 Canonical State Chaining

The complete state itself advances forward.

$$s_0 \to s_1 \to s_2 \to s_3.$$

At the root level:

$$\rho_0 \to \rho_1 \to \rho_2 \to \rho_3.$$

The arrow means derivability.
It does not mean:

"A global network placed these states in chronological order."

It means:

"The successor is a valid deterministic transformation of the committed parent."

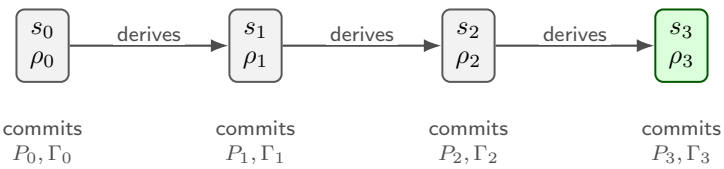



**Figure 11:** Canonical DSM state derivation. Each successor is a valid deterministic derivation of its committed parent.

Every state commits both current facts and its permitted transition structure.

## 31 Bitcoin Chain Versus DSM State Chain

**Bitcoin Comparison**

Bitcoin chains blocks:

$$B_0 \to B_1 \to B_2 \to B_3.$$

Each block contributes to one shared proof-of-work history.
DSM chains authenticated state derivations:

$$s_0 \to s_1 \to s_2 \to s_3.$$

The DSM state already commits current relationship heads, consumed resources, policy, progression, mode evidence, precommitments, and guards.
**Architectural consequence:**
DSM does not need a global event history to define whether an unrelated local transition is valid. The authenticated state and derivation rule are themselves sufficient for the transition-validity test.

## 32 Logical Generations

DSM uses logical generations.

A generation is not a wall-clock timestamp.
Suppose:

$$V_0 \to V_1 \to V_2.$$

Then $V_2$ is later because it is derivable from $V_1$, not because a universal clock says it happened later.
This distinction is important.
DSM state progression is ordered by dependency.
It is not ordered by universal time.

## 33 Precommitment

Precommitment means the parent state commits its allowed candidate future space before one of those futures becomes realized.

Given parent $s$:

$$\boxed{P(s) = \{c_1, c_2, \ldots, c_n\}.}$$

For example:

$$P(s) = \{c_{\text{release}}, c_{\text{refund}}, c_{\text{recovery}}\}.$$

The three candidates are not three completed transitions.
They are three cryptographically committed possibilities.

## 34 Candidate Structure

A candidate may be written as:

$$\boxed{c_i = (s_i, b_i, g_i, K_i, d_i).}$$

where:

- $s_i$ is the candidate successor;
- $b_i$ is the branch identifier;
- $g_i$ is the guard descriptor;
- $K_i$ is the required resource-key set;
- $d_i$ is the candidate digest.

The digest is:

$$d_i = H(\mathrm{enc}(s_i)).$$

The candidate must be bound to the parent.
An arbitrary future state that was never committed cannot simply be inserted later.

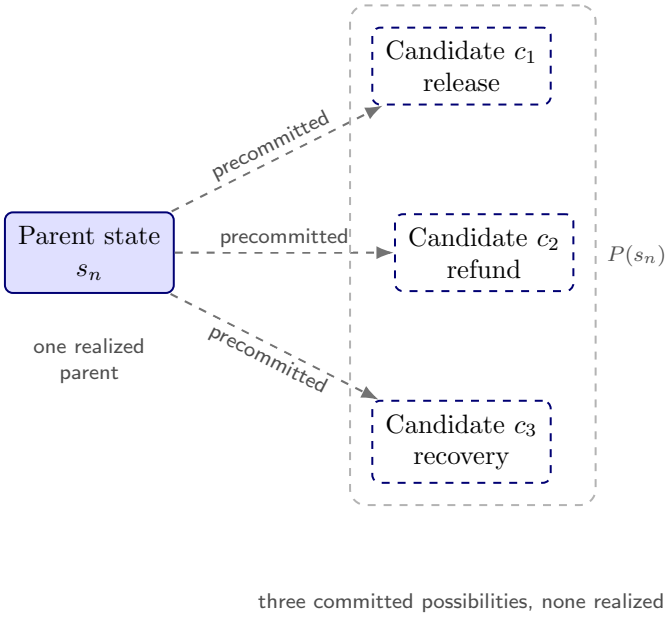



**Figure 12:** Precommitted Future Space

## 35 Candidate Forks Are Allowed

DSM permits:

$$|P(s)| > 1.$$

Therefore:

$$c_1, c_2, c_3$$

may all coexist as candidate branches.

This is intentional.
Candidate branching is not itself a double execution.
The safety theorem applies to *realized* conflicting branches.

Many candidates may exist.

One linear resource may be consumed once in a valid realized lineage.

## 36 Precommitment Chaining

Precommitment is not temporary metadata.
It is part of canonical state evolution.
State $s_n$ commits:

$$P_n, \qquad \Gamma_n.$$

Suppose candidate $c_i \in P_n$ realizes and creates:

$$s_{n+1}.$$

That new state commits a new future space:

$$P_{n+1}, \qquad \Gamma_{n+1}.$$

Thus:

$$(s_n, P_n, \Gamma_n) \to (s_{n+1}, P_{n+1}, \Gamma_{n+1}) \to (s_{n+2}, P_{n+2}, \Gamma_{n+2}).$$

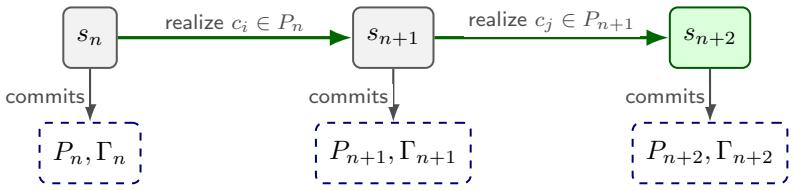



**Figure 13:** Chained Precommitment

Each realized successor becomes the parent of the next precommitted future space.

**Bitcoin Comparison**

Bitcoin outputs can encode spending conditions, and advanced Bitcoin constructions can commit alternative paths.
DSM makes the candidate successor set itself a first-class component of the general state model.

$$P(s) = \{c_1, \ldots, c_n\}.$$

**Architectural consequence:**
The verifier does not ask:

> "What arbitrary new state is somebody trying to execute?"

It asks:

"Is this successor one of the futures already committed by the parent?"

That sharply constrains the valid state-transition space.

## 37 Guards

A precommitted candidate does not automatically realize.

It must satisfy a deterministic guard.

For branch $b_i$:

$$g_i(s, w) \in \{\text{True}, \text{False}\}.$$

Here $w$ is witness material.

A witness may be:

- a signature;
- a cryptographic preimage;
- a completion certificate;
- a receipt;
- a recovery proof;
- a mode-specific evidence bundle.

A guard should bind:

- the parent;
- the branch identifier;
- the required resource keys;
- the accepted witness type;
- the deterministic witness predicate;
- the conflict class.

## 38 Guard Families

For state $s$:

$$\boxed{\Gamma_s = \{(b_i, g_i, K_i)\}_{i=1}^{n}.}$$

Each branch has a guard and a resource set.

A well-formed guard family requires deterministic verification and correct binding to the state and branch.

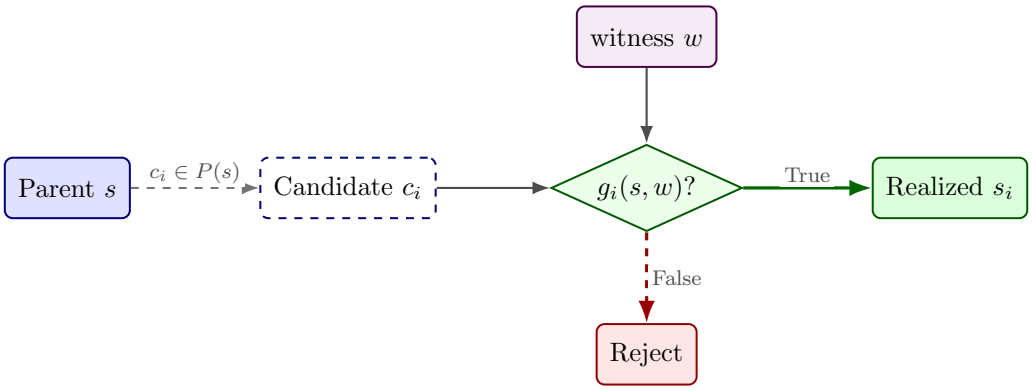



**Figure 14:** Guarded Candidate Realization

## 39 Guards Alone Are Not the Exclusion Mechanism

Suppose two conflicting guards are both satisfiable:

$$g_1(s, w_1) = \text{True},$$

$$g_2(s, w_2) = \text{True}.$$

DSM must still remain safe.

Therefore the safety theorem cannot rely only on assuming that every pair of conflicting guards is logically exclusive.

The load-bearing mechanism is shared resource consumption.

Conflicting branches must contend for the same authoritative linear-resource key.

**Bitcoin Comparison**

Bitcoin Script also provides deterministic spending conditions.
DSM's guard is integrated with a precommitted candidate successor and an explicit conflict-resource set.
Thus:

$$\text{candidate} \to \text{guard} \to \text{resource consumption} \to \text{successor}$$

forms one deterministic transition structure.
**Architectural consequence:**
Even overlapping successful guards do not imply that two conflicting successors can both become realized state, because shared linear-resource consumption remains independently enforced.

## 40 Linear Resources

A linear resource is an object whose committed generation can produce at most one realized successor inside its conflict class.

Examples include:

- a bilateral relationship parent;

- a spendable balance object;
- a vault generation;
- a recovery generation;
- an emission-source generation;
- an offline anchor step;
- another canonically defined single-consumption state object.

## 41 Resource Descriptors

Let $x$ be the canonical description of a resource.

For a relationship:

$$\boxed{x_{\mathrm{rel}} = (\texttt{relationship}, k_{A \leftrightarrow B}, h_n).}$$

The descriptor is derived from committed state.

A branch is not free to invent a different descriptor merely to evade conflict.

## 42 Resource-Consumption Keys

DSM derives the authoritative consumption key as:

$$\boxed{\kappa_{\mathrm{res}}(s, x) = H(\texttt{DSM/consume resource/v1} \parallel \rho_{\mathrm{core},s} \parallel x).}$$

Let:

$$K = \kappa_{\mathrm{res}}(s, x).$$

If two branches consume the same resource $x$, then both derive the same:

$$K.$$

That equality is the point.

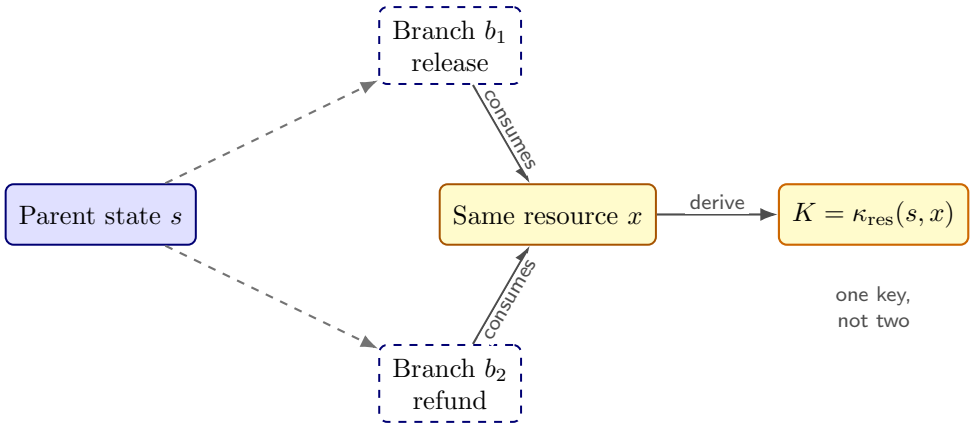



**Figure 15:** Two Branches, One Resource Key

The branches are different.

The consumed resource is the same.
Therefore the authoritative key is the same.

## 43 Why Branch-Specific Exclusion Keys Would Be Wrong

Suppose exclusion were defined as:

$$K_1 = H(s \parallel x \parallel b_1)$$

and

$$K_2 = H(s \parallel x \parallel b_2).$$

Then:

$$K_1 \neq K_2.$$

Both could appear unconsumed.
That would fail to represent the fact that both branches consume one parent resource.
DSM therefore derives the exclusion key from the resource, not from the choice of conflicting branch.
Branch-local identifiers may still exist for audit or indexing, but they are not the authoritative exclusion mechanism.

## 44 Conflict Classes

Define the conflict class for key $K$:

$$\boxed{C_K(s) = \{b_i : K \in K_i\}.}$$

The branches inside $C_K(s)$ are mutually conflicting with respect to the same linear resource.
This allows conflict to be explicitly scoped.
Not every branch in DSM conflicts with every other branch.
Only branches sharing relevant linear state need exclusion.

## 45 The Consumed Set

DSM records consumed resource keys in:

$$\Sigma_s.$$

Initially:

$$K \notin \Sigma_s.$$

After a valid transition consuming $K$:

$$\boxed{\Sigma_{s'} = \Sigma_s \cup \{K\}.}$$

Therefore:

$$K \in \Sigma_{s'}.$$

Consumption is monotonic:

$$\Sigma_s \subseteq \Sigma_{s'}.$$

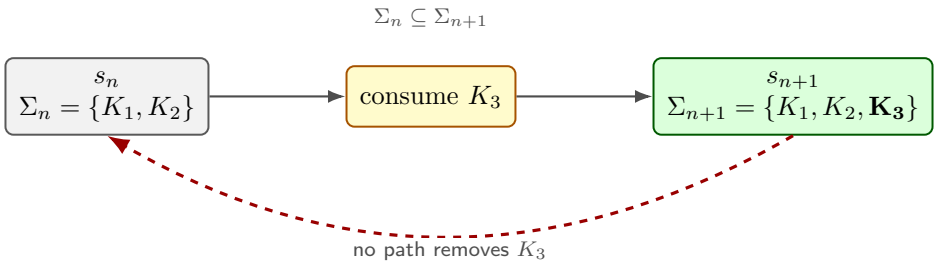



**Figure 16:** Monotonic Consumption

## 46 The Consumed Set Can Also Be an SMT

The consumed set may itself be represented as an authenticated sparse map.

For example:

$$\Sigma[K] = \begin{cases} 0, & \text{not consumed,} \\ 1, & \text{consumed.} \end{cases}$$

A proof can demonstrate:

$$K \notin \Sigma_s$$

before execution and:

$$K \in \Sigma_{s'}$$

after execution.

This ties linearity directly into authenticated state.

## 47 Bitcoin UTXOs Versus DSM Linear Resources

**Bitcoin Comparison**

Bitcoin's UTXO model already contains an important linear concept.
A UTXO can be consumed once.
Once spent in the accepted chain state:

$$U \notin \text{UTXOSet}.$$

DSM generalizes this idea from monetary outputs to arbitrary committed state resources.
A DSM linear resource can be:

- a monetary balance generation;
- a relationship parent;
- a vault generation;
- a recovery generation;

- a policy-controlled object;
- an offline anchor step.

**Architectural consequence:**
DSM promotes single-consumption semantics to a general state-machine primitive.
Instead of only asserting:

"this coin output cannot be spent twice",

DSM can assert:

"this committed state resource cannot produce two conflicting realized successors."

**Tradeoff / boundary:**
Bitcoin's narrower UTXO model is deliberately simple and is well suited to its purpose as a global monetary system.

## 48 The Complete Realization Predicate

A potential transition:

$$f_i : s \to s_i$$

becomes realized only if:

$$\boxed{\begin{aligned}\text{Realize}(s, s_i, w) = \ & \text{CandidateOK}(s, s_i) \\ & \wedge \text{GuardOK}(s, s_i, w) \\ & \wedge \text{StructuralOK}(s, s_i) \\ & \wedge \text{LinearityOK}(s, s_i) \\ & \wedge \text{PolicyOK}(s, s_i) \\ & \wedge \text{ModeOK}(s, s_i).\end{aligned}}$$

## 49 Potential and Realized Morphisms

A mathematically useful interpretation is:

$$f_i : s \to s_i$$

is a potential morphism.
It represents a possible structure-preserving transformation.
When all required predicates become true, the potential morphism becomes a realized morphism.
Therefore DSM can be understood as:

$$\boxed{\text{a deterministic detector of which precommitted state morphisms are currently realizable.}}$$

## 50 Realized Histories

A realized history is:

$$s_0 \to s_1 \to s_2 \to \cdots \to s_n$$

such that every edge is a valid DSM realization.

The consumed set threads forward:

$$\Sigma_0 \subseteq \Sigma_1 \subseteq \Sigma_2 \subseteq \cdots \subseteq \Sigma_n.$$

This is essential.

A branch is not repeatedly evaluated against an old snapshot while ignoring the state created by earlier realized transitions.

## 51 The Core Uniqueness Theorem

**Theorem 1** (Conflict-Local Realized Uniqueness)**.** *Let $b_1$ and $b_2$ be two conflicting branches over the same committed linear resource $x$.*

*Let:*

$$K = \kappa_{\text{res}}(s, x).$$

*Then $b_1$ and $b_2$ cannot both consume $K$ as separate realized transitions in one valid DSM history.*

*Proof.* Assume the opposite.

Initially:

$$K \notin \Sigma_s.$$

Suppose $b_1$ realizes first.

Validity requires:

$$\Sigma_1 = \Sigma_s \cup \{K\}.$$

Therefore:

$$K \in \Sigma_1.$$

Now suppose $b_2$ also realizes as another consumption of the same resource.

Linearity requires:

$$K \notin \Sigma_1.$$

But we already established:

$$K \in \Sigma_1.$$

Thus both statements would have to be true:

$$K \in \Sigma_1$$

and

$$K \notin \Sigma_1.$$

Contradiction.

Therefore both conflicting branches cannot be co-realized as separate consumptions of the same resource in one valid history.

□

### Two forms of the theorem

The formal paper states uniqueness in two forms, because guard families come in two kinds.

**Selector families.** Some conflict classes carry a deterministic selector: a vault decision function, a recovery decision object. At most one branch can ever be fulfilled, so at most one candidate passes the predicate even against the untouched parent. For these families the theorem holds *statically*, at the level of the predicate.

**Shared-key families.** Other conflict classes let several guards be true at once. Release, refund, and recovery witnesses may all be available. Every branch passes the predicate against $\Sigma_s$. For these families the theorem holds in its *realized-history* form: the first acceptance updates $\Sigma$, and every later conflicting candidate fails linearity against the updated set.

The realized-history form is the operational safety statement. It holds for every well-formed family, and it is the form that is machine checked as an invariant (Section 79).

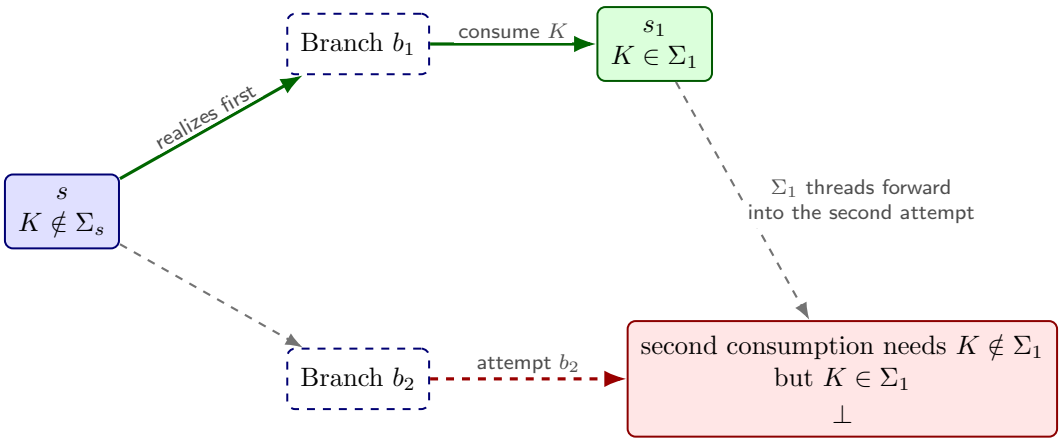



**Figure 17:** The Contradiction

## 52 Tripwire

DSM refers to the same-parent conflict exclusion property as **Tripwire**.

Tripwire does not mean malicious bytes cannot exist.

An attacker can construct:

$$c_1$$

and:

$$c_2.$$

The attacker can transmit both.

The theorem is not:

"conflicting bytes cannot be created."

The theorem is:

conflicting same-resource branches cannot both be derived as realized state in one valid lineage.

This distinction is important.
Anyone may write:

$$1 = 2$$

on a sheet of paper.
That does not mean the equation is derivable from ordinary arithmetic.
Likewise, anyone may manufacture conflicting serialized objects.
That does not make them jointly derivable DSM state.

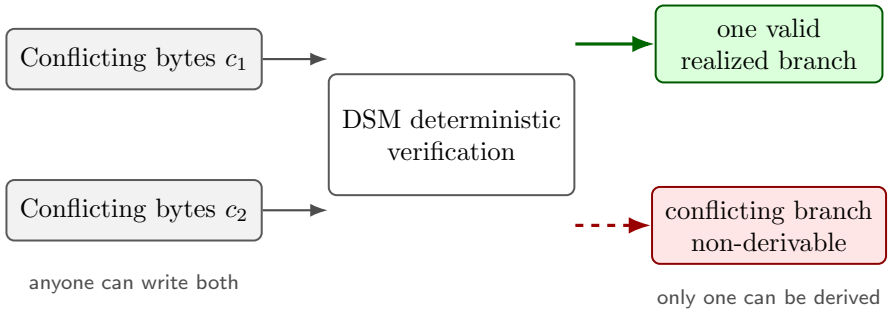



**Figure 18:** Conflicting byte strings may exist and may be transmitted, but deterministic DSM verification does not permit conflicting same-resource branches to coexist as realized state in one valid lineage.

## 53 Safety and Liveness

DSM safety does not imply guaranteed progress.
A valid candidate may fail to realize because:

- a counterparty refuses;
- required data is unavailable;
- a witness is never produced;
- storage is temporarily unreachable;
- policy conditions are not met;
- an offline proof cannot be produced.

Therefore:

safety $\neq$ liveness.

Safety answers:

> Can two conflicting realizations both become valid?

Liveness answers:

> Will some valid transition eventually occur?

These are different properties.

## 54 Concurrency Without Global Ordering

Suppose transition $f$ touches resource set $X$.

Suppose transition $g$ touches resource set $Y$.

If:

$$X \cap Y = \varnothing$$

and neither transition depends on state modified by the other, then they may commute:

$$f(g(s)) = g(f(s)).$$

Their relative order is unnecessary.

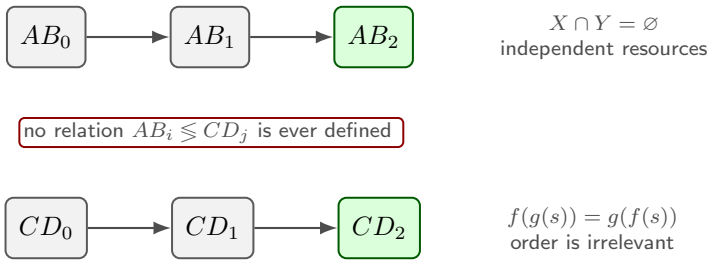



**Figure 19:** Independent Concurrent State

There is no need to define:

$$AB_1 < CD_1$$

or:

$$CD_1 < AB_1.$$

Both can be valid independently.

## 55 Token Conservation

Suppose Alice has balance:

$$B_A$$

and Bob has:

$$B_B.$$

Alice transfers amount:

$$\alpha.$$

Then:

$$B'_A = B_A - \alpha,$$

$$B'_B = B_B + \alpha.$$

Therefore:

$$B'_A + B'_B = (B_A - \alpha) + (B_B + \alpha).$$

The transfer terms cancel:

$$B'_A + B'_B = B_A + B_B.$$

So ordinary transfer is zero-sum.

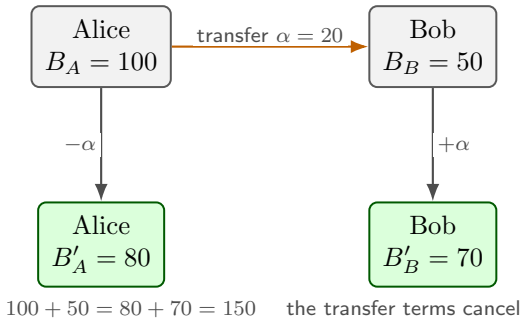



**Figure 20:** Conservation

## 56 Why Conservation Is Not Enough

Suppose Alice has 10.

Branch one:

$$A : 10 \to 0, \quad B : 0 \to 10.$$

Branch two:

$$A : 10 \to 0, \quad C : 0 \to 10.$$

Each branch individually conserves value.

But if both consumed the same source parent independently, the source would have effectively produced 20 units.

Therefore:

$$\boxed{\text{conservation} + \text{linearity}}$$

are both required.

# Part III
# What Is Built on It

Vaults, smart commitments, multi-party workflows, asset policy, emissions, and markets, each a precommitted branch family over linear resources.

## 57 Deterministic Limbo Vaults

A Deterministic Limbo Vault illustrates precommitted alternative outcomes.

Suppose vault generation $V_n$ allows:

$$P(V_n) = \{c_{\text{release}}, c_{\text{refund}}, c_{\text{recovery}}, c_{\text{abort}}\}.$$

Each branch has a distinct guard.

But all terminal branches consume:

$$x_{V_n}.$$

Therefore:

$$K_{V_n} = \kappa_{\text{res}}(s, x_{V_n}).$$

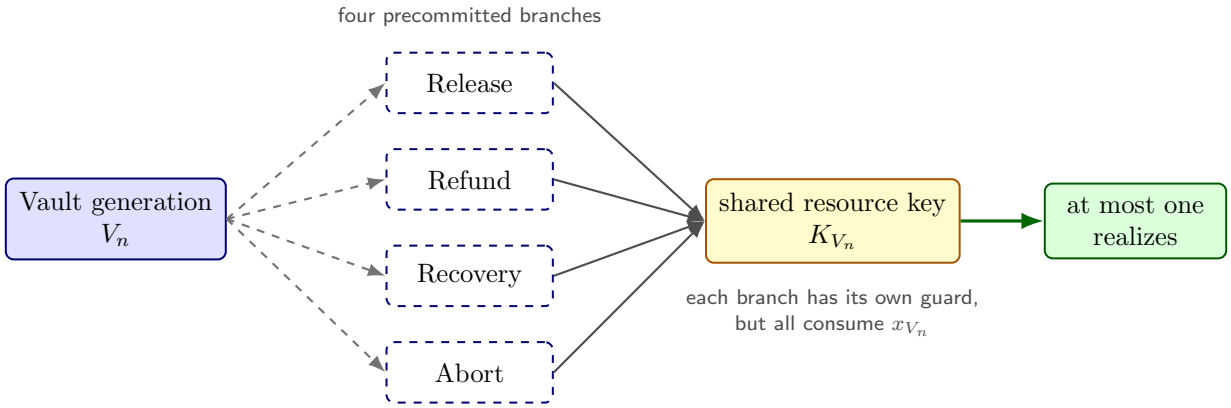



**Figure 21:** Deterministic Limbo Vault

Once one terminal branch consumes:

$$K_{V_n},$$

another terminal branch cannot consume the same generation in that lineage.

**Bitcoin Comparison**

Bitcoin can encode alternative spending conditions.
DSM explicitly models the alternatives as precommitted successor branches over one shared linear vault generation.
**Architectural consequence:**
The exclusivity of release, refund, recovery, and abort does not depend on a global transaction-ordering decision.
They are mathematically recognized as alternative consumers of one committed state resource.

## 58 Smart Commitments

An Ethereum reader will have been waiting for this section.

DSM has no contract virtual machine. That is a design decision, not a limitation discovered later, and the reasoning is worth stating in full because the obvious objection is "then it cannot do what Ethereum does."

### What a contract is

An Ethereum contract is code plus mutable storage, executed by every node in a globally agreed order. Any transaction can call any function; the function may loop, recurse, call other contracts, and modify storage arbitrarily. The global order is what makes the result well-defined, and the gas market is what keeps execution finite.

Everything that makes Ethereum powerful comes from that general-purpose programmable execution and dynamic composition. So does most of what makes it dangerous: reentrancy, unbounded recursion, dynamic call-stack behaviour, gas-budget execution failure, and the difficulty of knowing in advance what a transaction will do once it starts calling other contracts.

### What a smart commitment is

A smart commitment is a bounded deterministic predicate over committed inputs:

$$\mathcal{C} = \{\Delta_{\text{in}},\ \Delta_{\text{out}},\ \text{invariants},\ \text{external commitments},\ \text{encumbrances},\ \text{intent bounds},\ \text{budget}\}.$$

It is not code that runs. It is a guard family (Section 37) attached to a precommitted candidate space (Section 33) over linear resources (Section 40). The reader has already seen the whole mechanism; this section only names it.

A smart commitment can use:

- checked integer arithmetic and comparison;
- Boolean composition;
- hashing and signature verification;
- Sparse Merkle inclusion and non-inclusion;
- membership in committed bounded sets;
- iteration with a cardinality fixed at commit time.

It cannot use recursion, dynamic dispatch, or unbounded loops, and every predicate family declares a static evaluation budget.

### Why the restriction costs less than it looks

The things people build on Ethereum are not, for the most part, unbounded computations. They are conditional transfers: move this value if that condition holds. The unbounded VM is the delivery mechanism, not the product.

DSM keeps the product and drops the mechanism. Every application below is a precommitted branch family with guards over linear resources, verified by the counterparty who accepts the step.

| Ethereum pattern | DSM smart commitment | Where in this document |
|---|---|---|
| ERC-20 token | CPTA fungible policy | Section 60 |
| ERC-721 / soulbound | CPTA NFT / SBT with allowlist proofs | Section 60 |
| Multisig wallet | multi-party authority branches | Section 59 |
| Escrow, HTLC | DLV with hash-fulfillment guard | Section 57 |
| Airdrop, emission schedule | DJTE join-triggered emission from a source vault | Section 61 |
| Oracle | external commitment, co-signed reference window | Section 59 |
| AMM / DEX | DLV with committed market policy | Section 62 |
| Perpetuals, liquidation | precommitted liquidation branch, activity-based funding | Section 62 |
| Social recovery | recovery generation | Section 63 |
| Time-lock | iteration budget on the local chain | below |

### What is genuinely different

Two things do not carry across unchanged, and the honest version of this section says so.

**No clocks.** Ethereum has block timestamps and heights. DSM has neither, by design. Where a contract would say "after time $t$," a smart commitment says "after $n$ accepted transitions on this chain" using an iteration budget $\beta$ that decrements only on accepted local steps. This is strictly weaker for wall-clock deadlines and strictly stronger for determinism: no verifier ever disagrees about whether the condition holds.

**No open-ended composition.** On Ethereum any contract can call any other. In DSM a commitment can only reference what was committed when it was created: the candidate space, the guards, the external commitments by hash. Composition across parties is done by bilateral composition and shared external commitments (Section 59), not by a call stack.

Those are the two costs. In exchange, reentrancy, unbounded recursion, dynamic call-stack behaviour, and gas-budget execution failure are structurally excluded, and every outcome of a commitment is bounded by the committed candidate space and the static evaluation budget, so it can be enumerated before acceptance.

**Ethereum Comparison**

Bitcoin Script is deliberately not Turing complete. Ethereum's founding argument was that this was too restrictive to build applications on.
DSM takes Bitcoin's side of that argument and then shows the applications can be built anyway, because a precommitted candidate space with guards is expressive enough for conditional transfer, and conditional transfer is what the applications are.
**Architectural consequence:**
Bounded predicates over committed inputs mean a verifier can enumerate every possible outcome of a commitment before accepting it. There is no reentrancy, no gas griefing, and no "what does this contract actually do" question.
**Tradeoff / boundary:**
"Everything Ethereum does" means every application pattern in the table. It does not mean arbitrary computation. A workload that genuinely needs unbounded on-chain computation, rather than conditional transfer, is not a DSM workload.

## 59 Multi-Party Workflows Through External Commitments

DSM is bilateral at the protocol layer. A reader will reasonably ask how three or more parties agree on anything if every chain has exactly two ends.

The answer is that they do not agree *on* a chain. They agree on a hash.

### External commitments

An external commitment is a digest of data that lives outside the DSM core:

$$Y = H(\texttt{DSM/external/v1} \parallel X).$$

DSM never executes $X$. It verifies only equality, inclusion, signature, or proof predicates bound to $Y$. A guard can say "this branch is fulfilled if a witness proves $H(\cdot) = Y$," and nothing more is asked of $X$.

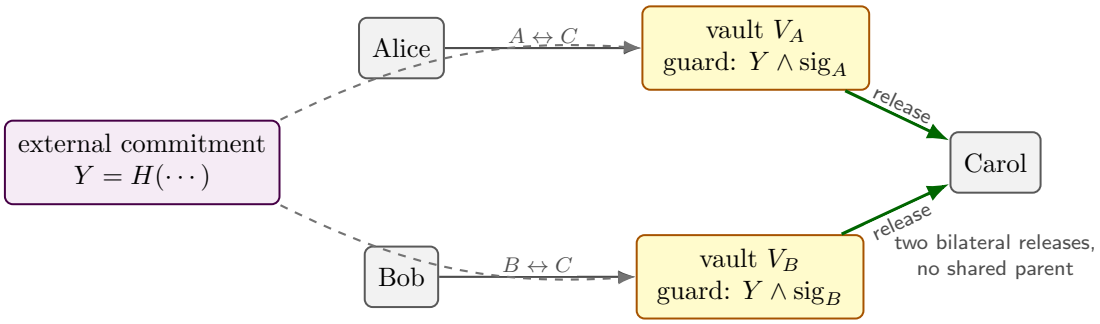
That one rule is enough to bind any number of bilateral chains to the same external fact, because two chains that both name $Y$ refer to that fact without sharing a parent.

### Three parties, two chains, one commitment

Carol is to be paid by both Alice and Bob if she delivers service $Y = H(\texttt{DSM/external/v1} \parallel \text{delivery proof})$.

There is no three-party ledger and no contract account. There are two relationships and one hash.

1. Alice funds a vault $V_A$ on $A \leftrightarrow C$ whose release guard requires $Y$ and Alice's acknowledgement.
2. Bob funds a vault $V_B$ on $B \leftrightarrow C$ whose release guard requires the same $Y$ and Bob's acknowledgement.
3. Carol presents the delivery proof in each relationship. Each release realizes independently as an ordinary bilateral step.




If Bob disputes and refuses to acknowledge, Alice's vault still releases or refunds under her own precommitted branches. Bob's refusal is a liveness failure on $B \leftrightarrow C$; it cannot touch $A \leftrightarrow C$.

### Authority does not multiply the parent

Multi-party guards can be thresholds or Boolean formulas over signatures:

$$g_{\text{release}} = \text{sig}(A), \qquad g_{\text{cancel}} = \text{sig}(B), \qquad g_{\text{recover}} = \text{sig}(C) \wedge \text{sig}(D).$$

Suppose every witness for every branch is available at once. Release, cancel, and recover are all fulfilled. This is the shared-key situation of Section 7: all three branches consume the same derived key for the resource they share, so the first one folded into the lineage marks it consumed and the other two fail linearity.

authority chooses an eligible branch; it cannot multiply the consumed parent.

This is why DSM does not need a contract account to referee between authorised parties. Overlapping authority does not multiply the resource: authority changes branch eligibility, and it does not permit more than one consumption of the parent.

**Bitcoin Comparison**

Bitcoin multisig puts all signers on one output and settles the result on one chain. Ethereum puts all parties into one contract's storage.

DSM keeps every party on their own bilateral chains and binds them through a shared hash. No party ever writes to another party's chain.

**Architectural consequence:**

There is no shared mutable object for the parties to contend over, so there is no ordering question between them. Each bilateral step is accepted by its own counterparty against its own frontier; the external commitment is what makes those independent steps refer to the same fact.

**Tradeoff / boundary:**
Atomicity across the bilateral legs is not automatic. If Alice's leg releases and Bob's does not, that is the outcome, and each vault's precommitted refund or abort branch is the remedy. Workflows that require all-or-none across several parties' resources use the one-bundle settlement of Section 62.

## 60 CPTA and Deterministic Policy

DSM can place authority and policy inside committed state.

Let $\Pi$ represent a deterministic policy. Possible operations might include:

$$\text{transfer}, \quad \text{claim}, \quad \text{emission}, \quad \text{burn}, \quad \text{recovery}.$$

An authorized actor may satisfy a policy condition. That does not exempt the actor from linearity. Even an authorized branch must obey $K \notin \Sigma$. Therefore:

$$\boxed{\text{authority determines eligibility; it does not override single-consumption state.}}$$

### A token is an asset with a policy, not a contract with an API

The Ethereum reader will map CPTA onto ERC-20 and get it slightly wrong, so it is worth being exact about the difference.

ERC-20 is not a boundary on what a token can be. It is a standard interface: balances, transfers, approvals, total supply, allowances. A token contract implements that interface and may put any logic it likes behind it. Interoperability comes from every application calling the same function names. The token's rules live in the token contract, and once a token is handed to another contract, a pool, a lending market, a bridge, it is that contract's code that decides what happens to it next.

A CPTA token is the other way round. The asset is bound to a committed policy describing which state transitions involving that asset are valid: who may issue it, whether further issuance exists, what conditions a transfer must meet, what authority each operation requires, which operations exist at all. The policy is a hash-committed input to every verifier that ever touches the asset. It does not have a function interface, and nothing implements it; it is checked.

The consequence is that the rules travel with the asset. When a CPTA token enters a limbo vault, the vault has its own policy and the token keeps its own. A market swap of $A$ for $B$ is valid only if every one of the following holds:

$$P_{\text{DLV}} \ \wedge \ \Pi_A \ \wedge \ \Pi_B \ \wedge \ \text{reserve arithmetic} \ \wedge \ \text{economic-root admission}.$$

There is no operation that lets an application bypass a token's policy by wrapping it, because validity is the *intersection* of the operation's rules and each asset's rules. If $\Pi_A$ says a movement is invalid, putting $A$ in a vault does not make it valid.

The last term deserves one sentence so it is not misread. Economic validity is decided by the predicates; a valid transition produces an exact economic write set; that write set is admitted into the device's economic root, which DSM's implementation calls $R_{\text{econ}}$; and the write-once register of Section 10 then makes the admitted root durable and conflict-detectable. The order is

$$\text{economic validity} \ \rightarrow \ \text{exact write set} \ \rightarrow \ \text{economic-root admission} \ \rightarrow \ \text{register: anti-equivocation}.$$

The register is the last step and only the last step. It is not a policy engine and it decides nothing about whether the transition was valid.

So DSM is simultaneously more customisable and more restrictive than ERC-20. The creator has considerable freedom in defining the policy. Once committed, no later transfer, vault, or application can route around it.

### What the combination enables

The intersection rule is easy to state and easy to underrate. Here is what it actually buys once a CPTA asset is inside a limbo vault.

**A market cannot validly move a token in a way its committed policy forbids.** On Ethereum a token's safety inside a pool depends on the pool contract. If the pool is upgradeable, buggy, or malicious, the token's own rules do not help; they ended at the contract boundary. In DSM the vault's policy $P_M$ can only *narrow* what $\Pi_A$ already permits. A vault that tried to move $A$ in a way $\Pi_A$ forbids produces an invalid successor, and the trader's own verifier rejects it. There is no "audit the pool to know if the token is safe there" step, because the token is checking.

**Compliance-scoped assets can trade in open markets.** A token whose policy says "only to holders of credential $c$" can be placed in a public vault. Anyone can read the vault; only a credentialed trader can produce a valid fulfilment, because $\Pi_A$ is evaluated on the market successor like on any other transfer. The issuer did not have to build a permissioned exchange, and the market did not have to know the rule existed.

**Supply commitments survive every hop.** If $\Pi_A$ fixes issuance, no vault, route, or emission mechanism can mint more $A$, because each of them is just a transition over $A$ and each is checked against $\Pi_A$. A Bitcoin reader will recognise this: the twenty-one-million cap holds in every context because every context is a transaction. CPTA gives an arbitrary asset the same property.

**Multi-asset routes check themselves.** A route through several vaults touching $A$, $B$, and $C$ satisfies each asset's policy on each leg automatically, since every leg is a transition and every asset carries its own predicate. The route constructor does not assemble a special safety argument; the intersection does it.

**Issuer, vault, and trader are three independent authors.** The issuer committed $\Pi_A$ once. The liquidity provider committed $P_M$ and $P_R$ once. The trader supplies the live side. None of the three can override the other two, none needs to be online for the other two's rules to hold, and none had to coordinate with the others when writing them.

That last point is the one worth sitting with. Three parties wrote three predicates at three different times without talking to each other, and any market transition must satisfy all three. That is composability without a shared contract, and it falls out of nothing more than "validity is an intersection."

### What a token policy should not say

A natural next question: does every token then need a rule saying which other tokens it may be exchanged for? No, and it would be a mistake to design it that way.

A token's policy says what makes a transfer of *that token* valid. It does not enumerate trading pairs. The exchange relationship between $A$ and $B$ is the vault's business, in $P_{\mathrm{DLV}}$. For a swap, $\Pi_A$ asks whether this movement of $A$ is permitted under $A$'s rules; $\Pi_B$ asks the same for $B$; the vault policy asks whether this $A$-for-$B$ exchange is permitted under this vault's rules.

A more restrictive token can certainly say "only against approved assets," "only through approved vault policies," or "only with counterparties who hold this credential." That is an optional capability, and it is how soulbound and compliance-scoped assets are built. It is not the default, because a default that required editing the token's committed policy every time someone opened a legitimate new pair would defeat the purpose of having a stable committed policy at all.

**Ethereum Comparison**

An ERC-20 token's rules live in its contract and stop at the contract's boundary; once the token is inside a pool or a bridge, that application's code decides its fate.
A CPTA token's rules are a committed policy that every verifier evaluates on every transition the asset is part of, including transitions inside vaults and markets.
**Architectural consequence:**
Asset-level invariants hold across every financial operation, not only inside the token's own contract. The token's policy, the vault's policy, and the arithmetic must all pass; there is no wrapping trick.
**Tradeoff / boundary:**
This is a claim about enforcing asset-level invariants, not a claim that DSM is universally more secure than Ethereum, which has a mature and well-studied security model of its own. And the flexibility cuts both ways: a badly written policy is committed just as firmly as a good one. Cryptographic commitment guarantees faithful enforcement of the policy, not that the policy was well designed.

# 61 Deterministic Emissions (DJTE)

A Bitcoin reader has a specific question about any token system: where do the units come from, who decides who gets them, and what stops that party from printing more.

Bitcoin's answer is mining: a coinbase per block, a fixed schedule, halvings, and proof-of-work as the lottery. That answer needs a chain, a block cadence, and a clock. DSM has none of those, so it needs a different one.

## Emission is a vault transition, not minting

All units exist from the start inside a locked source vault under a CPTA policy. Nothing is ever created; the vault *reveals* units by moving them out. The vault state at emission index $e$ carries the remaining supply, and every emission transition must prove

$$\text{remaining}_{e+1} = \text{remaining}_e - \text{amount}_e, \qquad \text{amount}_e \le \text{remaining}_e.$$

That is the entire supply-cap argument. The remaining counter starts at the total and can only decrease; when it reaches zero the vault emits zero forever. The schedule is halving-shaped: sixteen epochs, each with twice the emission count and half the per-emission amount of the last. It will look familiar.

## What triggers an emission

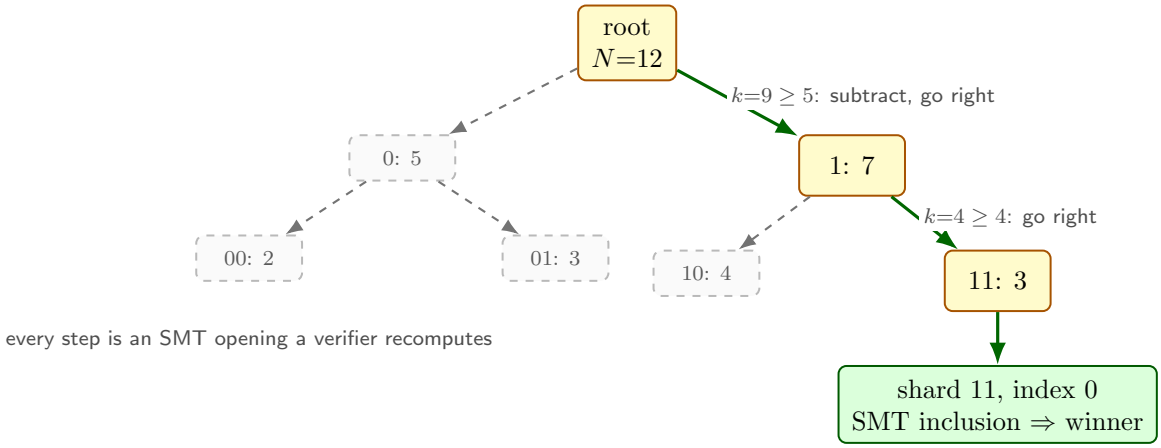
There is no block to trigger it, so DJTE is *join-triggered*. When a device passes the spend-gate (it has paid for its genesis storage replicas), it produces a Join Activation Proof. Each valid proof is consumed exactly once to drive one emission, by the same mechanism the reader has now seen several times: the proof's hash must be proved absent from a spent-set Sparse Merkle Tree before, and present after.

So the emission rate is set by how many devices join, and nothing else. No timer, no difficulty adjustment, no miner.

## Who receives it

The recipient is drawn uniformly over *every* activated identity, and the draw is proof-carrying: the transition must carry evidence that the selection was computed correctly, and any verifier can check it offline.

1. Seed: $R_0 = H(\texttt{DJTE.SEED} \parallel \text{tip} \parallel e \parallel \text{jap\_hash})$. Nothing in it is choosable after the fact.
2. Total: a proof from a count tree that there are $N$ activated identities.
3. Rank: $k = \text{UniformIndex}(R_0, N)$, exact rejection sampling, no modulo bias.
4. Descent: walk the count tree from the root. At each level, if $k$ is less than the left child's count go left; otherwise subtract it and go right. After $b$ levels you are at one shard with a local index.
5. Inclusion: a proof that the leaf at that index in that shard is the winner.




Because ranks are routed by counts, a shard with more identities gets proportionally more draws and an empty shard gets none. That is not a skew; it is exactly what uniform-over-individuals means. The paper proves the sharded descent is equivalent to sampling the concatenated global list, and that the $O(b)$ descent cost is information-theoretically minimal: choosing one of $2^b$ shards needs $b$ bits, so no cheaper proof-carrying scheme exists.

### No coordinator

Nobody runs the lottery. The seed is a hash of committed state, the count tree and shard accumulators are committed roots, and the emission transition is verified by whoever receives it. Storage nodes mirror the objects and are not asked to sign, order, or count anything. Two verifiers holding the same committed state compute the same winner, and the source vault's generation is a linear resource exactly like every other, so two competing emissions from one $D_e$ collide on the same key.

**Bitcoin Comparison**

Bitcoin's issuance is a lottery weighted by hashpower, run by block cadence, capped by a schedule every node enforces.
DJTE's issuance is a lottery weighted uniformly over activated identities, run by device joins, capped by a decreasing counter inside a vault every verifier enforces.
**Architectural consequence:**
The lottery is a deterministic function of committed roots and a consumed join proof. There is no miner to bribe, no timing to manipulate, and no party who could emit more than the counter allows.
**Tradeoff / boundary:**
Uniform-over-identities is only as fair as identity creation is expensive. DJTE is mechanically correct under Sybil behaviour, but the *social* fairness depends on the spend-gate making identity grinding cost more than an emission is worth. That is DSM's spend-gate property, not DJTE's.

### Aside: spam resistance without a global mempool

The DJTE paper also answers a question a Bitcoin reader will ask about any system with no mempool and no fee market: what stops flooding?

Most of the answer is structural. A transition must bind to a receiver's committed context (the expected parent, index, and policy anchors), so random bytes cannot be represented as a valid candidate at all and fail the cheapest check first. Inboxes are contact-gated: a device processes objects only from relationships it has established, so a victim can stay fully online while declining to ingest a hostile sender. And a zero-effect transition is not a transition, so "send nothing a million times" is not representable.

The residual case, valid high-volume traffic against storage capacity, is priced by prepaid credit bundles: a sender-authored, state-advancing, accepted transition costs one credit, and refills go through the spend-gate. Receiving never debits, so a victim's credits cannot be drained remotely. There is no time window in any of this. Bitcoin's fee market solves congestion with a clock and a global queue; DSM removes the global queue and prices only the sender's own accepted steps.

## 62 Sovereign Finance (SoFi)

The hardest test of "you can build the applications" is a market. A decentralised exchange needs liquidity that trades while its owner is asleep, prices that are deterministic, and no operator who can reorder trades. On Ethereum that is a pool contract plus the global order.

SoFi is DSM's answer. It is built entirely from pieces this document has already introduced, plus one scoped mechanism it introduces honestly.

### A vault holds the liquidity

A liquidity provider funds a Deterministic Limbo Vault by moving value *out* of ordinary spendable balance and *into* encumbered vault reserves:

$$B_{\mathrm{LP}} \xrightarrow{\text{fund}} R_{\mathrm{DLV}}.$$

There is no second spendable copy. The LP owns and controls the vault by committing its policies at creation:

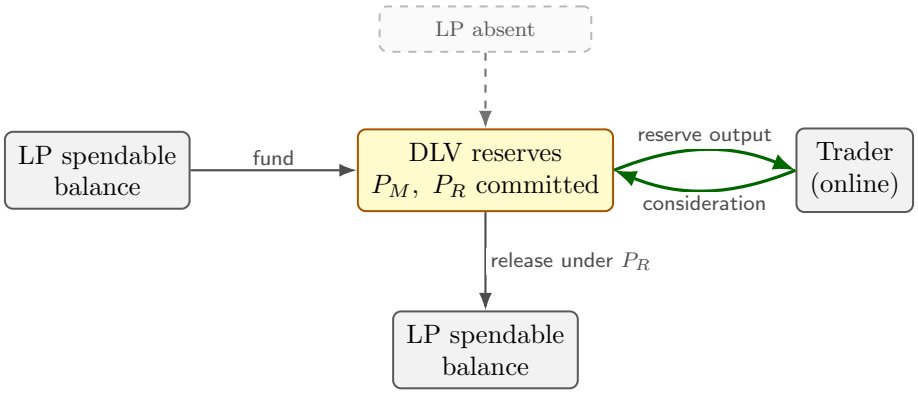
- $P_M$: the bounded market policy (the invariant, fees, and a per-transition size ceiling);
- $P_R$: the release/close policy under which reserves may come back to ordinary balance.

Once funded, the LP is bound by those policies exactly as a trader is. The owner cannot reach into the vault and take the reserves out; a release is a governed DLV transition like any other.

### Why an absent LP can trade

In an ordinary DSM relationship, a present party can send its own value toward an absent one but cannot make the absent party's value move outward. There is no fresh authority for that.

A funded vault *is* the authority, committed in advance. The value is already inside an executable state object and the conditions under which it may be exchanged were fixed before any trader existed. A live trader supplies consideration and receives the corresponding reserve output while the LP is absent. The trader is online; only the LP is away.




reserves move only through committed policy, in either direction

## The one new problem, and the one new mechanism

Everything above is bilateral DSM. But a public vault is executable by *anyone*, and two unrelated traders can read the same vault parent and each construct a valid fulfilment before either sees the other. Within one trader's own chain, Tripwire handles it. Across two strangers' chains there is no shared history to be linear in.

**Client-side binding over a shared vault parent.** A public vault creates one additional problem: unrelated traders may concurrently construct valid candidates from the same vault parent. The LP therefore commits a storage-member set $S$ and a threshold $q$ when the vault is created. Each member independently enforces write-once semantics for the vault-parent binding key and returns the value it holds.

The trader's SDK authenticates those responses and locally determines whether $q$ distinct members acknowledge the same binding value for the derived vault-parent key. The members do not parse or validate the trade, do not know or evaluate $q$, do not vote with one another, and do not produce a collective decision.

The resulting object is a client-assembled binding certificate over one named resource, not a replicated transaction log and not a consensus result. It establishes no ordering between unrelated vaults or transactions.

A reader may still want to call it consensus. Call it that, then, and ask the question that actually matters: where is the bottleneck? A blockchain's consensus is a bottleneck because every transaction in the system passes through one ordering decision. Here there is no such decision. A binding over vault $V_1$ and a binding over vault $V_2$ touch different keys, are computed by different traders, and never wait for each other. Ten thousand vaults trading at once are ten thousand independent certificates, each the size of one quorum read, and adding a vault adds a key, not a load on anything shared. Whether the word "consensus" applies to a five-member write-once read is a question about labels. What the model achieves is that there is no shared sequencer whose throughput caps the system, and that is the property the label was supposed to be about.

## Binding is not realization

Once the client obtains $q$ valid matching acknowledgements, that certificate makes one candidate *binding-final* for the vault parent under the vault's committed threshold rule: no competitor can consume it. It does *not* move the reserves.

Reserves move only when the trader's own bilateral successor is accepted under ordinary DSM and produces an acceptance witness carrying the accepted successor commitment, its authentication, and an inclusion proof under the accepted root. Only binding plus that witness is a realized settlement.

$$\text{COMMIT} \rightarrow \text{parent locked} \rightarrow \text{trader accepts own step} \rightarrow \text{witness} \rightarrow \text{reserves advance}$$

The consequence is that a trader who wins the lock and then refuses to complete cannot distort the vault's advertised reserves, fabricate an input, or force a price change. What they can do is leave the parent locked. That is an availability cost, stated plainly in the specification, not a safety hole.

### What this gives you

- **Independent vaults, atomic aggregation.** A large trade may draw from several LPs' vaults at once. The route binds all of their parents in one settlement bundle; it executes fully or not at all. The vaults remain independently owned. There is no pooled custody.
- **Deterministic pricing.** Allocation and pricing are checked integer arithmetic; two conforming SDKs given the same vault states produce byte-identical routes.
- **No global mempool or validator-reordering MEV surface.** There is no global pending queue and no validator with a reordering surface. A settlement bundle is visible to its storage set before binding, but visibility grants no priority.
- **Perpetuals and liquidation.** Funding is denominated in trade activity rather than elapsed time, and liquidation is a precommitted branch the position holder committed at open. No clock, no oracle feed as authority; reference prices are co-signed windows over verified trade digests.
- **Clockless.** No timestamp, height, or duration appears in any validity rule.

**Ethereum Comparison**

On Ethereum a DEX is a pool contract: pooled custody, global ordering, priority-fee auctions, and a well-documented MEV industry built on watching the mempool.
SoFi has sovereign vaults inside each LP's own state, local verification, horizontal aggregation without pooling, and no ordering surface to extract from.
**Architectural consequence:**
Liquidity trades while its owner is absent without handing custody to a contract, and the only shared decision in the whole system is a consume-once lock on one vault parent, computed by the client from an owner-chosen storage set.
**Tradeoff / boundary:**
Two costs are explicit in the specification. First, a market trade is online: it needs the vault's storage set reachable, so SoFi does not inherit DSM's offline-bearer path. Second, a trader who wins binding and walks away can lock that vault parent indefinitely; the beta profile defines no timeout escape, because a timeout would be a clock. Both are liveness properties. Neither lets reserves move without a completed bilateral exchange.

## 63 Recovery as a Linear Generation

Suppose recovery generation $R_n$ permits:

$$P(R_n) = \{c_{\text{succession}}, c_{\text{cancel}}, c_{\text{failure}}\}.$$

All three terminal alternatives consume:

$$x_{R_n}.$$

Therefore:

$$K_R = \kappa_{\mathrm{res}}(s, x_{R_n}).$$

Once one terminal outcome realizes:

$$K_R \in \Sigma.$$

The same recovery generation cannot terminate a second conflicting way.

# Part IV
# Offline DSM and Conflict-Local Finality

The offline operating mode, what it depends on, what a two-recipient attack looks like there, and what finality means when there is no chain to wait for.

## 64 Offline Bearer DSM

Offline operation creates a physical-state problem.

If two devices cannot consult online infrastructure, there are two separate questions:

1. Is the state transition mathematically unique under DSM?
2. Did the transition originate from the intended physical device?

DSM keeps these questions separate.
Software linearity answers the first.
Hardware identity evidence can strengthen the second.

## 65 Offline Origin

An offline origin may be committed by coordinates such as:

$$(R_i, h_i, u_i),$$

where:

- $R_i$ identifies the authenticated device/root state;
- $h_i$ identifies the relevant frontier;
- $u_i$ is the committed progression coordinate.

All candidates originating from the same anchor step consume the same offline resource:

$$x_{\mathrm{offline}}.$$

Therefore:

$$K_{\text{offline}} = \kappa_{\text{res}}(s, x_{\text{offline}}).$$

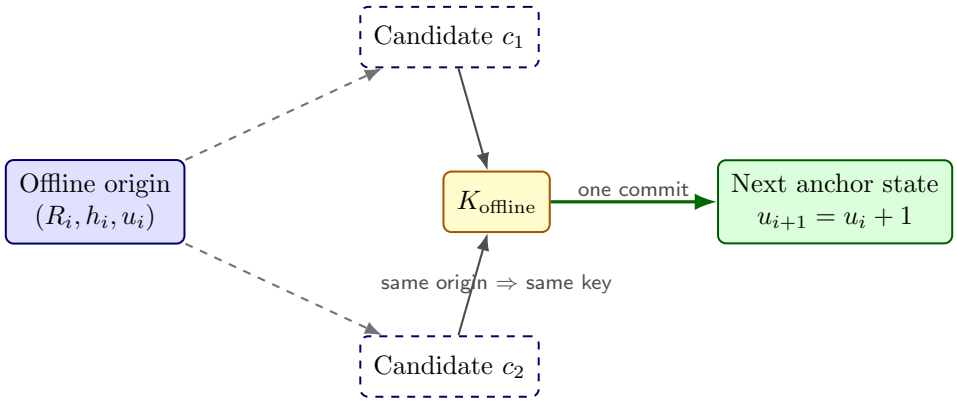



**Figure 22:** Offline Anchor Step

## 66 The Committed Software Counter

Suppose:

$$u_i = n.$$

A valid next anchor generation requires:

$$u_{i+1} = n + 1.$$

The new value is committed into authenticated state.
Thus logical monotonicity belongs to the software state machine.

## 67 Hardware Counter Versus Software Authority

A hardware counter may be useful for:

- anti-rewind evidence;
- clone resistance;
- stale-state detection;
- physical-device exposure limits.

But the hardware counter is not the transaction authority.
The uniqueness theorem remains:

$$K \notin \Sigma$$

before consumption and:

$$K \in \Sigma'$$

after consumption.

## 68 The Observer Model, Inverted

Anyone who has looked at offline digital cash will recognise the shape of a secure element that co-signs payments.

Chaum and Pedersen proposed it in 1992 as the *wallet with observer*: a tamper-resistant chip inside the user's wallet that co-signs every payment and refuses to sign the same value twice. Brands' cash sits in the same model. Fielded smartcard purses used it, and most recent offline CBDC and secure-element payment designs still do.

In that model the hardware *is* the double-spend authority. It is trusted to enforce uniqueness. That means the protocol inherits the hardware's entire attack surface as its own: extract the key, break the counter, or emulate the chip, and double spending follows.

DSM refuses that assignment.

Transfer uniqueness, offline included, is a software theorem of the guarded linear kernel (Sections 51 and 65). It holds with the anchor hardware deleted. The proof mentions no chip, no counter read, and no measurement.

What software cannot do is tell an enrolled device from a perfect copy of its host-readable bytes. That is the one job left, and it is the only job the hardware is given.

| | Observer model | DSM offline bearer |
|---|---|---|
| Who enforces uniqueness? | the chip | the state predicate |
| What does the chip prove? | "this value was not spent" | "this is the enrolled device" |
| Break the chip and... | double spend | forge a device, not a transfer |

The last row is the inversion. In the observer model a hardware compromise is a protocol compromise. In DSM a hardware compromise lets an attacker impersonate a device; it does not let two successors of one origin both become realized state, because that exclusion never lived in the hardware.

**Bitcoin Comparison**

Bitcoin readers know hardware wallets, and know that a hardware wallet protects a key, not the ledger. If the wallet is compromised the coins can be stolen, but the UTXO set does not become inconsistent.

DSM holds the same line one layer deeper. The offline anchor protects the identity of the physical device, not the uniqueness of the transfer. Uniqueness comes from the same resource-consumption theorem that runs online.

**Architectural consequence:**

The security argument for offline transfer does not reduce to "trust the chip." The chip answers one narrow question, and the answer to that question cannot create a second valid consumption of a consumed resource.

**Tradeoff / boundary:**

A perfect live clone of every non-exportable factor is indistinguishable from the original by any offline-only protocol. That is the stated boundary of the identity claim. It is a boundary on *who the device is*, not on *whether a transfer is unique*.

## 69 Three-Factor Offline Identity

An offline release may be supported by multiple signatures over the same state-advance message.

Conceptually:

1. seed-rooted DSM signature;

2. chip-rooted non-exportable signature;
3. measurement-gated host/partition signature.

These witnesses answer:

Did this state advance come from the enrolled physical execution environment?

They do not replace the software state theorem.

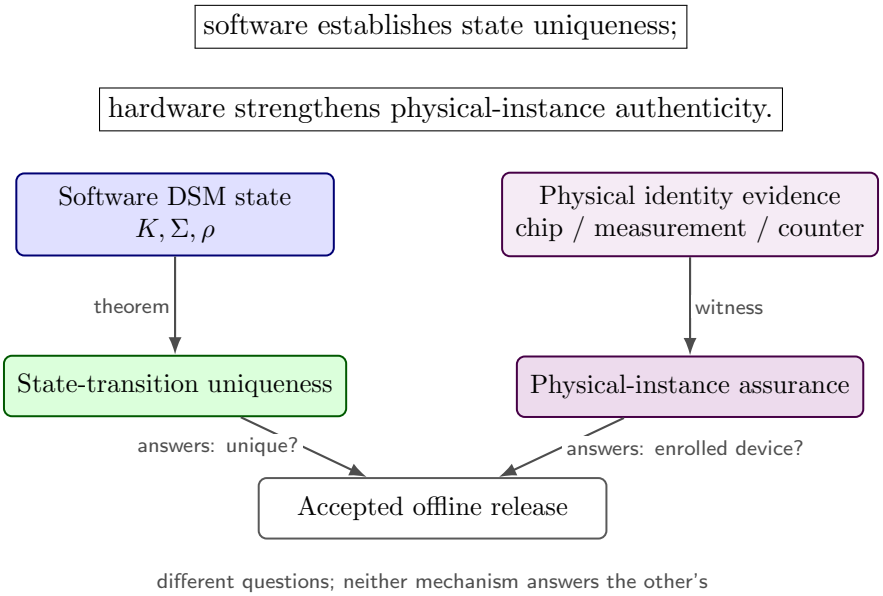



**Figure 23:** Software and Hardware Roles

**Bitcoin Comparison**

Bitcoin transactions can be signed offline, but ordinary settlement requires eventual reconciliation with the Bitcoin network's accepted chain state.
DSM's offline mode is designed around an explicit authenticated state origin and locally verifiable linear-resource progression.
**Architectural consequence:**
Offline uniqueness still comes from the same software resource-consumption theorem used online. Hardware adds identity and anti-clone evidence without becoming the canonical transaction register.

## 70 Two Offline Recipients

Part I traced the online double spend to its refusal at the register. The offline case deserves the same treatment, because it is where every offline-cash design in history has failed, and because the previous sections have described the machinery abstractly.

### The setup

Alice's device holds an offline origin at coordinates $(R_i, h_i, u_i)$ with $u_i = n$. She is in a place with no connectivity, and so are Bob and Carol. She pays Bob from that origin, then walks across the room and pays Carol from the same origin.

Neither recipient can reach the register. Neither can reach the other. Each sees only what Alice hands them.

### What each recipient sees

Bob receives a state advance from $(R_i, h_i, n)$ to $(R_{i+1}, h_{i+1}, n+1)$, consuming $K_{\text{offline}} = \kappa_{\text{res}}(s, x_{\text{offline}})$, carried by the three-factor witness of Section 69: the seed-rooted DSM signature, the chip-rooted non-exportable signature, and the measurement-gated partition signature, all over the same state-advance message. Bob verifies the software predicate against the origin, verifies each witness against the enrolled device's committed identity, and accepts.

Carol is handed something that claims to be the same thing: an advance from $(R_i, h_i, n)$, consuming the same $K_{\text{offline}}$, with a three-factor witness. She has no way to ask whether $n$ has already been consumed. So the question is not what she can observe about Alice's history. It is whether Alice can *produce* that object at all.

### What the genuine device can and cannot do

After paying Bob, the enrolled device's committed progression coordinate is $n+1$. The software state machine will only construct the next advance from $n+1$. To pay Carol from $n$, Alice needs a state-advance message from the stale coordinate, signed by all three factors.

The seed-rooted signature she can produce: it is software, she holds the seed, she can run modified code. The other two she cannot. The chip-rooted signature is issued only inside the measured execution environment, and the measurement gate binds that signature to the enrolled partition running the enrolled code. Modified code that would sign a stale coordinate is not the measured code, so the partition witness fails, and the chip does not co-sign an advance the partition did not attest. Carol's verifier sees a witness with one valid factor of three and refuses.

This is the whole content of Section 69 made concrete. The software theorem says two consumptions of $K_{\text{offline}}$ cannot both be realized in one lineage. The hardware does not repeat that theorem. It answers the only question the theorem leaves open offline: did this advance come from the enrolled device running the code that respects the theorem?

### The clone

Now suppose Alice has done the expensive thing: extracted every non-exportable factor and built a second device that is, at the moment of use, indistinguishable from the enrolled one. Both devices hold the origin at $n$. Device one pays Bob; device two pays Carol. Both witnesses verify. Both recipients accept.

This is the stated boundary, and the document does not pretend otherwise: a perfect live clone of every factor is indistinguishable from the original by any offline-only protocol. That was true of the Chaum–Pedersen observer, of every smartcard purse, and of every offline CBDC design, and it is true here. What differs is what happens next.

### What happens later

Both advances name position $n+1$ under the same device identity. The first of Bob or Carol to reconnect registers Alice's root at $n+1$. The second finds the cell already holding a different root. The register refuses the second write and returns the held digest.

At that moment the loser holds something no observer-model system provides: two witnessed state advances from one committed origin, each signed by the same chip-rooted identity, naming the same position with different roots. That is not a suspicion. It is a proof of equivocation, attributable to a physical device identity rather than to an anonymous key, and it is verifiable by anyone who holds both objects.

What the protocol does with that proof is policy: the device identity is burned, the losing recipient has an attributable claim, and the value at stake was bounded in advance by the physical-device exposure

limits that the offline policy committed. What the protocol does *not* do is let both advances become realized state. Only one root occupies $n+1$; only one consumption of $K_{\text{offline}}$ is in any lineage.

### The two questions, answered separately

Section 64 said offline operation poses two questions and that DSM keeps them apart. The trace shows why that separation is the whole design.

Is the transition unique? Yes, by the software theorem, with the hardware deleted: two consumptions of one $K_{\text{offline}}$ cannot both realize, and the register makes that fact durable the moment either recipient reconnects.

Did it come from the enrolled device? Yes if the witness verifies and the device was not perfectly cloned; and if it was, the clone is detected and attributed at reconnection, with the loss bounded by committed exposure policy.

**Bitcoin Comparison**

An unconfirmed Bitcoin payment accepted offline is exposed until the transaction reaches the chain, and if a conflicting transaction is mined first, the merchant holds nothing and can prove nothing about who defrauded them.
An offline DSM payment is exposed only to a perfect live clone of the payer's enrolled device, and if that clone is used, the defrauded recipient holds a verifiable, device-attributable proof of equivocation the moment either party reconnects.
**Architectural consequence:**
The offline double spend requires hardware extraction rather than a race, is detected at first reconnection rather than never, and produces attributable evidence rather than a silent loss.
**Tradeoff / boundary:**
Detection is not recovery. Whether the defrauded recipient is made whole is a question for the exposure policy and the issuer, not for the state machine. And the clone boundary is real: no offline-only protocol distinguishes a perfect clone, and DSM does not claim to.

## 71 Conflict-Local Finality as a State Property

DSM uses finality in a specific sense.

It does not mean:

> Every machine in the world has observed the transition.

It means:

> The consumed committed resource cannot produce another conflicting realized successor inside the same valid state lineage.

Suppose:

$$K = \kappa_{\text{res}}(s, x).$$

After realization:

$$K \in \Sigma_{s'}.$$

A second consumption requires:

$$K \notin \Sigma_{s'}.$$

That fails.
Therefore the exclusion follows from the state itself.

## 72 Bitcoin Confirmation Versus DSM Finality

**Bitcoin Comparison**

Bitcoin confirmation confidence increases as additional proof-of-work blocks extend the chain containing a transaction.
DSM does not use confirmation depth as the core validity mechanism.
Its question is:

$$\text{Accept}(s, s', w) = \text{True?}$$

and:

$$K \notin \Sigma_s?$$

**Architectural consequence:**
A resource's single-consumption property does not become mathematically stronger merely because unrelated future blocks are produced.
Once the canonical DSM successor consumes the resource, a conflicting second consumption is non-derivable under the same realized lineage.

## 73 End-to-End Worked Example

Suppose Alice's device $A$ has a bilateral relationship with Bob's device $B$.

Current relationship head:

$$h_{AB,n}.$$

Relationship key:

$$k_{AB}.$$

Current relationship mapping:

$$R_n(k_{AB}) = h_{AB,n}.$$

Current consumed set:

$$\Sigma_n.$$

Current core root:

$$\rho_{\text{core},n} = \text{SMT}(R_n, u_n, \Sigma_n, \Pi_n, \Omega_n).$$

Current future candidates:

$$P_n.$$

Suppose candidate:

$$c_B = (s_{n+1}, b_B, g_B, K_B, d_B).$$

The candidate digest is:

$$d_B = H(\mathrm{enc}(s_{n+1})).$$

The relationship-parent descriptor is:

$$x_{AB,n} = (\texttt{relationship}, k_{AB}, h_{AB,n}).$$

The resource-consumption key is:

$$K_{AB,n} = H(\texttt{DSM/consume resource/v1} \parallel \rho_{\mathrm{core},n} \parallel x_{AB,n}).$$

The current state must prove:

$$K_{AB,n} \notin \Sigma_n.$$

The guard must verify:

$$g_B(s_n, w) = \mathrm{True}.$$

The relationship advances:

$$h_{AB,n} \to h_{AB,n+1}.$$

Thus:

$$R_{n+1}(k_{AB}) = h_{AB,n+1}.$$

The resource is consumed:

$$\Sigma_{n+1} = \Sigma_n \cup \{K_{AB,n}\}.$$

The next core root is:

$$\rho_{\mathrm{core},n+1} = \mathrm{SMT}(R_{n+1}, u_{n+1}, \Sigma_{n+1}, \Pi_{n+1}, \Omega_{n+1}).$$

The next state commits its future space:

$$P_{n+1}, \qquad \Gamma_{n+1}.$$

Then:

$$\rho_{n+1} = H(\rho_{\mathrm{core},n+1} \parallel \mathrm{digest}(P_{n+1}) \parallel \mathrm{digest}(\Gamma_{n+1})).$$

The result is:

$$\boxed{s_n \to s_{n+1}.}$$

The old relationship parent is consumed.
The new relationship head becomes current.
The new root becomes canonical for that lineage.
The next candidate future space is already cryptographically bound to the new state.

This is the core realization pipeline, common to every mode. Online acceptance additionally applies the register step of Section 11: before the candidate is evaluated, the sender's cell at the next position must be empty or hold exactly the presented root.

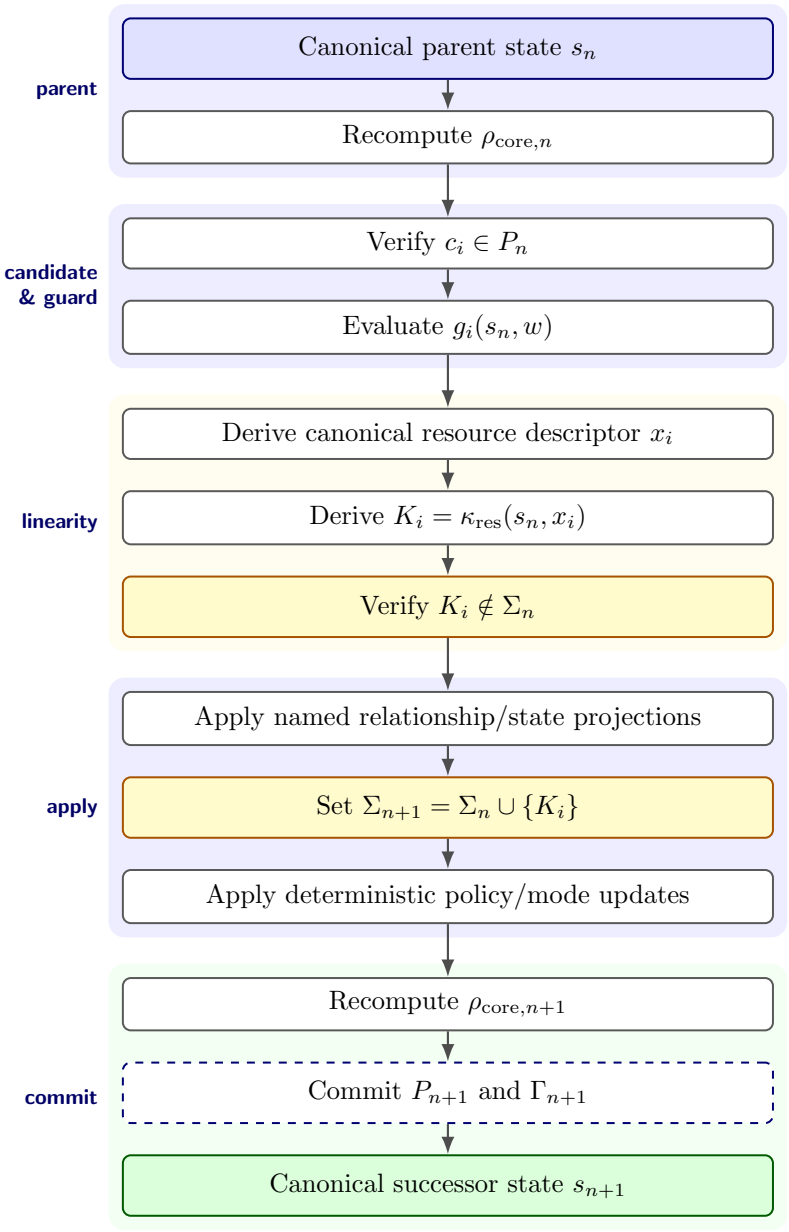



**Figure 24:** Core DSM Realization Pipeline

# Part V
# Assessment

Why every piece is necessary, what is assumed, what is proved, and the whole system in one picture.

## 74 Why Every Piece Is Necessary

No individual DSM mechanism is sufficient by itself.

### Hash chaining alone

Hash chains establish ancestry.

They do not by themselves prevent two successors from referencing the same parent.

### SMTs alone

SMTs authenticate state.

They do not define which transitions are legal.

### Precommitment alone

Precommitment constrains possible futures.

It does not by itself stop two conflicting committed futures from both trying to realize.

### Guards alone

Guards determine fulfillment.

They do not provide the strongest exclusion rule if multiple guards can be satisfied.

### Resource keys alone

Resource keys work only if conflicting branches are forced to derive the same key.

### Consumed state alone

The consumed set is useful only if it is cryptographically authenticated and carried forward monotonically.

Therefore:

$$
\begin{array}{|l|}
\hline
\quad \text{canonical encoding} \\
+\ \text{determinism} \\
+\ \text{bilateral hash chains} \\
+\ \text{SMT authentication} \\
+\ \text{precommitment} \\
+\ \text{guards} \\
+\ \text{canonical resource derivation} \\
+\ \text{shared consumption keys} \\
+\ \text{monotonic consumed state} \\
\\
\quad \Longrightarrow \text{guarded-linear-kernel realized-history safety.} \\
\hline
\end{array}
$$

That conclusion is the kernel's: within one valid realized lineage, no linear resource is consumed twice. The full online story of Part I, in which two counterparties who never speak must both be protected from a stale root, requires one more piece on top of the kernel: authenticated position binding through the write-once economic-root register (Sections 8 and 10). The kernel makes the second consumption invalid; the register makes the stale parent undeniable to an independent recipient.

## 75 Architecture Comparison

| Property | Bitcoin | DSM |
|---|---|---|
| Primary model | UTXO monetary state derived from a globally accepted proof-of-work blockchain | General guarded linear state derived from authenticated state transitions |
| Global ordering | Fundamental to accepted chain history | Not part of the state-validity mechanism |
| State progression | Blocks and transactions advance a shared public history | Canonical state generations advance by deterministic derivation |
| Pairwise state | Not fundamentally organized as bilateral chains | Explicit bilateral forward-only relationship chains |
| Merkle role | Transaction inclusion commitment in blocks | Authenticated current-state relationships, resources, and other state components |
| Future state | Spending conditions constrain valid spends | Exact candidate successor spaces may be precommitted |
| Conditional execution | Script/witness conditions | Deterministic guard families over candidate branches |
| Single consumption | UTXO outputs | General linear state resources |
| Conflict handling | Accepted blockchain history determines the surviving spend | Conflicting candidates share a resource key that can be consumed once |
| Finality | Proof-of-work confirmation depth and accepted chain | Conflict-local non-derivability after canonical resource consumption |
| Independent state | Still ultimately represented relative to one blockchain history | Independent bilateral/resource state can advance without universal ordering |
| Availability | Peer-to-peer block/transaction propagation | Storage and routing are separated from state authority |
| Offline bearer model | Offline signing possible; ordinary settlement requires later chain reconciliation | Optional authenticated offline origin plus software linearity and physical-device evidence |

## 76 What DSM Gains by Eliminating Global Ordering

DSM's architectural advantage is not merely speed.

It changes what must be coordinated.

Suppose there are one million independent state relationships.

If none shares a resource with another, then their relative ordering is not a state-validity question.

DSM permits them to remain mathematically independent.

The governing rule is:

| dependencies are explicit state relationships, not a global ordering requirement. |
|---|

Where ordering is semantically required, DSM can encode that ordering as state.
Where ordering is not semantically required, DSM does not invent one.
Therefore the validity architecture remains deterministic without a global sequencing layer.

### What this looks like at scale: a game

Take the most demanding consumer economy there is: a game with a hundred million players, hundreds of millions of items, and a match running at sixty frames a second.

The blockchain answer to "put this economy on-chain" has been tried, on Ethereum first and then on faster and object-oriented chains. It runs into the same wall each time: the chain is a shared execution environment, so every economic action becomes work for the whole network. Faster chains raise the ceiling. None removes it, because each still asks a shared network to order and validate every state change.

DSM's proposition is the inverse. Do not globally execute what does not need global agreement.

The game keeps running on the studio's servers and the players' devices, at game speed. Movement, physics, shooting, matchmaking are not economic events and never touch DSM. Underneath, the things that actually represent scarce value, a rare skin, a currency, a crafting input, a tournament prize, are CPTA assets with committed policies in the players' own device state. When Alice gives Bob the skin, Bob verifies its lineage and policy himself, at acceptance, against Alice's economic root. Nobody else ran anything.

| not the game on a chain; sovereign ownership underneath the game. |
|---|

This is why DSM is horizontally scalable in the ordinary sense of the term. Adding a player adds that player's device and its relationships. It does not add that player's actions to every other participant's workload, because there is no every-other-participant role. A hundred million players are a hundred million concurrent verifiers of their own state, and the only shared surfaces are the storage set's write-once cells, which are per-device, per-position, and content-blind.

The caveat belongs in the same paragraph. Horizontal scalability is a property of the architecture: no global execution, no global ordering, per-device economic roots. How close the implementation gets to it at a hundred million devices is an empirical question about verification cost, storage throughput, anti-equivocation under load, and recovery. Those are benchmarks to run, not theorems, and this document does not claim them.

## 77 What Bitcoin Optimizes For

Bitcoin's design solves a different primary problem.

It provides one highly replicated, proof-of-work-secured monetary history.

That makes the question:

> What is the accepted Bitcoin chain?

fundamental to its operation.

DSM does not attempt to reproduce a global ledger under a different name.

It changes the state model itself.

The core question becomes:

> Is this successor deterministically derivable from this committed parent, under the exact guards, resources, policies, and proofs already bound to that state?

## 78 Security Assumptions

DSM's guarantees depend on explicit assumptions.

These include:

1. cryptographic hash collision resistance;
2. signature unforgeability;
3. canonical serialization correctness;
4. correct relationship-key derivation;
5. correct resource-descriptor derivation;
6. conflicting branches deriving the same authoritative resource key;
7. correct SMT proof verification;
8. correct monotonic consumed-state updates;
9. faithful implementation of the deterministic transition predicate;
10. for offline physical claims, security of the relevant fused hardware and measurement boundary;
11. **durable register non-equivocation:** a register member that has acknowledged value $v$ for key $k$ does not subsequently acknowledge $v' \neq v$ for the same key, including after restart, restoration, or storage migration;
12. **threshold intersection:** for every owner-committed member set $S$ and threshold rule $q$, two independently constructed acceptance certificates for conflicting values cannot both satisfy the rule unless some member violates the previous assumption;
13. **member identity authentication:** a client counting distinct responses toward $q$ can authenticate that each response came from the member identity named by the committed storage-set catalogue.

Failure to reach the required threshold is a liveness failure. Violation of durable non-equivocation or of the intersection property is a safety failure.

The concrete primitives behind the first two assumptions are post-quantum: BLAKE3 for hashing, SPHINCS$^+$ for signatures, and Kyber for key encapsulation. The safety theorem itself is scheme-agnostic; any collision-resistant hash and unforgeable signature will do, and the choice of post-quantum primitives means the assumptions do not weaken when large quantum computers arrive.

No cryptographic protocol eliminates assumptions.

The important requirement is that the safety theorem follows from clearly stated assumptions rather than from informal trust.

## 79 Formal Verification

The safety theorems in this document are not only stated. They are supported by machine-checked artifacts, and the scope of those artifacts is stated precisely so that the reader can see what is covered and what is not.

### What has been proved

**Lean 4.** The key-scoped uniqueness theorem and Tripwire are proved over an abstract guarded model:

- an abstract type of states and an abstract type of resource keys;
- a step relation scoped by consumed keys;
- a history fold that threads the consumed set forward;
- the theorem that no valid realized history contains a key-scoped realized fork.

The uniqueness and Tripwire core depends on no axioms. The desired result is not assumed anywhere and then rediscovered; it follows from resource-key uniqueness, consumed-set threading, and the absence check before each accepted step.

**TLA$^+$.** A companion model checks the same statements on concrete guard families, in both the per-state form and the realized-history form of Section 7. Three checks are worth naming:

- a well-formed family, in which one conflict class resolves to a single successor and a disjoint branch consumes a second key, satisfies both safety and realized-history uniqueness;
- a deliberately malformed family, in which conflicting branches are given *different* keys for a shared resource, is shown to *violate* uniqueness. This is the falsification that confirms guard-family well-formedness is load-bearing rather than decorative;
- a relationship-scoped model, in which keys embed the relationship identity and each relationship has one acceptance locus, confirms that same-parent multi-receiver forks are unconstructible in online DSM.

### What the artifacts do not cover

The TLA$^+$ module treats resource keys as given inputs and guard well-formedness as an opaque flag. It exercises the consumed-set threading and the linearity layer. It does not exercise resource-key *derivation* or descriptor injectivity; those are discharged by the derived-keys-only rule and the implementation enforcement skeleton of the formal paper, not by the model checker.

The Lean development does not attempt to prove hardware behaviour, firmware correctness, secure-element behaviour, or transport liveness. Those appear in this document as assumptions or implementation obligations.

No offline-specific uniqueness artifact is required: offline steps are accepted through the same realized-history fold the artifacts already check.

The economic-root register and the client-side threshold layer are not covered by either artifact. They are an implementation and fault-model obligation, stated as assumptions in Section 78, not a theorem of the kernel.

### The standing caveat

$$\text{proved model property} \neq \text{automatically proved implementation.}$$

A production implementation must faithfully refine the abstract model. The artifacts prove that the model is safe; conformance testing and the enforcement skeleton are what tie the code to the model.

**Bitcoin Comparison**

Bitcoin's safety argument is empirical and economic: the chain has held under adversarial conditions for a long time, and the cost of rewriting it is public.
DSM's core safety argument is a proof over an abstract model, with the model's boundaries written down.
**Architectural consequence:**
The claim "two conflicting consumptions cannot both be realized" is a theorem with a machine-checked proof, not a track record. A reader can inspect exactly which assumptions it rests on.
**Tradeoff / boundary:**
A proof about a model says nothing about a bug in the code that implements it. Bitcoin's decade of adversarial uptime is evidence of a kind no proof supplies. The two arguments are different in nature, and the honest position is to want both.

## 80 One Mathematical Picture of DSM

Start with:

$$s_n = (R_n, u_n, P_n, \Gamma_n, \Sigma_n, \Pi_n, \Omega_n, \rho_n).$$

Compute:

$$\rho_{\text{core},n} = \text{SMT}(R_n, u_n, \Sigma_n, \Pi_n, \Omega_n).$$

Select candidate:

$$c_i \in P_n.$$

Verify guard:

$$g_i(s_n, w) = \text{True}.$$

Derive resource descriptor:

$$x_i.$$

Derive resource key:

$$K_i = \kappa_{\text{res}}(s_n, x_i).$$

Verify:

$$K_i \notin \Sigma_n.$$

Apply the named state transformation:

$$R_n \to R_{n+1}.$$

Consume the resource:

$$\Sigma_{n+1} = \Sigma_n \cup \{K_i\}.$$

Update:

$$u_n \to u_{n+1},$$

$$\Pi_n \to \Pi_{n+1},$$

$$\Omega_n \to \Omega_{n+1}.$$

Recompute:

$$\rho_{\mathrm{core},n+1}.$$

Commit:

$$P_{n+1}, \qquad \Gamma_{n+1}.$$

Then:

$$\rho_{n+1} = H(\rho_{\mathrm{core},n+1} \parallel \mathrm{digest}(P_{n+1}) \parallel \mathrm{digest}(\Gamma_{n+1})).$$

Thus:

$$\boxed{s_n \to s_{n+1}.}$$

And because:

$$K_i \in \Sigma_{n+1},$$

the consumed parent cannot be used again as a second conflicting consumption inside that valid lineage.

## 81 Intuitive Analogy

Imagine a sealed mathematical workbook.

Each page contains:

1. the current state;
2. a fingerprint of the previous state;
3. the current bilateral relationship heads;
4. a compact authenticated tree root;
5. a list of one-use resources already crossed out;
6. the exact possible next pages;
7. the conditions that would permit each next page;
8. deterministic policy.

A valid next page must have been listed in advance.

It must satisfy the correct condition.

It must consume a resource that has not already been crossed out.

When that resource is used, the next page permanently commits the fact that it was consumed.

The new page then commits its own possible successors.

Two unrelated sections of the workbook may progress independently.

But two pages that claim to consume the same one-use resource cannot both belong to one valid derivation.

That is the intuitive DSM construction.

## 82 Final Summary

DSM is built from several mechanisms that reinforce one another.

### 1. Canonical deterministic state

$$x \to \mathrm{enc}(x) \to H(\mathrm{enc}(x)).$$

The same logical state must produce the same bytes and commitment.

### 2. Bilateral relationship chains

$$h_{r,0} \to h_{r,1} \to h_{r,2}.$$

Relationships progress independently.

### 3. Sparse Merkle authentication

$$k_r \mapsto h_r$$

is committed into a compact authenticated root.

### 4. Canonical state chaining

$$s_n \to s_{n+1}$$

means the successor is deterministically derivable from its parent.

### 5. Precommitment

$$P(s) = \{c_1, \ldots, c_n\}.$$

The future state space is constrained before realization.

### 6. Guards

$$g_i(s, w) = \mathrm{True}$$

determines whether one candidate's fulfillment condition has been met.

### 7. Linear resources

$$K = \kappa_{\mathrm{res}}(s, x).$$

All conflicting branches consuming resource $x$ derive the same authoritative key.

### 8. Monotonic consumption

$$K \notin \Sigma_s$$

before realization and:

$$K \in \Sigma_{s'}$$

after realization.

### 9. Conflict-local uniqueness

Two conflicting branches cannot both consume the same linear resource in one valid realized history.

### 10. No global ordering requirement

State validity is determined from explicit committed dependencies.

A universal transaction ordering layer is not required.

If an application requires an ordering relation, that relation can be encoded as deterministic state rather than supplied by global consensus.

## 83 The Core DSM Statement

The entire construction can be summarized as:

canonical bytes
+ deterministic verification
+ bilateral state chains
+ Sparse Merkle commitments
+ canonical state roots
+ precommitted candidate futures
+ deterministic guards
+ linear-resource descriptors
+ shared consumption keys
+ monotonic consumed state

$\Longrightarrow$ conflict-local realized-state uniqueness without global ordering.

Bitcoin solves double spending by placing monetary transactions into a globally accepted proof-of-work history and deriving the valid UTXO state from that history.

DSM removes global ordering from the validity architecture entirely.

It does not replace one global ordering system with a smaller global ordering system. Applications with genuinely shared public resources may use scoped write-once binding evidence; that mechanism establishes exclusivity for the named resource without constructing a universal transaction order.

It replaces ordering authority with explicit authenticated state constraints.

The verifier asks:

Is this successor mathematically derivable from the committed parent?

The decisive facts are already present in or cryptographically bound to the state:

- the current relationship heads;
- the canonical state root;
- the candidate successor set;
- the deterministic guard family;
- the exact linear resources involved;
- the consumed-resource state;
- the deterministic policy;
- the required signatures and proofs.

Many futures may be precommitted.
Many independent relationships may progress concurrently.
Many machines may store and relay the data.
But for one committed linear resource:

**one valid realized lineage cannot contain**
**two conflicting consumptions of the same resource.**

That is the central mathematical idea of DSM.