Deterministic State Machines as Guarded Linear Constraint Systems: *double spend prevention as a state property*

Brandon Ramsay
Irrefutable Labs Inc.

June 2026, revised July 2026

Abstract

This paper formalizes the Deterministic State Machine, or DSM, as a guarded, linear, forward only, constraint based computation model. DSM does not require a blockchain, validator set, sequencer, gas market, mempool, clock based ordering rule, or global consensus layer for ordinary state evolution. State validity is determined by local verification of cryptographically precommitted candidate futures, deterministic fulfillment guards, explicit resource consumption keys, canonical encodings, signatures, and Sparse Merkle Tree commitments.

The architectural shift is that finality moves from a network decision to a state property. Instead of requiring some external party, committee, sequencer, validator set, or global clock to agree that a transition happened, DSM asks whether the proposed transition consumes the committed state resource and advances it deterministically. If it does, the successor is locally realized. If it does not, it is not state. Finality is therefore not provided as a service by an ordering network; it is a consequence of the consumed resource itself. This is global realization without global ordering: local acceptance composes into a coherent realized history because incompatible histories cannot merge.

The central claim is not that DSM forbids all branching at the level of precommitment. DSM explicitly permits precommitment forking. Multiple candidate futures may be prepared for a common parent, including Deterministic Limbo Vault release paths, refund paths, hash fulfillment paths, recovery paths, abort paths, contingent payment paths, CPTA token operations, offline bearer steps, and other mutually exclusive branches. The invariant is:

multiple futures may be described, but for a fixed consumed resource, at most one may be realized.

A resource is the thing that produces exactly one realized successor. A relationship parent, spendable object, vault generation, token balance object, source vault generation, recovery generation, or offline anchor step is a resource in this sense. Candidate branches are only possible futures. Realization consumes the resource.

A generation is a logical version of a resource family. It is not a time period, not a clock interval, not a scheduling unit, and not an external ordering coordinate. Generations are ordered only by DSM derivability and resource consumption.

This paper makes explicit the machinery required for that statement. Guard exclusivity is not left as convention. A DSM precommit set is valid only if its guard family is well formed. A well formed guard family gives either a deterministic branch selector, a mechanically proven exclusive guard class, or a set of branch predicates that are structurally tied to the same consumed resource. Parent consumption is modeled through canonical resource consumption keys and a consumed parent set committed by the DSM root. Branch local keys may exist for indexing or audit, but the shared resource key is what enforces linear exclusion.

A hardened form of the model is also stated: for any conflict class, all conflicting candidates must derive the same canonical consumption key for every resource they share, and those keys

must be derived from the committed parent state rather than supplied as discretionary branch data. A branch may additionally consume resources disjoint from the conflict without weakening exclusion. This removes the implementation hazard where release, refund, recovery, or policy branches accidentally consume different keys for a shared resource, while still admitting branches such as contingent alternative payees that consume an extra disjoint resource.

The role of guard exclusivity is clarified. Pairwise guard exclusivity is useful and required for guard classes that claim exclusive fulfillment, but it is not the only safety mechanism. When conflicting branches share the same derived resource consumption key, linearity alone prevents co realization even if several guards are simultaneously fulfilled. Thus the load bearing safety rule is not merely that guards should not overlap. The load bearing safety rule is that conflicting futures must fight over the same derived resource.

Tripwire is formalized as the theorem that two conflicting realized successors from the same consumed resource cannot both be derivable. More precisely, conflicting candidates may be constructed or transmitted as bytes, but they cannot both validate under the DSM step predicate and cannot both go through as accepted state transitions for the same consumed resource. Forks are therefore not selected against by consensus. They are excluded by guarded realization and exposed by reconciliation if conflicting invalid bytes are presented outside the valid model.

Offline bearer mode is treated as an optional mode predicate, and this revision states its corrected form. Transfer uniqueness, including offline transfer uniqueness, is a software theorem of the guarded linear kernel. An SMT committed anchor counter makes the offline origin an explicit coordinate of the committed state, every offline candidate from that origin consumes the same derived anchor step resource key, and realized history uniqueness applies without a hardware term. Hardware is thereby demoted to the one job software cannot perform: uniqueness of the physical device instance. Every offline release carries a three factor identity witness, a seed rooted DSM signature, a PUF rooted non exportable chip signature, and a measurement gated partition sealed host signature over the same root advance message, and the physical monotonic counter survives only as a tracker of the SMT counter: a non rewind floor, a stale image tripwire, and an offline exposure cap. The receiver witnessed counter positioned commit of earlier drafts is superseded: a scalar counter read never binds the transition it brackets, and the binding was never needed.

The safety theorems of this paper are supported by machine checked artifacts. The general key scoped fork exclusion theorems are proved in Lean 4 over an abstract guarded model, with the uniqueness and Tripwire core depending on no axioms. A companion TLA$^+$ development model checks the same statements in both a per state form and a realized history form on concrete guard families, and by deliberate falsification of a malformed family confirms that guard family well formedness is load bearing. A relationship scoped model checks that same parent multi receiver forks are unconstructible in online DSM. Appendices A and B point to these artifacts. Their exact scope and boundary are stated in the Claim Boundary section.

Contents

1 Introduction **4**

2 Scope and Design Boundary **5**

3 Related Work and Novelty Boundary **6**

3.1 Detection Based Offline Electronic Cash 6

3.2 Observer and Tamper Resistant Hardware as Transaction Authority 6

3.3 Consensus and Off Chain Channels 7

3.4 Novelty Boundary 7

4 Adversarial and Cryptographic Model **8**

5 Core Intuition **9**

6 Logical Generations **10**

7 State Model **10**

8 Relationship Projections **11**

9 Morphism Detection **12**

10 Precommitment Forking **12**

11 Guard Algebra **13**

12 Admissible Guard Classes **16**

13 Consumption Keys and Parent Linearity **18**

14 Hardened Resource Key Discipline **20**

15 Layered Realization Predicate **21**

16 Policy Discipline **23**

17 Conflict Local Realization Uniqueness **23**

18 Deterministic Limbo Vaults as Guarded Branch Systems **25**

19 CPTA as a Guarded Policy System **26**

20 Multi Party Authority as Guarded Bilateral Composition **28**

21 Tripwire **28**

22 Reachability and Canonical Realized History **30**

23 Linearity and Forward Progress **30**

24 Concurrency and Message Delivery **31**

25 Linear Resource Interpretation **32**

26 Cut Elimination Interpretation **32**

27 Category Theoretic Formulation **33**

28 Denotational Semantics **34**

29 Operational Semantics **34**

30 Offline Bearer Mode: Software Uniqueness, Hardware Identity **35**
30.1 Identity Domains . . . 36
30.2 The SMT State Counter . . . 36
30.3 Offline Uniqueness Is the Kernel Theorem . . . 37
30.4 Three Factor Release Witness . . . 38

31 Storage Nodes and Non Authority **40**

32 Claim Boundary **40**
32.1 Machine Checked Scope . . . 41

33 Architectural Consequences **41**
33.1 No Global Ordering Requirement . . . 41
33.2 No Replace By Fee at Protocol Level . . . 41
33.3 No Protocol Level Global Ordering MEV Surface . . . 41
33.4 No Consensus Inclusion Fee . . . 42
33.5 No Consensus Capture Attack Class . . . 42
33.6 No Protocol Level Censorship Gate . . . 42

34 Structural Privacy **42**

35 Comparison to Blockchain Systems **42**

36 Mapping Summary **43**

37 Main Result **43**

38 Conclusion **44**

A Lean 4 Development **44**

B TLA$^+$ Development **45**

C Offline Bearer Interface Skeleton **45**

D Implementation Enforcement Skeleton **47**

E Terminology **48**

F References **50**

1 Introduction

Bitcoin begins from the goal of allowing online payments to be sent directly from one party to another without passing through a financial institution. The usual blockchain solution removes a financial intermediary but replaces it with a global coordination layer: miners, validators, sequencers, mempools, fee markets, external ordering coordinates, block ordering, and consensus finality.

DSM takes a different route. It does not attempt to improve global ordering. It removes the need for global branch selection by enforcing a stronger local invariant:

from any consumed parent resource, at most one precommitted branch may be realized.

This formulation is important. DSM does allow several possible futures to be precommitted. For example, a Deterministic Limbo Vault may lay out release, refund, abort, and recovery branches. A CPTA token policy may lay out transfer, emission, mint, burn, claim, and recovery branches. A recovery policy may lay out tombstone, cancel, succession, and failure branches. These branches are not executed merely because they are described. They become live only when their deterministic fulfillment guard is satisfied and the required resource consumption key is still unconsumed.

The key architectural distinction is:

blockchains resolve competing executed futures after the fact,

whereas DSM enforces:

only one precommitted future may become validated and realized for the same consumed resource.

This distinction drives the rest of the model. If there are no competing valid realized futures for the same consumed resource, then there is no need for a global branch selection authority for that resource. There may still be local refusal, application policy, routing delay, storage unavailability, or failure to satisfy a guard. Those are local validity or availability problems.

DSM does not require a single global serial order of all independent actions. Two unrelated relationships may advance independently. The consensus free claim is therefore not global singleton execution. It is conflict local fork exclusion.

2 Scope and Design Boundary

DSM has three layers.

(1) **Core DSM.** Core DSM is the guarded linear state system. It uses canonical encodings, hash adjacency, relationship projections, precommitted candidate futures, deterministic guard families, explicit resource consumption keys, Sparse Merkle Tree commitments, and signatures.

(2) **Online DSM.** Online DSM uses local receipt exchange and object delivery. It does not require a hardware anchor. A received object is accepted if it is adjacent to the receiver's local frontier and the proof material verifies. Otherwise it is queued or rejected.

(3) **Offline bearer DSM.** Offline bearer mode is optional. It is used when a receiver accepts without online reconciliation at the moment of exchange. It adds a hardware identity layer for physical duplication resistance: a three factor fused device identity whose witnesses accompany each offline release. It adds nothing to transfer uniqueness, which remains a software property of the kernel, and it is not a general trust requirement of DSM.

Storage nodes are outside the authority path. They are byte mirrors and routing infrastructure. A storage node may deliver arbitrary bytes. Devices verify all acceptance predicates locally. Storage node correctness is obtained from deterministic placement, mirrored byte commitments, audits, client side proof verification, and economic admission rules. Storage nodes are not signers of DSM state transitions and are not hardware attestation authorities.

Earlier DSM drafts used DBRW terminology for anti clone experiments. This paper supersedes that mechanism for authority path claims. DBRW is not part of the core transition predicate, not part of storage node authority, and not a required anti clone primitive. Offline bearer anti clone claims are scoped to the optional hardware identity construction. Online DSM remains non blocking and does not require a hardware anchor. This paper likewise supersedes the receiver witnessed counter positioned commit of earlier offline bearer drafts. Section 30 states the corrected model, in which offline uniqueness is an instance of the kernel's realized history uniqueness and hardware evidence is scoped to device identity.

3 Related Work and Novelty Boundary

DSM transfers value between mutually distrustful parties, including offline, without a global ledger. Three bodies of prior work address that problem, and DSM sits against all three. This section states where DSM overlaps with each and where the boundary is, so that the contribution is not mistaken for a repackaging of known offline cash machinery.

3.1 Detection Based Offline Electronic Cash

The classical line begins with Chaum's blind signatures (1982), which make a withdrawn coin unlinkable to its later spending. Chaum, Fiat, and Naor (1988) extended this to the offline setting, where a payee accepts a coin without contacting the issuer at the moment of payment. Their answer to offline double spending is detection rather than prevention: a coin spent once stays anonymous, but spending it twice reveals enough to identify the spender, and the fraud surfaces later, when the coins are deposited at the bank. Brands (1993) gave an efficient construction of the same paradigm using restrictive blind signatures, and a large subsequent literature refines it. Recent offline central bank digital currency proposals continue this line. Kempen and Pouwelse (2024), for example, build an offline first digital euro with zero knowledge transaction privacy and retroactive double spending detection with anonymity revocation at deposit.

The defining property of this entire family is that offline double spending is not prevented at the point of exchange. It is permitted, then detected and punished afterward through identity revelation once the tokens reach a settlement authority. Anonymity is the primary goal, and the double spend deterrent is an after the fact consequence of the anonymity construction. DSM differs in what it optimizes and in where exclusion happens. DSM does not prevent double spending by revoking anonymity at a bank; it excludes co realization at acceptance, as a property of the consumed state, with no deposit step and no issuer on the common path. DSM does share one structural feature with this family, and the paper is explicit about it: against a full compromise adversary who originates conflicting proposals to disconnected first contact receivers, DSM falls back to exposure and non mergeability at reconciliation (Theorem 5), which is philosophically adjacent to double spend detection. The difference is that this fallback is the boundary case, not the mechanism. The mechanism is prevention in the state predicate for every receiver holding a frontier.

3.2 Observer and Tamper Resistant Hardware as Transaction Authority

A parallel line makes hardware the double spend authority. Chaum and Pedersen (1992) introduced the wallet with observer: a tamper resistant chip embedded in the user's wallet co signs each payment and refuses to sign the same value twice, so that double spending is prevented rather than merely detected, while the user's own device preserves privacy. Brands (1993) also situates his cash

in the wallet with observer model. Fielded smartcard purses of the same era, and a substantial share of recent offline CBDC and secure element payment designs, follow this template: value and the no double spend rule live inside tamper resistant hardware, and breaking that hardware breaks the scheme.

This is the design DSM most directly inverts, and the inversion is the central claim of the companion specification. In the observer model the hardware is the transaction authority. It is trusted to enforce uniqueness, so the protocol inherits the hardware's full attack surface as its own. DSM refuses that assignment. Transfer uniqueness is a software theorem of the guarded linear kernel (Theorem 10, Corollary 4), and it holds with the anchor hardware deleted. Hardware is demoted to one job it alone can do, which the observer literature never isolates as a separate concern: proving that the physical device is the enrolled device, so that a software clone holding every host readable byte cannot pass as the original. The offline release therefore carries a three factor identity witness, and the measurement gated host seal makes the single appliance emission property strict: under enrolled firmware the appliance cannot mint two first contact releases from one origin, and firmware that fails the enrolled measurement cannot invoke the host signer at all. Hardware witnesses identity. It never decides whether a transfer is unique.

3.3 Consensus and Off Chain Channels

Blockchain systems (Nakamoto, 2008) prevent double spending by imposing a single global order on transactions through consensus, and resolve conflicting spends by which one the ordered history includes. Payment and state channels, such as the Lightning Network (Poon and Dryja, 2016), move the common case off chain by exchanging signed channel states directly between two parties, but they still root safety in the consensus layer: disputes are settled by broadcasting the latest signed state on chain within a timeout, with penalty transactions and watchtower style monitoring guarding against a counterparty publishing a stale state. The security argument reduces to the global ledger for dispute resolution.

DSM has neither a global order nor an on chain court of last resort for ordinary operation. There is no mempool, no sequencer, no validator set, and no settlement broadcast on the common path. Two unrelated relationships never contend for a global slot; they advance independently. Conflicting successors of one consumed resource are excluded locally by resource linearity, not selected among by an ordering authority, as the section on conflict local realization uniqueness (Section 17) and the later comparison to blockchain systems make precise. Unlike a channel, DSM does not assume a live counterparty or a timely dispute window. A stalled or hostile counterparty is a liveness or availability failure, and the safety property does not depend on anyone broadcasting anything within a deadline.

3.4 Novelty Boundary

The boundary can be stated directly.

- Prior offline electronic cash handles double spending either through tamper resistant hardware acting as the transaction authority, or through spending limits, or through after the fact detection and anonymity revocation at deposit.
- Blockchain systems handle double spending through global ordering and consensus, and channels inherit that consensus as their dispute layer.
- DSM moves double spend exclusion into the state predicate itself. A valid transition consumes a committed resource, and two incompatible successors of that resource cannot both become

one coherent realized history. This holds online and offline, and it holds with no hardware term.

- Hardware in DSM is not the transaction authority. It protects offline device identity and prevents physical duplication of the bearer appliance, nothing more. The measurement gated host seal strengthens this: under enrolled firmware a single appliance cannot emit two first contact releases from one origin, and rogue firmware cannot call the host signer.

What is genuinely new is the placement of uniqueness and the demotion of hardware, proved rather than assumed. Uniqueness is neither a consensus outcome nor a hardware guarantee but a linear resource property of the committed state, and hardware is reduced to an identity witness whose failure can forge a device but not a transfer. The formal framing as a guarded linear constraint system, in which precommitment admits additive choice among candidate futures while realization forbids contraction of the consumed resource, is the tool that makes the claim checkable rather than rhetorical, and it is the second contribution of this paper.

4 Adversarial and Cryptographic Model

Definition 1 (Adversarial Delivery)**.** The adversary may delay, reorder, replay, drop, duplicate, or corrupt messages. The adversary may control storage nodes, present stale objects, fork local databases, and attempt to make different receivers accept conflicting packages. The adversary may observe public bytes and all data intentionally disclosed to a counterparty.

Definition 2 (Software Clone)**.** A software clone receives all host readable wallet state, including local databases, cached proofs, chain history, public enrollment data, transport state, and user space keys that are not protected by the current authority mode. A software clone does not receive non exportable state of an optional offline anchor.

Definition 3 (Perfect Live State Clone)**.** A perfect live state clone extracts or emulates the exact current non exportable state required by the current authority mode, without divergence. No offline only protocol can distinguish a perfect exact emulation from the original device. This is the boundary of offline physical duplication resistance.

Assumption 1 (Hash Soundness)**.** The hash function used by DSM is collision resistant and second preimage resistant over canonical encodings and domain separated inputs.

Assumption 2 (Signature Soundness)**.** The signature schemes used by DSM are secure against chosen message forgery for their stated parameter sets and domain separated messages.

Assumption 3 (Canonical Encoding Injectivity)**.** Canonical encodings are injective over well formed DSM objects. Two distinct well formed objects cannot have the same canonical byte encoding.

Assumption 4 (Resource Descriptor Canonicalization)**.** Every linear resource committed by a well formed DSM state has a canonical descriptor that is derivable from committed parent state and protocol fields and is not discretionary branch input. The descriptor map is injective over the linear resources committed by a state: two distinct resources have distinct canonical descriptors, and equal descriptors denote the same resource. Consequently the derived resource consumption key of Definition 28 is a function of the resource alone at a fixed parent state, and distinct resources yield distinct keys. This is the resource level analogue of Canonical Encoding Injectivity. It is a standing assumption of the model, and it is what guarantees that conflicting branches, which by the Conflict Relation contend for a common resource, contend for a common derived key.

Assumption 5 (Optional Offline Identity Evidence)**.** When offline bearer mode is used, the offline identity witnesses cannot be forged except by breaking the stated assumptions. The chip witness key is generated in slot, is non exportable, and is protected at rest by the die's physically unclonable function. The host witness key is generated inside the secure partition and sealed to it, and the host signing service is measurement gated: it is available only when the running firmware measurement equals the enrolled measurement, so firmware not matching that measurement cannot invoke it and receives a hard unsealing failure with no fallback signing path. The chip and host witnesses for a transfer are minted only during commit, after the one physical counter decrement that consumes the offline origin, so no valid witness for an origin exists while that origin is still spendable, and neither witness is exported before commit. Both witness signature schemes are secure against chosen message forgery. The physical monotonic counter cannot be rewound below its enrolled bound. A measurement boundary breach, meaning firmware not matching the enrolled measurement that nevertheless invokes host signing or extracts a minted witness before its committing counter step, is outside this assumption and is grouped with the perfect live state clone boundary. Extraction and perfect emulation of all live non exportable factors is the stated boundary.

5 Core Intuition

DSM treats state as a collection of linear resources. A resource is the object that produces exactly one realized successor. It may expose a future space with several candidate branches, but it may be consumed only once.

The oversimplified model is:

$$s_i \multimap s_{i+1}.$$

The correct DSM model is:

$$s_i \multimap \{(g_1, s_1), (g_2, s_2), \ldots, (g_n, s_n)\},$$

with the conflict local realization invariant:

$$|\{s_j : g_j \text{ is fulfilled and } s_j \text{ consumes fixed resource } K\}| \leq 1.$$

The predecessor resource is not reusable. Candidate branches may be prepared, but once one branch is fulfilled and ratified, all incompatible candidates are dead with respect to that consumed resource.

The fundamental rule is therefore not:

there is only one possible branch,

and not:

there is only one possible global next transition.

The correct rule is:

for a fixed consumed resource, there is only one realizable branch.

A system that permits multiple executed branches for the same consumed resource must eventually decide which branch survives. That is the function of consensus. DSM instead prevents co realized branching for the same consumed resource under its acceptance predicate.

6 Logical Generations

DSM has no clocks, no clock based finality, and no external time based validity rule. The word generation is used only for logical versioning of a resource family.

Definition 4 (Generation)**.** A generation is a logical version of a resource family, derived from committed DSM state and advanced only by valid DSM transitions. It is not a time period, not an elapsed interval, not a scheduled round, and not an external ordering coordinate.

Definition 5 (Generation Resource)**.** A generation resource is a linear resource associated with a particular logical generation of a vault, token policy, source vault, recovery controller, claim family, or other DSM resource family. A generation resource may expose several candidate futures, but at most one may validate and realize.

Remark 1. The generation terminology exists only to avoid ambiguity between repeated logical versions of the same resource family. It does not introduce time, clocks, deadlines, or external sequencing into DSM.

7 State Model

Let $\mathcal{R}$ be the finite set of relationship identifiers for a device.

Each relationship $r \in \mathcal{R}$ has a local forward only hash chain:

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

The parent in one relationship chain is not transferable to another relationship chain. A parent committed under relationship r cannot be replayed as a parent under relationship q, for $r \neq q$.

Definition 6 (DSM Global State)**.** A DSM global state is a tuple:

$$s = (R, u, P, \Gamma, \Sigma, \Pi, \Omega, \rho),$$

where:

- $R : \mathcal{R} \to \mathcal{H}$ maps each relationship to its current relationship chain head.
- u is a monotonic progression object, such as a local counter, sequence digest, receipt frontier, or mode specific anchor counter. In offline bearer mode the anchor counter is committed as an SMT leaf and consumed with the whole state, as specified in Section 30.
- P is the precommit object or precommit set fixing the candidate future space.
- Γ is the guard family associated with the precommitted branches.
- Σ is the consumed parent set, represented canonically, for example as a Sparse Merkle Tree of resource consumption keys.
- Π is policy and authority data.
- Ω is mode evidence. In ordinary online DSM, $\Omega = \bot$. In offline bearer mode, Ω contains the offline anchor state.
- ρ is the Sparse Merkle Tree root committing to all state components.

The canonical global commitment is layered so that guard descriptors, resource consumption keys, and candidate digests may bind the parent root without inducing a hash cycle. Define a core root over the components that do not depend on the candidate future space or the guard family:

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

Guard descriptors, the derived resource consumption keys of Definition 28, and the candidate digests of Definition 11 bind ρ_{core} rather than the full root. The full canonical global commitment is then:

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

Wherever a guard descriptor, resource consumption key, or candidate digest is said to bind the parent root or ρ_s in the remainder of the paper, the binding is to ρ_{core}. This makes the commitment a directed acyclic hash graph: the precommit set P and the guard family Γ commit to ρ_{core}, and ρ commits to P and Γ, so no field transitively hashes itself. An equivalent formulation treats precommitment as its own realized step, in which candidates bind the root of the state being extended and the family committed at step n constrains step $n + 1$; this matches the bilateral flow in which a counterparty co signs a precommit before the successor exists. Either formulation removes the cycle; the layered core root is used throughout.

Definition 7 (Well Formed DSM State)**.** A DSM state $s = (R, u, P, \Gamma, \Sigma, \Pi, \Omega, \rho)$ is well formed if:

(1) every component is canonically encoded;

(2) relationship identifiers are domain separated;

(3) relationship heads are valid chain tips;

(4) the candidate set P is finite or policy bounded;

(5) the guard family Γ is well formed;

(6) Σ is a valid canonical consumed parent set;

(7) Π is a deterministic policy object;

(8) Ω is either empty or valid for the selected mode;

(9) ρ_{core} recomputes byte exactly from $R, u, \Sigma, \Pi, \Omega$, and ρ recomputes byte exactly from ρ_{core}, digest(P), and digest(Γ).

8 Relationship Projections

For each relationship r, define a projection:

$$\pi_r : S \to S_r,$$

where S is the global DSM state space and S_r is the local relationship state space for relationship r.

For a global state $s = (R, u, P, \Gamma, \Sigma, \Pi, \Omega, \rho)$,

$$\pi_r(s) = R(r).$$

A realized global transition must commute with every relationship projection it touches. If

$$s_i \xrightarrow{f} s_{i+1}$$

is a valid realized global transition, and relationship r is named by the transition, then:

$$R_i(r) \xrightarrow{f_r} R_{i+1}(r)$$

is the corresponding local relationship transition, and:

$$\pi_r(s_{i+1}) = f_r(\pi_r(s_i)).$$

Unrelated relationship projections are preserved unless explicitly named by the transition.

Definition 8 (Projection Commutativity)**.** A transition $s \to s'$ satisfies projection commutativity if, for every relationship r:

$$\pi_r(s') = \begin{cases} f_r(\pi_r(s)) & \text{if } r \text{ is named by the transition,} \\ \pi_r(s) & \text{otherwise.} \end{cases}$$

9 Morphism Detection

DSM is naturally described as a morphism detector.

Definition 9 (Potential Morphism)**.** A potential DSM morphism is a precommitted candidate branch:

$$f_i : s \to s_i$$

paired with a guard descriptor g_i. The potential morphism is syntactically available because it is hash bound into the candidate future space.

Definition 10 (Realized Morphism)**.** A potential morphism $f_i : s \to s_i$ becomes a realized DSM morphism iff:

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

A candidate successor is not accepted because it was selected by an ordering authority. It is accepted because it is the fulfilled, structure preserving morphism out of the current state.

10 Precommitment Forking

DSM supports precommitment forking. This is the mechanism by which DSM expresses conditional workflows without a Turing complete contract virtual machine and without blockchain style conflict resolution after execution.

Definition 11 (Candidate Future)**.** Given a DSM state s, a candidate future is a tuple:

$$c_i = (s_i, b_i, g_i, K_i, d_i),$$

where:

- s_i is the candidate successor state;
- b_i is a branch identifier;
- g_i is the guard descriptor;
- K_i is the canonical resource consumption key set for the branch;
- $d_i = \mathrm{H}(\mathrm{enc}(s_i))$ is the candidate digest.

The candidate is admissibly described if it is canonically encoded, hash bound to the parent state s, included in the precommitted future set $P(s)$, and covered by the well formed guard family $\Gamma(s)$.

Definition 12 (Candidate Set)**.** The candidate set of a DSM state s is:

$$C(s) = \{s_i \mid \mathrm{CandidateOK}(s, s_i)\}.$$

The set $C(s)$ may contain more than one element.

Definition 13 (Candidate Fork)**.** A candidate fork from predecessor s is a pair of distinct candidate successors s_a, s_b such that:

$$s_a \neq s_b \quad \text{and} \quad s_a, s_b \in C(s).$$

Candidate forks are permitted.

11 Guard Algebra

The shorthand guard exclusivity is not sufficient as an informal assumption. DSM requires a precommit set to carry a well formed guard family. A candidate set that does not prove guard well formedness is invalid before any branch can realize.

Definition 14 (Guard Descriptor)**.** A guard descriptor g is a canonical object specifying:

(1) the parent state digest it is bound to;

(2) the branch identifier it may activate;

(3) the resource consumption key or keys;

(4) the accepted witness type;

(5) the deterministic verification predicate for a witness;

(6) the conflict class of the branch.

Definition 15 (Guard Witness)**.** A guard witness w is a canonical byte string, proof, preimage, receipt bundle, completion certificate, recovery certificate, refund certificate, or mode specific evidence object accepted by the guard verifier.

Definition 16 (Guard Family)**.** For a parent state s, a guard family Γ_s is a finite collection:

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

where b_i is a branch identifier, g_i is a guard descriptor, and K_i is the resource consumption key set for branch b_i.

Definition 17 (Conflict Relation)**.** Two candidate branches b_i, b_j conflict, written:

$$b_i \#_s b_j,$$

if they cannot both be part of the same realized history for the same linear resource from parent s. This holds when they consume the same relationship parent, the same spendable object, the same DLV generation, the same CPTA balance object, the same source vault generation, the same recovery generation, the same offline anchor step, or any overlapping linear resource committed by s.

Definition 18 (Conflict Class at a Key)**.** For a parent state s and a derived resource consumption key K, the conflict class at K, written $C_G(s)$ with $G = K$, is the set of candidate branches from s whose resource consumption key set contains K. Membership is the equivalence induced by K: two branches lie in the same class at K iff both consume K. The pairwise conflict relation $\#_s$ need not be transitive, since two branches may each share a distinct resource with a third while sharing none with each other; but conflict at a fixed key is an equivalence, and it is the keyed class that the uniqueness results below quantify over. By Assumption 4 the branches that contend for a given resource are exactly the branches that derive its key, so a keyed class is well defined.

Rule 1 (Guard Family Well Formedness)**.** A guard family Γ_s is well formed iff all of the following hold.

(G1) **Parent binding.** Every guard descriptor includes the canonical digest of parent state s, its branch identifier, and its resource consumption key set.

(G2) **Deterministic verification.** For any guard descriptor g and witness w, VerifyGuard(s, g, w) is deterministic.

(G3) **Branch binding.** A witness accepted for branch b_i cannot be reinterpreted as a witness for branch $b_j \neq b_i$ without changing a domain separated digest or signature message.

(G4) **Conflict coverage.** If $b_i \#_s b_j$, then either $K_i \cap K_j \neq \emptyset$, or the two branches are covered by a deterministic selector:

$$\text{Select}_s(w) \in \{b_1, \ldots, b_n, \bot\}$$

that returns at most one branch for the conflict class. This condition is the informal baseline: under (G7) together with Assumption 4 its first disjunct holds automatically, since conflicting branches share a resource and therefore a derived key, so $K_i \cap K_j \neq \emptyset$. It is retained for readability and need not be checked independently of (G7).

(G5) **Exclusive fulfillment.** For any guard class that claims pairwise exclusive fulfillment, and for any conflicting branches $b_i \#_s b_j$, there do not exist accepted witnesses w_i, w_j such that:

$$\text{VerifyGuard}(s, g_i, w_i) = 1 \wedge \text{VerifyGuard}(s, g_j, w_j) = 1.$$

This condition must be proven by the guard construction when the construction relies on exclusive fulfillment.

(G6) **Canonical precommit.** The digest of Γ_s is committed by the parent state root ρ_s. A verifier rejects any successor whose guard family differs from the committed family.

(G7) **Derived key agreement.** Every resource consumption key is derived from committed parent state and canonical resource descriptors under Rule 2 and Assumption 4, and is never supplied as branch data. For conflicting branches $b_i \#_s b_j$, the derived keys of every resource the two branches share coincide:

$$b_i, b_j \in G \Rightarrow K \in K_i \cap K_j \text{ for the key } K \text{ of each shared resource.}$$

A branch may additionally consume resources disjoint from the conflict without weakening exclusion, so full equality of the sets K_i and K_j is neither required nor correct. If a conflict class intentionally uses a deterministic selector, the selector itself must be committed by Γ_s, must output at most one branch, and each selected terminal branch must consume the selected resource key exactly once.

Lemma 1 (Conflicting Branches Share a Derived Key)**.** *Under Assumption 4 and Rule 2, if $b_i \#_s b_j$ then the two branches share a committed linear resource, and the derived key of that resource lies in $K_i \cap K_j$. Distinct resources yield distinct derived keys, so agreement between conflicting branches is required only on the resource they share, and each branch may consume additional disjoint resources.*

Proof. By the Conflict Relation, $b_i \#_s b_j$ holds because the two branches overlap on a linear resource committed by s. By Assumption 4 that resource has a unique canonical descriptor and therefore a single derived consumption key, which both branches consume; by Rule 2 a verifier treats that derived key as authoritative, so neither branch can substitute a different key for the shared resource. Injectivity of the descriptor map gives distinct keys for distinct resources, so no agreement is forced on resources the branches do not share. □

Remark 2 (G5 and G7 hierarchy)**.** G5 and G7 play different roles. G5 is the guard level exclusivity rule. It is useful for selector guards, DLV decision functions, recovery decision objects, and other classes that prove at most one branch can be fulfilled. G7 is the structural hardening rule. It is the load bearing fallback for safety when multiple guards can be fulfilled at the same time. Under Assumption 4 and Rule 2, G7 is not an independent axiom: conflicting branches share a committed resource, hence a derived key, so the sharing is derived rather than asserted (Lemma 1). What G7 adds as a discipline is only that keys are taken from the derivation and never from branch data. If conflicting branches share the derived key of the resource they contend for, then linearity prevents co realization even when several guards evaluate to true, and a branch may still consume additional disjoint resources, so contingent alternative payees and extra fee legs are admissible. Therefore DSM safety is not merely that guards do not overlap. DSM safety is that conflicting candidates cannot realize unless they consume the same one use resource, and that resource can be consumed only once.

The two rules also differ in where they exclude co realization, and this distinction is made precise in Section 17. A G5 class excludes co realization at the level of guard fulfillment: no two conflicting witnesses can both verify, so exclusion holds already at the static per state predicate. A class that relies on G7 alone permits two conflicting candidates to both satisfy the static per state predicate against a common unconsumed parent set; exclusion for such a class is realized when the successor is accepted and the consumed parent set is updated. Both are proved below: the static form under the well formedness hypothesis that makes G7 classes resolve to a common canonical successor, and the realized history form that holds for every well formed family.

Definition 19 (Fulfilled Branch)**.** A branch b_i is fulfilled for parent s if there exists a canonical witness w_i such that:

$$\text{VerifyGuard}(s, g_i, w_i) = 1.$$

Lemma 2 (Guard Fulfillment and Linearity Exclude Co Realization)**.** *For a well formed guard family Γ_s, if $b_i \#_s b_j$, then b_i and b_j cannot both produce realized successors from the same consumed resource.*

Proof. There are two admissible cases. First, the guard class proves exclusive fulfillment under G5. Then no accepted witnesses can make both branch guards true. Second, the conflict class is hardened by G7. Then by Lemma 1 the two branches share the derived key of the resource they contend for. Even if both guards are fulfilled, a realized transition must consume that shared key, and the consumed parent set permits it to be consumed only once, so at most one branch is recorded as realized. Thus guard fulfillment and linearity jointly exclude co realization; in the G7 case the exclusion is a property of the realized history, made precise in Section 17. □

12 Admissible Guard Classes

Guard well formedness must be mechanically checkable. This section gives standard guard classes whose exclusivity obligations discharge G5, G7, or both. A DSM implementation may support additional guard classes only when each additional class provides an equivalent machine checkable proof.

Definition 20 (Selector Guard Class)**.** A selector guard family has a deterministic selector:

$$\text{Select}_{s,G}(w) \in B_G \cup \{\bot\},$$

where G is a conflict class and B_G is the finite branch set for that class. Branch $b \in B_G$ is fulfilled iff:

$$\text{Select}_{s,G}(w) = b.$$

Proposition 1 (Selector Guards Are Exclusive)**.** *A selector guard family satisfies exclusive fulfillment for its conflict class.*

Proof. The selector is deterministic and returns at most one branch value for a witness and conflict class. Therefore no single canonical witness can select two branches. If two witnesses are supplied, the guard construction still requires them to evaluate through the same deterministic conflict class selector. A well formed selector class rejects a branch family whose verifier can accept two different selected values for the same conflict class. Hence two incompatible branches cannot both be fulfilled. □

Definition 21 (Shared Resource Guard Class)**.** A shared resource guard family is one in which every incompatible branch in a conflict class contains the same canonical resource consumption set K_G. Fulfillment of any branch requires consuming K_G.

Proposition 2 (Shared Resource Guards Are Linear)**.** *In a shared resource guard family, two incompatible branches cannot both realize from the same parent.*

Proof. Every incompatible branch requires the same resource consumption set K_G. Linearity validity requires every key in K_G to be absent before the step and present after the step. Once one realized successor consumes K_G, no later successor in the same realized lineage can consume the same key set again. If two claimed successors both consume K_G from the same parent, they are conflicting claims for the same linear resource and are subject to the per key uniqueness theorem below. □

Definition 22 (Hash Fulfillment Guard Class)**.** A hash fulfillment guard is bound to a committed digest $y = \mathrm{H}(D \parallel m)$ under a domain D. A branch is fulfilled by a preimage m only when:

$$\mathrm{H}(D \parallel m) = y$$

and the guard's branch binding domain matches the branch identifier and conflict class.

Proposition 3 (Hash Fulfillment Guards Are Exclusive When Selector Bound or Resource Bound)**.** *A hash fulfillment branch family is exclusive when the fulfilled preimage is first mapped through a deterministic branch selector or all incompatible branches share the same resource consumption key set.*

Proof. A raw hash preimage alone proves knowledge of a committed value. It does not by itself select among incompatible branches unless the branch selection is domain bound. If the preimage is mapped through a deterministic selector, the selector guard argument applies. If incompatible branches share a resource consumption key set, the shared resource argument applies. Therefore a hash fulfillment guard is admissible only when it is selector bound or shared resource bound. □

Definition 23 (DLV Generation Guard Class)**.** A DLV generation guard family binds all terminal branches of a vault generation to one vault identifier, one logical generation, one consumed parent, and one vault generation resource key set. A deterministic decision function maps completion evidence to at most one terminal branch.

Proposition 4 (DLV Generation Guards Are Exclusive)**.** *A well formed DLV generation guard family satisfies exclusive fulfillment and resource key consistency.*

Proof. All incompatible terminal branches are bound to the same vault generation and share the same vault generation resource key set. The DLV decision function returns at most one terminal branch. Therefore incompatible branches cannot both satisfy the guard verifier, and even a repeated realization attempt would have to consume the same vault generation resource key set. □

Definition 24 (CPTA Guard Class)**.** A CPTA guard family binds token operations to an immutable policy commitment, a token genesis, and the relevant token resource descriptor. Transfer, mint, burn, deterministic emission, claim, and recovery branches are ordinary guarded candidates. Each conflicting branch must consume the same token balance resource, source vault resource, claim resource, or policy generation resource as appropriate.

Proposition 5 (CPTA Guards Are Linear Policy Guards)**.** *A well formed CPTA guard family satisfies exclusive realization for conflicting token operations.*

Proof. A CPTA policy is deterministic and committed by digest. Transfers consume balance resources. Deterministic emissions consume a source vault generation and an emission or claim resource. Mint and burn consume the relevant policy authority generation and balance resource. If two token operations conflict, they either target the same balance object, source vault generation, claim object, or policy generation. G7 requires those conflicting operations to resolve to the same

canonical resource key for the resource they share. Therefore two conflicting CPTA operations cannot both realize. □

Definition 25 (Recovery Generation Guard Class)**.** A recovery generation guard family binds recovery, cancel, abort, tombstone, and succession branches to a recovery generation identifier, a parent authority state, and a shared recovery generation resource key set. A deterministic recovery decision object selects at most one terminal authority branch.

Proposition 6 (Recovery Generation Guards Are Exclusive)**.** *A well formed recovery generation guard family satisfies exclusive realization.*

Proof. The recovery decision object is deterministic and branch bound. It returns at most one terminal branch for the recovery generation. All terminal branches consume the shared recovery generation key set. Thus the selector and shared resource arguments both apply. □

Definition 26 (Offline Anchor Step Guard Class)**.** An offline anchor step guard family binds offline bearer candidates to the same offline origin, that is, the sender device root, offline frontier root, and committed anchor counter (R_i, h_i, u_i) of Section 30, to the same receiver challenge domain and transition digest, and to the same derived anchor step resource key of Definition 66. The key is derived from the committed parent under Rule 2; it is not branch data.

Proposition 7 (Offline Anchor Step Guards Are Linear)**.** *A well formed offline anchor step guard family satisfies exclusive realization by shared resource linearity alone.*

Proof. Two distinct offline candidates from one origin both consume the derived anchor step key, hence conflict, and by Realized History Uniqueness (Theorem 3) at most one is folded into any realized history. No hardware evidence appears in this argument. Hardware evidence enters only the mode validity predicate of Section 30, where it decides device identity rather than uniqueness. □

13 Consumption Keys and Parent Linearity

Parent linearity must be enforced structurally. DSM models this through canonical resource consumption keys and a consumed parent set Σ.

The key point is that the key used to exclude conflicting branches must not include the branch identifier. Otherwise release, refund, abort, and recovery branches over the same resource would receive different exclusion keys and would not share the structural lock. DSM therefore separates resource consumption keys from optional branch local keys.

Definition 27 (Resource Descriptor)**.** A resource descriptor x is the canonical description of the linear object being consumed. Examples include:

- a relationship parent descriptor;
- a spendable object descriptor;
- a DLV vault generation descriptor;
- a CPTA balance object descriptor;
- a CPTA source vault emission generation descriptor;
- a recovery generation descriptor;

- an offline anchor step descriptor.

A resource descriptor is derived from committed parent state and protocol fields. It is not discretionary branch input. By Assumption 4 the derivation is canonical and injective over the resources committed by a state.

Definition 28 (Resource Consumption Key)**.** For a parent state s and a linear resource descriptor x, the resource consumption key is:

$$\kappa_{\mathrm{res}}(s, x) = \mathrm{H}(\texttt{"DSM/consume_resource/v1"} \parallel \rho_{\mathrm{core},s} \parallel x),$$

where $\rho_{\mathrm{core},s}$ is the core root of s. This is the shared key used to enforce parent resource linearity. Mutually exclusive branches over the same consumed resource must share the $\kappa_{\mathrm{res}}(s, x)$ of that resource, and under the hardened rule must derive it rather than supply it.

Definition 29 (Branch Local Consumption Key)**.** For a parent state s, branch b, and resource descriptor x, the optional branch local key is:

$$\kappa_{\mathrm{branch}}(s, b, x) = \mathrm{H}(\texttt{"DSM/consume_branch/v1"} \parallel \rho_{\mathrm{core},s} \parallel b \parallel x).$$

Branch local keys may be used for audit, indexing, branch specific receipts, or mode specific bookkeeping. They do not replace the shared resource consumption key required for conflict exclusion.

Definition 30 (Transition Consumption Set)**.** The resource consumption set of a candidate transition $s \to s'$ is:

$$\mathrm{Keys}(s, s') \subseteq \{\kappa_{\mathrm{res}}(s, x) : x \text{ is a linear resource descriptor committed by } s\}.$$

The set $\mathrm{Keys}(s, s')$ contains the shared resource keys consumed by the transition. Branch local keys may be carried separately, but are not sufficient for conflict exclusion.

Definition 31 (Consumed Parent Set)**.** The consumed parent set Σ_s is a canonical set, represented for example as an SMT, whose leaves record resource consumption keys already consumed in the realized history represented by state s.

Definition 32 (Consumption Validity)**.** A transition $s \to s'$ with branch b and resource consumption key set $K_b = \mathrm{Keys}(s, s')$ is consumption valid iff:

(1) all keys in K_b are absent from Σ_s;

(2) $\Sigma_{s'} = \Sigma_s \cup K_b$, with the union proven by canonical SMT update proofs;

(3) every key in K_b is derived from $\rho_{\mathrm{core},s}$, the branch family, and the linear resource consumed by the transition;

(4) if branch local keys are present, each branch local key is bound to the corresponding shared resource consumption key;

(5) no conflicting candidate in the same conflict class derives a different key for a resource it shares with this transition.

Definition 33 (Linearity Validity)**.** LinearityOK(s, s') holds iff consumption validity holds, relationship parents are not transferred across relationship identifiers, and no named resource is consumed more than once in the successor state.

Lemma 3 (Parent Consumption Is Local and Structural)**.** *A valid realized transition from s to s' structurally records the consumption of its parent resource by updating Σ_s into $\Sigma_{s'}$.*

Proof. This is exactly the consumption validity predicate. The successor root commits to $\Sigma_{s'}$. A verifier recomputes the consumed parent set update and rejects if the key was already present, if the successor does not contain the required consumed marker, or if conflicting candidates in the same conflict class derive different keys for a shared resource. □

Remark 3. The consumed parent set prevents reuse of a parent along a realized chain. The guard family well formedness rule prevents malformed branch families from entering the valid state space. The resource consumption key, not the branch local key, is the exclusion key.

14 Hardened Resource Key Discipline

This section states the production rule that prevents future features from breaking DSM linearity.

Rule 2 (Derived Resource Keys Only)**.** A verifier must derive authoritative resource consumption keys from parent state and canonical resource descriptors. It must not accept a branch supplied resource key as authoritative unless that key is proven to equal the derived key.

Definition 34 (Relationship Resource Key)**.** For a relationship between devices A and B, let the relationship leaf key be:

$$k_{A\leftrightarrow B} = \mathrm{H}(\texttt{"DSM/smt-key/v1"} \parallel \min(\mathrm{DevID}_A, \mathrm{DevID}_B) \parallel \max(\mathrm{DevID}_A, \mathrm{DevID}_B)).$$

For a parent state s with core root $\rho_{\mathrm{core},s}$, the resource descriptor is:

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

and the authoritative resource consumption key is:

$$K_{\mathrm{rel}} = \kappa_{\mathrm{res}}(s, x_{\mathrm{rel}}).$$

Because $k_{A\leftrightarrow B}$ embeds both device identifiers, the resource descriptor, and therefore the derived key, is scoped to this relationship. A parent chain node h_n under relationship $A \leftrightarrow B$ yields a different key from any node of a different relationship, even if two relationships happen to reuse the same node name.

Definition 35 (DLV Generation Resource Key)**.** For vault identifier V, vault generation g, core root $\rho_{\mathrm{core},s}$, and consumed parent p, the DLV resource descriptor is:

$$x_{\mathrm{dlv}} = (\texttt{"dlv"}, V, g, p),$$

and the authoritative key is:

$$K_{\mathrm{dlv}} = \kappa_{\mathrm{res}}(s, x_{\mathrm{dlv}}).$$

Definition 36 (CPTA Resource Keys)**.** For token genesis G_T, policy commitment P_T, and core root $\rho_{\text{core},s}$, CPTA resource descriptors include:

$$x_{\text{bal}} = (\texttt{"cpta-balance"}, G_T, P_T, \text{holder}),$$

$$x_{\text{vault}} = (\texttt{"cpta-source-vault"}, G_T, P_T, V, g),$$

$$x_{\text{claim}} = (\texttt{"cpta-claim"}, G_T, P_T, \text{claimant}, \text{claim_id}),$$

$$x_{\text{policy}} = (\texttt{"cpta-policy-generation"}, G_T, P_T, g).$$

The corresponding keys are derived by κ_{res}. A conflicting CPTA operation must consume the same relevant key for the resource it shares.

Rule 3 (Single Consumption Invariant)**.** For any state s, any conflict class at key K, and any two candidate successors $s_i \neq s_j$ in $C_G(s)$, both consume K:

$$s_i, s_j \in C_G(s) \Rightarrow K \in \text{Keys}(s, s_i) \cap \text{Keys}(s, s_j).$$

The shared key K may be consumed at most once. Therefore at most one candidate in the conflict class at K may be realized, and the branches need not agree on any resource key they do not share.

Theorem 1 (Hardened Single Consumption)**.** *Under Assumptions 1 through 4, Rule 2, and Consumption Validity, at most one candidate in a conflict class at a key K can validate and realize into an accepted history.*

Proof. Let s_i and s_j be candidates in the conflict class at K. By Definition 18 both consume K, and by Lemma 1 the sharing of K is forced by the shared resource under Assumption 4. Consider the realized history as the sequence of accepted transitions against the verifier's live consumed parent set. If s_i is accepted first, then Σ is updated to include K. A later attempt to accept s_j fails because every accepted transition requires the keys in its consumption set to be absent from the current consumed parent set before the step, and K is no longer absent. If both are claimed as accepted successors from the same parent against the same consumed parent set, then they are two claimed realizations of the same absent to present transition for K. This contradicts consumption validity, canonical successor recomputation, or hash soundness. Hence at most one candidate in the class is accepted. The candidates may differ on resources they do not share; only agreement on K is used. □

Remark 4. This theorem is the code level rule: conflicting futures must fight over the same lock. The lock is derived from the parent resource. It is not chosen by the branch. The exclusion is a property of the accepted history against a live consumed parent set; the machine checked realized history invariant of Appendix B is exactly this statement.

15 Layered Realization Predicate

The DSM step predicate is deliberately factored. This prevents the policy layer, mode layer, or transport layer from hiding the fork exclusion invariant.

Definition 37 (Candidate Validity)**.** CandidateOK(s, s') holds iff:

(1) s' is canonically encoded;

(2) H(enc(s')) is committed by $P(s)$;

(3) the branch identifier, transition digest, and consumed resource keys match the committed candidate record;

(4) the guard descriptor belongs to the committed well formed guard family Γ_s;

(5) the derived resource key set for the candidate matches the committed resource key set.

Definition 38 (Guard Validity)**.** GuardOK(s, s') holds iff the branch selected by s' has a canonical witness accepted by the guard verifier in Γ_s.

Definition 39 (Structural Validity)**.** StructuralOK(s, s') holds iff:

(1) projection commutativity holds for every relationship projection;

(2) all unrelated relationship parents are preserved;

(3) the progression object advances according to its mode;

(4) all SMT roots recompute byte exactly;

(5) all inclusion and non inclusion proofs verify;

(6) all signatures required by the transition verify over domain separated canonical bytes.

Definition 40 (Policy Validity)**.** PolicyOK(s, s') holds iff the deterministic policy object Π_s accepts the transition under the parent state and candidate record. Policy may restrict which precommitted branch may be accepted, but it may not select among conflicting realized branches by external ordering coordinates, fee, arrival order, storage node preference, validator vote, or discretionary ordering after commitment.

Definition 41 (Mode Validity)**.** ModeOK(s, s') holds iff the selected mode evidence verifies. For ordinary online DSM, this predicate is empty except for local frontier checks. For offline bearer DSM, this predicate verifies the identity witnessed advance defined in Section 30.

Definition 42 (DSM Realized Step)**.** The realized step predicate is:

$$\begin{aligned}\mathrm{Step}(s, s') \iff \mathrm{WF}(s) \wedge \mathrm{CandidateOK}(s, s') \wedge \mathrm{GuardOK}(s, s') \wedge \mathrm{StructuralOK}(s, s') \\ \wedge \mathrm{LinearityOK}(s, s') \wedge \mathrm{PolicyOK}(s, s') \wedge \mathrm{ModeOK}(s, s').\end{aligned}$$

Here LinearityOK is evaluated against the consumed parent set Σ_s recorded in s. The predicate Step(s, s') therefore expresses that s' is an admissible successor of s relative to the consumed parent set carried by s. Acceptance into a realized history additionally advances that consumed parent set; the interaction between the two is made precise in Section 17.

Definition 43 (Key Scoped Realized Step)**.** For a resource consumption key K, define:

$$\mathrm{Step}_K(s, s') \iff \mathrm{Step}(s, s') \wedge K \in \mathrm{Keys}(s, s').$$

This is the realized step predicate scoped to a fixed consumed parent resource.

16 Policy Discipline

Policy is a common place for accidental reintroduction of consensus like selection. DSM therefore constrains policy.

Definition 44 (DSM Admissible Policy)**.** A policy object Π is DSM admissible if:

(1) it is canonical and hash committed by the parent state;

(2) it is deterministic over canonical parent, candidate, guard, witness, and proof bytes;

(3) it does not reference external ordering coordinates, consensus ordering, fee ordering, validator ordering, storage node arrival order, or any clock based rule in acceptance predicates;

(4) it cannot mutate the committed candidate set after precommitment;

(5) it cannot authorize two conflicting branches from the same consumed parent;

(6) it preserves token conservation, non negativity, and other linear resource invariants when those resources are present;

(7) it cannot assign different resource consumption keys to conflicting candidates for a resource they share.

Lemma 4 (Policy Cannot Reintroduce Fork Selection)**.** *If* Π *is DSM admissible, then policy validity cannot make two conflicting branches from the same consumed parent both realized into an accepted history.*

Proof. A DSM admissible policy cannot mutate the candidate set, cannot select by external ordering, cannot authorize two conflicting branches from the same consumed parent, and cannot split conflicting candidates across different keys for a shared resource. Therefore it can only reject candidates or accept a candidate already allowed by the well formed guard family and linearity predicates. Since those predicates already exclude co realization of conflicting branches in the realized history, policy cannot reintroduce it. □

17 Conflict Local Realization Uniqueness

This section states the fork exclusion result. It is stated in two forms, because a well formed guard family excludes co realization at two different points depending on which well formedness rule it relies on, as previewed in Remark 2.

The static per state form is a property of the step predicate at a single parent state: for a family in which conflicting fulfilled branches resolve to the same canonical successor, at most one successor satisfies the key scoped predicate. This covers G5 classes outright, and G7 classes under the hypothesis that shared resource consumption together with canonical successor recomputation forces a single successor.

The realized history form is a property of an accepted history against a live consumed parent set: no accepted history records two distinct successors consuming the same resource key from the same parent. This holds for every well formed family, including a G7 class in which two conflicting candidates could each independently satisfy the static predicate against a common unconsumed parent set. The realized history form is the operational safety statement, and it is the one machine checked as an invariant in Appendix B.

Both forms are proved below. The distinction matters because a purely static reading of uniqueness over the stateless predicate would be too strong for a G7 only class: two conflicting candidates can each satisfy Step_K against the same parent when the parent set does not yet record the shared key. What excludes them is the consumed parent set update at acceptance, which is a property of the history, not of the isolated predicate.

Definition 45 (Key Scoped Realized Successor Set)**.** For a DSM state s and resource consumption key K, the key scoped realized successor set is:

$$R_K(s) = \{s' \mid \mathrm{Step}_K(s, s')\}.$$

Definition 46 (Canonical Successor Resolution)**.** A conflict class at key K and parent s is canonically resolving if, whenever two branches b_i, b_j in the class are both fulfilled, their committed candidate successors are byte identical: $s_i = s_j$. A G5 class is canonically resolving because at most one branch in the class is ever fulfilled. A G7 class is canonically resolving when its committed candidate records map the shared consumed resource to a single canonical successor state, so that fulfillment of any branch in the class recomputes the same s'.

Theorem 2 (Static Uniqueness of Realized Successor per Consumed Parent)**.** *Under Assumptions 1 through 4, Rule 2, and Consumption Validity, let s be a well formed DSM state and K a resource consumption key such that the conflict class at K is canonically resolving in the sense of Definition 46. Then there exists at most one realized successor consuming K:*

$$\forall s_1, s_2, \quad \mathrm{Step}_K(s, s_1) \wedge \mathrm{Step}_K(s, s_2) \Rightarrow s_1 = s_2.$$

Proof. Assume $\mathrm{Step}_K(s, s_1)$ and $\mathrm{Step}_K(s, s_2)$. By definition of Step_K, both successors consume the same resource key K, and by Step both satisfy candidate validity, guard validity, structural validity, linearity validity, policy validity, and mode validity against the consumed parent set of s.

Both s_1 and s_2 are selected by fulfilled branches whose consumption set contains K, so both lie in the conflict class at K (Definition 18). By hypothesis that class is canonically resolving, so the committed candidate successors of any two fulfilled branches in it are byte identical. A valid successor is canonically bound by its candidate digest and recomputed SMT root; if s_1 and s_2 carried distinct candidate digests they would be distinct fulfilled branches of the class with distinct successors, contradicting canonical resolution, and if they carried the same candidate digest but differed as states, canonical encoding injectivity or hash soundness is violated. Hence $s_1 = s_2$. □

Definition 47 (Realized History)**.** A realized history is a finite sequence of accepted transitions:

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

in which each step $s_t \to s_{t+1}$ satisfies $\mathrm{Step}(s_t, s_{t+1})$ evaluated against the consumed parent set Σ_{s_t}, and $\Sigma_{s_{t+1}} = \Sigma_{s_t} \cup \mathrm{Keys}(s_t, s_{t+1})$. Equivalently, a realized history is the fold of the accept transition over an initial state, threading the consumed parent set forward. A key scoped realized fork in a history is a pair of accepted steps out of the same parent occurrence consuming the same key toward distinct successors.

Theorem 3 (Realized History Uniqueness)**.** *Under Assumptions 1 through 4, Rule 2, and Consumption Validity, for every well formed guard family, no realized history contains a key scoped realized fork. That is, if a realized history accepts $s \to s_1$ and $s \to s_2$ from the same parent occurrence s, both consuming resource key K, then $s_1 = s_2$.*

Proof. Suppose a realized history accepts two steps out of the same parent occurrence s, both consuming K. The first accepted step updates the live consumed parent set from Σ_s to $\Sigma_s \cup \{K, \ldots\}$, so K is present afterward. Any subsequent accepted step out of the same parent occurrence is evaluated against a consumed parent set that already contains K; its LinearityOK conjunct requires K to be absent and therefore fails. Hence at most one step consuming K is accepted out of a given parent occurrence, and if the history records two, they must be the same accepted step with the same successor, so $s_1 = s_2$. This holds regardless of whether the family relies on G5 or on G7 alone: G7 permits both candidates to satisfy the static predicate against the common pre acceptance set, but only one can be folded into the history before the shared key is marked consumed. □

Corollary 1 (Candidate Multiplicity Does Not Imply Realized Forking)**.** *It is possible that:*

$$|C(s)| > 1,$$

while no realized history contains a key scoped realized fork, and, when the conflict class at K is canonically resolving, additionally:

$$|R_K(s)| \leq 1.$$

Proof. Candidate multiplicity follows from precommitment forking. Realized history uniqueness follows from Theorem 3 for every well formed family. The stronger static bound on $|R_K(s)|$ follows from Theorem 2 under the canonical resolution hypothesis. □

Proposition 8 (Disjoint Realizations May Coexist)**.** *If two realized transitions from s consume disjoint resource key sets and touch disjoint relationship projections, DSM does not require them to be equal. They are independent transitions rather than a realized fork.*

Proof. The uniqueness results are scoped to a fixed consumed resource key. If $\mathrm{Keys}(s, s_1) \cap \mathrm{Keys}(s, s_2) = \emptyset$ and the named relationship projections are disjoint, then the two transitions do not compete for the same linear parent. Projection commutativity preserves unrelated relationships, and SMT updates for disjoint leaves commute when their canonical proofs verify. Thus the two transitions may both be valid independent local advances. They are not a double spend and not a conflict local realized fork. □

18 Deterministic Limbo Vaults as Guarded Branch Systems

A Deterministic Limbo Vault, or DLV, is a guarded precommitment construction. It encumbers value under a condition set and releases value only when a deterministic fulfillment proof exists.

Let a vault configuration be:

$$V = (L, C, H_V),$$

where L is the lock, C is the condition set, and H_V is the vault commitment context.

A typical DLV unlock secret has the form:

$$sk_V = \mathrm{H}(\texttt{"DSM/dlv_unlock/v1"} \parallel L \parallel C \parallel \sigma),$$

where σ is a stitched proof of completion.

The precommitted branch space may include:

$$(g_{\text{release}}, s_{\text{release}}), (g_{\text{refund}}, s_{\text{refund}}), (g_{\text{abort}}, s_{\text{abort}}), (g_{\text{recovery}}, s_{\text{recovery}}).$$

These are not four executed states. They are guarded candidates.

Definition 48 (DLV Decision Function)**.** For a vault generation g, define a deterministic decision function:

$$\text{Decision}_{V,g}(\sigma) \in \{\text{release}, \text{refund}, \text{abort}, \text{recovery}, \perp\}.$$

A DLV branch b is fulfilled only if there exists a canonical witness σ such that:

$$\text{Decision}_{V,g}(\sigma) = b.$$

Rule 4 (DLV Guard Well Formedness)**.** A DLV branch family is valid only if:

(1) all branches are bound to the same vault generation and consumed parent;

(2) all branches share the vault generation resource consumption key set;

(3) the decision function returns at most one terminal branch;

(4) every branch witness is bound to the vault configuration, parent root, generation, and branch identifier;

(5) no branch supplied key can override the derived DLV generation key.

Proposition 9 (DLV Branch Exclusivity)**.** *For a correctly constructed DLV, incompatible fulfillment branches cannot both realize from the same parent.*

Proof. All incompatible DLV branches are bound to the same vault generation and consumed parent. The DLV decision function is deterministic and returns at most one terminal branch, so the class is a G5 class and is canonically resolving; Theorem 2 applies. The branches also share the vault generation resource consumption key set, so Theorem 3 applies as well and a realized chain cannot consume the same vault generation twice. □

19 CPTA as a Guarded Policy System

A Context Policy and Token Anchor, or CPTA, is an immutable token policy object committed by canonical bytes. CPTA does not introduce a separate execution model. It is a structured source of deterministic policy guards and resource descriptors inside DSM.

Definition 49 (CPTA Policy Object)**.** A CPTA policy object is a canonical object:

$$T = (G_T, P_T, \text{kind}, \text{supply}, \text{authority}, \text{emission}, \text{constraints}),$$

where G_T is the token genesis, P_T is the policy commitment, and the remaining fields define deterministic token behavior. Any token operation referencing the token must include P_T, and a verifier rejects if the referenced policy commitment differs from the creation commitment.

Definition 50 (CPTA Candidate Branches)**.** For a parent token state s, a CPTA may define candidate branches such as:

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

Each branch is a DSM candidate future. It is not realized merely because it is present in the policy.

Definition 51 (CPTA Guards)**.** CPTA guards include:

- transfer guards checking signatures, balance non negativity, transferability, and policy commitment;
- mint guards checking threshold authority and cap preservation;
- burn guards checking threshold authority and balance non negativity;
- emission guards checking deterministic source vault debit, emission index, claim proof, and remaining supply;
- claim guards checking inclusion proof or anchored eligibility evidence;
- recovery guards checking deterministic recovery authority and generation state.

Rule 5 (CPTA Resource Discipline)**.** A CPTA operation must consume the relevant derived resource key set:

(1) a transfer consumes sender and recipient balance resources as required by the token operation;

(2) an emission consumes the source DLV generation resource and the emission claim resource;

(3) a mint consumes the policy authority generation and the destination balance resource;

(4) a burn consumes the source balance resource and policy authority generation when authority is required;

(5) an NFT or SBT claim consumes the claimant claim resource;

(6) recovery consumes the recovery generation resource.

Two conflicting CPTA operations are valid candidates only if they derive the same resource key for the resource they share.

Theorem 4 (CPTA Operations Are DSM Steps)**.** *Every accepted CPTA operation is a DSM realized step. It is accepted only if candidate validity, guard validity, structural validity, linearity validity, policy validity, and mode validity all hold.*

Proof. A CPTA operation is committed by its candidate digest and policy commitment. Its guard is the deterministic CPTA predicate for the selected operation. Its resources are the derived token, balance, source vault, claim, or policy generation keys. Its structural validity requires canonical SMT updates and signature verification. Its policy validity requires the immutable CPTA to accept the operation. Its linearity validity requires all consumed resource keys to be absent and then inserted. Therefore it is exactly a DSM realized step. □

Corollary 2 (CPTA Does Not Replace DLV)**.** *A CPTA source vault or emission construction may use a DLV as one of its resources. The CPTA supplies policy guards. The DLV supplies a vault guarded branch system. Both are instances of the same DSM guarded linear kernel.*

20 Multi Party Authority as Guarded Bilateral Composition

DSM remains bilateral at the protocol layer. Multi party workflows are composed from multiple bilateral relationships and shared external commitments. They do not require a global contract account, validator ordering, or a three party ledger.

The important property is stronger than ordinary threshold authorization. Authority may be overlapping, asymmetric, or even excessive. Several parties may produce witnesses for several branches. That still does not create a fork if all conflicting branches consume the same derived resource key for the resource they share. Authority can choose an eligible branch. Authority cannot multiply the consumed parent.

Definition 52 (External Commitment)**.** An external commitment is a digest:

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

where X is data outside the DSM core. DSM does not execute X. It verifies only equality, inclusion, signature, or proof predicates bound to Y.

Definition 53 (Multi Party Authority Branch)**.** A multi party authority branch is a candidate whose guard requires a threshold or Boolean formula over signed receipts, external commitments, or proof objects. For example:

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

Rule 6 (Multi Party Resource Discipline)**.** If multi party branches conflict over the same DSM resource, they must derive the same resource key for that resource. The fact that different parties authorize different branches does not permit the branches to consume different keys for the same resource.

Proposition 10 (Authority Does Not Create Concurrency Hazards)**.** *Even if every authority witness for every conflicting branch is available, at most one conflicting branch can realize into an accepted history.*

Proof. Suppose release, cancel, and recovery witnesses are all available. The guard predicates may all be true, so several branches are fulfilled simultaneously. This is the characteristic G7 situation: the branches conflict over the same parent resource, so by Lemma 1 they share the derived key of that resource. Both candidates may satisfy the static step predicate against the common pre acceptance consumed parent set, so the static uniqueness theorem does not by itself exclude them. Theorem 3 does: once one branch is folded into the realized history, that key is recorded in the consumed parent set, and every other conflicting branch then fails LinearityOK against the updated set. Therefore overlapping signatures do not create a realized fork in any accepted history. □

Corollary 3 (Bilateral Composition Supports Multi Party Policy)**.** *Multi party authority does not require a global execution account. It can be represented as proof carrying bilateral composition over external commitments, provided all conflicting branches derive the same key for a shared resource.*

21 Tripwire

Tripwire is the theorem that a fork from the same consumed parent cannot remain hidden as two valid realized DSM histories. It is not a consensus rule. It is a derivability rule.

Definition 54 (Key Scoped Realized Fork)**.** A realized fork from predecessor s at resource consumption key K is a pair of distinct accepted successors s_a, s_b in a realized history, both out of the same parent occurrence s, such that:

$$\mathrm{Step}_K(s, s_a) \wedge \mathrm{Step}_K(s, s_b) \wedge s_a \neq s_b,$$

with both steps folded into the same history in the sense of Definition 47.

Theorem 5 (Guarded Tripwire)**.** *Under Assumptions 1 through 4, Rule 2, and Consumption Validity, for every well formed guard family there does not exist a valid key scoped realized fork in any realized history:*

$$\neg\exists\, \textit{history } H,\ s, s_a, s_b, K : s_a, s_b \textit{ accepted in } H \textit{ from } s,\ \mathrm{Step}_K(s, s_a) \wedge \mathrm{Step}_K(s, s_b) \wedge s_a \neq s_b.$$

Proof. Immediate from Realized History Uniqueness (Theorem 3). If both s_a and s_b are accepted from the same parent occurrence consuming K, then $s_a = s_b$, contradicting $s_a \neq s_b$. When the conflict class at K is canonically resolving, the stronger static statement of Theorem 2 also excludes two distinct Step_K successors at the predicate level. □

Remark 5 (Tripwire validation boundary)**.** Tripwire does not claim that an adversary cannot construct, copy, or distribute conflicting candidate bytes. It claims that conflicting candidates cannot both validate and be folded into a realized history for the same consumed resource. Therefore they cannot both go through as accepted state transitions. At most one can be validated and realized. All other conflicting candidates are rejected as non derivable with respect to that consumed resource once the shared key is recorded consumed.

Remark 6 (Online same parent forks are unconstructible)**.** The different receivers accept conflicting packages shape of the adversarial delivery model cannot arise for an online relationship resource, for two structural reasons that are enforced by the kernel rather than assumed. First, the derived consumption key embeds the relationship identity: by the Relationship Resource Key definition, x_{rel} contains $k_{A\leftrightarrow B}$, so every relationship is its own straight hash chain and a parent under relationship r is not replayable under a different relationship q, even when two chains reuse the same node name. Two candidates that consume the same key are therefore necessarily candidates of the same relationship. Second, the topology is bilateral: the acceptor of a relationship step is exactly that relationship's counterparty, so there is no second, independent online receiver of the same relationship parent against which a conflicting successor could be separately realized. Within the single receiver, Theorem 3 already excludes a local realized fork.

Offline bearer mode is the only case that appears, superficially, to permit presentation of one portable bearer state to different receivers. The corrected offline construction removes that shape as a valid execution path. A correctly implemented enrolled appliance has one active SMT state, one offline frontier, one committed anchor counter leaf, one prepared or committed record for a frontier, and one physical counter tracker. A commit from (R_i, h_i, u_i) advances that state to $(R_{i+1}, h_{i+1}, u_i + 1)$ atomically, and by Assumption 5 the release witnesses are minted only at that commit, after the counter decrement, so no witness for the old origin exists to reuse. If two same origin offline releases exist, the boundary of Assumption 5 has already been crossed: for example by a perfect live clone, duplicated non exportable signing state, a measurement boundary breach, a concurrency bug that mints a second witness before committing, or a verifier that accepts branch supplied roots rather than derived SMT roots and counter leaves. Appendix B records the relationship scoped model and the structure removed contrast model.

22 Reachability and Canonical Realized History

Define the reflexive transitive closure of the realized step relation:

$$s \to^* t.$$

Global DSM evolution is generally a conflict free partial order of realized steps, not necessarily a single total order. Independent relationship updates may commute. For a fixed consumed resource lineage, however, realized history is unique.

Throughout this section a resource lineage means the family of resource descriptors for one linear resource across its successive generations, together with the per step resource consumption keys derived at each state. Because each key $\kappa_{\mathrm{res}}(s, x)$ binds the core root $\rho_{\mathrm{core},s}$ of the state it is consumed from, the key at step n and the key at step $n + 1$ are distinct even for the same underlying resource. We therefore index a lineage by a lineage identifier ℓ, the resource descriptor family modulo generation, and reserve the symbol K for the individual per step key of Definition 28 and its scoped step Step_K. A step in lineage ℓ is written $s \to_\ell s'$ and consumes whichever per step key K the descriptor family yields at s.

Definition 55 (Resource Local Canonical Realized History)**.** For a resource lineage ℓ, the canonical realized history of state s_n with respect to ℓ is the unique sequence of realized steps consuming the successive per step resource keys of that lineage:

$$s_0 \to_\ell s_1 \to_\ell \cdots \to_\ell s_n.$$

Candidate branches not realized are not part of the realized history. They are part of the precommitted possibility space.

Theorem 6 (Unique Conflict Local Realized History)**.** *Under Rule 2, if two states a and b are reachable by realized steps from the same valid ancestor s_0, and the two derivations claim incompatible consumption of the same canonically derived resource lineage ℓ, then they are prefix comparable on that resource lineage. In particular, if they are reached at the same resource local realized step index, then $a = b$ for that resource local successor.*

Proof. By induction on the length of the shorter resource local realized derivation. At each consumed parent occurrence in the lineage, the accepted successor is unique by Realized History Uniqueness (Theorem 3): the per step key at that occurrence is recorded consumed on acceptance, so no second distinct successor consuming it can be folded into the same history. Rule 2 ensures that each per step key is derived canonically from the core root rather than supplied by a branch, so the two derivations cannot diverge at the same lineage step by disagreeing on the key. They either produce the same resource local state at the same depth or one resource local derivation is a strict prefix of the other. □

Remark 7. This theorem does not impose a total order on disjoint relationships. It states the DSM invariant: conflicting histories for the same consumed resource lineage cannot both be derived.

23 Linearity and Forward Progress

A DSM realized transition advances the relevant progression object. Therefore any non empty realized transition sequence for the same resource lineage is acyclic when the progression object is strictly forward.

Theorem 7 (No Resource Local Realized Cycles)**.** *There is no non empty realized sequence for the same resource lineage ℓ:*

$$s_0 \to_\ell s_1 \to_\ell \cdots \to_\ell s_m$$

with $m > 0$ and $s_m = s_0$, provided the progression object for that lineage is strictly forward under the realized step predicate.

Proof. Every realized transition for the resource lineage advances the relevant progression object. Therefore after $m > 0$ realized transitions, the progression object has advanced. Hence the final state cannot equal the initial state. □

Remark 8. This paper uses normalization in the operational sense of unique conflict local forward reduction and absence of realized branching or cycles for a consumed resource. Infinite forward traces may still exist if policy permits unbounded state evolution. Strong termination is a separate bounded resource property, not required for Tripwire.

24 Concurrency and Message Delivery

DSM does not assume synchronous delivery.

Definition 56 (Pending Candidate)**.** A pending candidate is a received object that is precommitted or proof carrying but not currently adjacent to the receiver's local frontier.

Definition 57 (Online Acceptance)**.** In ordinary online DSM, a receiver accepts a candidate only if:

(1) the candidate is adjacent to the receiver's current local frontier for the named relationship or object;

(2) the candidate satisfies Step;

(3) no local pending lock for the same consumed parent blocks the transition.

A non adjacent object is queued or rejected. It is not partially applied.

Definition 58 (Pending Lock)**.** A pending lock exists for a parent if the receiver has accepted or is resolving a candidate whose branch family consumes the same parent resource. While the lock exists, conflicting offline initiation for that relationship or object is invalid.

Proposition 11 (Races Do Not Create Co Realization)**.** *Concurrent presentation of multiple candidates does not create two realized successors for the same consumed parent at an honest receiver.*

Proof. An honest receiver evaluates candidates against one local frontier for the named relationship or object, folding accepted steps into a single realized history. A candidate that is not adjacent to that frontier is queued or rejected. A candidate that is adjacent must satisfy Step. By Realized History Uniqueness, two conflicting adjacent candidates consuming the same resource key cannot both be folded into that receiver's history: the first accepted marks the key consumed and the second fails LinearityOK. □

Proposition 12 (Disjoint Relationship Concurrency)**.** *Concurrent presentation of candidates for disjoint relationships may produce independent valid advances.*

Proof. If the candidates consume disjoint resource keys and touch disjoint relationship projections, they are not conflicting candidates. Each is checked against its own local frontier and proof material. Projection commutativity preserves unrelated relationships. Therefore no global ordering rule is needed to decide between them. □

Remark 9. DSM proves safety, not unconditional liveness. A guard may never be fulfilled, a counterparty may refuse cooperation, a storage path may fail, or an offline receiver may lack required evidence. In those cases the state waits, aborts, recovers, refunds, or routes to online reconciliation according to policy.

25 Linear Resource Interpretation

DSM state behaves as a collection of guarded linear resources.

Definition 59 (Guarded Linear State Consumption)**.** A DSM parent resource p_i may expose multiple guarded candidates:

$$p_i \multimap \{(g_1, s_1), (g_2, s_2), \ldots, (g_n, s_n)\}.$$

Once one guard is fulfilled and the corresponding branch is realized, the parent resource p_i is consumed. No incompatible branch may later realize from the same parent resource.

In linear logic terms, DSM permits additive choice at the precommitment level but rejects contraction at the realization level.

The branch space for a fixed consumed parent resembles:

$$(g_1 \multimap s_1) \oplus (g_2 \multimap s_2) \oplus \cdots \oplus (g_n \multimap s_n),$$

where $\oplus$ is exclusive choice.

Proposition 13 (Double Spend as Resource Linearity Violation)**.** *A double spend is precisely an attempted violation of resource linearity. It attempts to consume the same parent resource into two distinct realized successors.*

Proof. A double spend requires two distinct realized successors from the same spendable parent, folded into a realized history. This is exactly the key scoped realized fork condition prohibited by Guarded Tripwire (Theorem 5). Therefore a double spend is a non derivable resource linearity violation. □

26 Cut Elimination Interpretation

A cut is an unresolved composition point between computations. In DSM, precommitment forking introduces syntactic cuts. Multiple candidate futures may be laid out. Fulfillment guards eliminate those cuts by allowing at most one candidate to realize for a fixed consumed parent.

Definition 60 (Syntactic Cut)**.** A syntactic DSM cut exists at state s if there are two distinct candidate successors:

$$\exists s_a, s_b : s_a, s_b \in C(s) \wedge s_a \neq s_b.$$

Syntactic cuts are permitted.

Definition 61 (Key Scoped Realized Cut)**.** A key scoped realized DSM cut exists in a realized history at parent occurrence s and resource key K if there are two distinct accepted successors:

$$\exists s_a, s_b : \mathrm{Step}_K(s, s_a) \wedge \mathrm{Step}_K(s, s_b) \wedge s_a \neq s_b,$$

both folded into the same history from s. Key scoped realized cuts are prohibited.

Theorem 8 (Key Scoped Realized Cut Freeness)**.** *Every valid DSM realized history is realized cut free at each consumed resource key.*

Proof. A key scoped realized cut is exactly a key scoped realized fork in a history. Guarded Tripwire proves that no such fork exists for a well formed family. □

Theorem 9 (Guarded Cut Elimination Equivalence)**.** *DSM state evolution is equivalent to guarded cut elimination in the following constrained sense: precommitment may introduce multiple syntactic branches, but fulfillment and resource consumption eliminate all incompatible branches except the unique realized successor for the consumed parent.*

Proof. By precommitment forking, a state may define multiple candidate futures. By guard family well formedness, incompatible candidates cannot all be folded into a realized history: either exclusive fulfillment prevents two guards from being true, or, when several are true, the shared resource key is recorded consumed on the first acceptance and blocks the rest. By unique conflict local realized history, each conflict local realized reduction is unique. By forward progression, resource local realized reductions are acyclic. Therefore DSM evolution is a sequence of guarded reductions in which syntactic cuts may exist in the candidate space but cannot survive as realized cuts. □

27 Category Theoretic Formulation

DSM may be formulated as a small category of state objects and guarded morphisms.

Objects are well formed DSM states. A morphism $f : s \to t$ is a realized DSM step satisfying $\mathrm{Step}(s, t)$. Candidate futures are potential arrows. A guard witness and valid resource consumption turn a potential arrow into a realized arrow.

Composition is ordinary transition composition when the target state of the first step is the source state of the second step and the consumed parent set is threaded forward. Identity morphisms are empty transitions that preserve all components and consume no resource key.

Proposition 14 (Conflict Local Thinness)**.** *For a fixed consumed resource key in a canonically resolving conflict class, the category is thin over that key: there is at most one realized morphism out of the parent consuming that key.*

Proof. This is the static uniqueness theorem restated categorically. If two realized morphisms consume the same key in a canonically resolving conflict class, their target states are equal. □

Proposition 15 (Realized History Thinness)**.** *For every well formed guard family, the path category of accepted realized histories contains no two distinct accepted outgoing morphisms from the same parent occurrence consuming the same key.*

Proof. This is the realized history uniqueness theorem restated categorically. Once the first morphism is composed into the path, the consumed parent set records the key. No second morphism consuming that key is composable from the same parent occurrence. □

28 Denotational Semantics

The denotation of a DSM parent state is a guarded finite possibility space together with a linear realization operator.

Let:

$$D_{\text{cand}}(s) = C(s)$$

be the candidate possibility set, and let:

$$D_{\text{real}}(s) = \{s' \mid \text{Step}(s, s')\}$$

be the statically realizable successor set against the consumed parent set recorded in s.

The operational realized history denotation threads the consumed parent set:

$$D_{\text{hist}}(s_0) = \{s_0 \to s_1 \to \cdots \to s_n : \forall t,\ \text{Step}(s_t, s_{t+1}) \text{ and } \Sigma_{s_{t+1}} = \Sigma_{s_t} \cup \text{Keys}(s_t, s_{t+1})\}.$$

The safety theorem is:

no $D_{\text{hist}}(s_0)$ contains two distinct accepted successors from one parent occurrence consuming one key.

This is the denotational form of Tripwire.

29 Operational Semantics

The operational semantics are a small step transition system over verifier state.

A verifier state contains a current DSM state, a consumed parent set, a pending candidate queue, and mode specific evidence.

An incoming candidate is processed by:

(1) canonical decoding;

(2) candidate digest verification;

(3) guard family verification;

(4) witness verification;

(5) resource key derivation;

(6) consumed parent set non inclusion checks;

(7) structure verification;

(8) policy verification;

(9) mode verification;

(10) consumed parent set update;

(11) successor root adoption.

If any check fails, the candidate is rejected or queued according to policy. No partial application is permitted. The accept transition is atomic at the level of the local verifier: either the successor root and consumed parent set are both adopted, or neither is adopted.

Proposition 16 (Atomic Local Acceptance)**.** *A local verifier cannot accept two conflicting candidates through interleaving if the consumed parent set update and successor root adoption are one atomic accept step.*

Proof. If a candidate is accepted, the consumed parent set is updated before any later conflicting candidate is evaluated against the new verifier state. A conflicting candidate must consume the same key and therefore fails the non inclusion check. If the first candidate is not accepted, no consumed parent set update occurs. In neither case can two conflicting candidates both be folded into the same local realized history. □

30 Offline Bearer Mode: Software Uniqueness, Hardware Identity

DSM does not require a hardware anchor in ordinary online operation. Offline bearer mode exists for the hard case: a receiver accepts without contacting an online reconciliation point at the moment of exchange.

Earlier drafts of this section, and the earlier fused anchor specification, made the physical monotonic counter the uniqueness authority for that case. The receiver was required to witness an authenticated counter read at the sender's FROM coordinate before commit and a second read at the TO coordinate after commit, with binding metadata tying both scalars to the transition. That construction is superseded, for two reasons.

First, the binding never held. A counter read primitive attests a scalar over an authenticated session. It does not see the root advance message. The transition binding wrapped around the scalar was receiver asserted metadata, not chip attested fact. In that older read based design, the scalar evidence itself did not distinguish which transition caused a decrement or bind the decrement to the DSM root advance. The corrected design removes that witness path entirely.

Second, the binding was never needed. Transfer uniqueness is already a theorem of the guarded linear kernel of this paper. Removing the online reconciliation point removes nothing that the kernel used, because the kernel never used one. What the kernel cannot supply is device identity: a software clone holds every host readable byte, and no software predicate distinguishes it from the original. Hardware is therefore assigned exactly one job.

software enforces uniqueness of state transitions;
hardware enforces uniqueness of the physical device instance.

Remark 10 (Relation to the companion phrasing)**.** The companion specification states that the device state is one resource consumed as a whole with no partial advance. That is the same linearity as here, read at the granularity of a single transition and scoped to the relationship it names: an accepted transition consumes its parent occurrence atomically, and the relationship resource key scopes each relationship to its own straight hash chain with a single acceptor (Remark 6). It is not a claim that the whole device advances as one global resource for every relationship at once. This paper is the precise statement: a device state is a collection of linear resources, disjoint resources advance independently, and uniqueness is scoped per consumed resource key.

30.1 Identity Domains

The device carries two permanent identities in hash separated domains. The online identity is derived from a BIP39 seed alone and requires no hardware:

$$I_{\text{on}} = \text{H}(\texttt{"DSM/identity/online-id/v1"} \parallel pk_{\text{on}}).$$

The offline identity exists only after an appliance birth and is the hash of an immutable anchor bundle:

$$I_{\text{off}} = \text{H}(\texttt{"DSM/identity/offline-id/v1"} \parallel B),$$

$$B = \text{H}\big(\texttt{"DSM/anchor-bundle/v2"} \parallel \text{H}(pk_{\text{on}}) \parallel \text{stpub} \parallel pk_{\text{chip}} \parallel pk_{\text{host}} \\ \parallel \text{le64}(H_0) \parallel \text{device_id} \parallel \mu_{\text{enrolled}} \parallel \text{policy_hash} \parallel S_{\text{birth}}\big).$$

The bundle binds three factors: the seed rooted DSM device keys held on the phone; a chip witness key pair generated in slot inside the secure element, non exportable, protected at rest by the die's physically unclonable function, with per die static identity stpub; and a host witness key pair generated inside the secure partition and sealed to it. Here μ_{enrolled} is the enrolled firmware measurement committed by the bundle. The host witness signing service is measurement gated: it is available only when the running firmware measurement equals μ_{enrolled}, and firmware that fails the measurement receives a hard unsealing failure with no fallback signing path. Because μ_{enrolled} is committed in B, and $I_{\text{off}} = \text{H}(\cdots B)$, this B denotes the same offline identity as the companion specification. An upgrade certificate

$$\sigma_{\text{on}} = \text{DsmSign}\big(sk_{\text{on}},\ \text{H}(\texttt{"DSM/identity/upgrade/v1"} \parallel B)\big)$$

binds I_{off} to I_{on}. Compromise of any one factor, or of any two, does not yield the offline identity. Relationships are tagged by domain at establishment. A device may operate online only for its entire life and add the offline domain later without disturbing any existing relationship; neither domain is legacy. Provisioning, the one way birth fuse, the compromise matrix, and the silicon validation of both factors on physical TROPIC01 dies are specified in the companion offline bearer specification.

30.2 The SMT State Counter

The offline domain adds exactly one structural element to the DSM global state: a monotone anchor counter committed as an SMT leaf and consumed with the whole state. Without a committed counter, the position of a state is implicit in its root history, staleness is decidable only by chain comparison, and hardware synchronization is not definable. With it, position is a scalar inside the root, and the root consumes it.

Definition 62 (Anchor State Leaf)**.** The anchor state leaf is the device SMT leaf keyed by the anchor bundle whose value commits the offline domain position:

$$L_i = \text{H}(\texttt{"DSM/anchor-state/v2"} \parallel B \parallel h_i \parallel \text{le64}(u_i)).$$

Definition 63 (Offline Frontier Root)**.** The offline frontier root is a dedicated forward only hash lineage advanced exactly once per offline bearer transfer:

$$h_{i+1} = \text{H}(\texttt{"DSM/anchor-root-advance/v2"} \parallel h_i \parallel D_{i+1}),$$

seeded at birth by h_0, where D_{i+1} is the transition digest over the canonical transition core. It is the object a counterparty durably tracks for the holder.

Definition 64 (Offline Origin)**.** The offline origin of a transfer is the tuple (R_i, h_i, u_i): the sender device root, the offline frontier root, and the anchor counter coordinate committed by the anchor state leaf of R_i. A fused implementation may merge R_i and h_i into one state root; it must still commit u_i.

Definition 65 (Counter Synchronized State)**.** Let H_0 be the enrolled physical counter value bound in B and H the live physical counter value. The appliance state is counter synchronized when $u_i = H_0 - H$ holds outside an in flight commit. Every offline commit advances both by exactly one, atomically with the relationship advance and the anchor leaf update.

The committed counter provides five things. Explicit position: u_i is proven to any receiver by an SMT inclusion proof against R_i, read from the state rather than inferred from history. Consumption with the whole state: the counter lives inside the root, so advancing it is part of the single root replacement that is the transfer, and the index cannot advance without the state or the state without the index. Staleness on sight: against any known later frontier, an older state is exposed by a scalar comparison of committed counters, with no chain walk and no reconciliation round. Fork comparability: Lemma 5 below. And the synchronization target: the physical counter of Definition 65 is meaningful only because there is a committed software register for it to track.

30.3 Offline Uniqueness Is the Kernel Theorem

Definition 66 (Offline Anchor Step Resource)**.** For a parent state s with offline origin (R_i, h_i, u_i), the anchor step resource descriptor is

$$x_{\text{anchor}} = (\texttt{"offline-anchor-step"}, B, h_i, u_i),$$

and the derived key is $K_{\text{anchor}} = \kappa_{\text{res}}(s, x_{\text{anchor}})$. Every offline bearer candidate from s consumes K_{anchor}, committed for the conflict class under rule G7 and derived under Rule 2.

Lemma 5 (Position Collision)**.** *Any two distinct offline candidates from one origin commit the same counter position $u_i + 1$ at the same leaf key under different roots, and consume the same derived key K_{anchor}. The fork is decidable by comparing the two anchor leaves, by any party holding both releases, with no reconciliation round.*

Proof. Acceptance requires the successor anchor leaf to commit exactly u_i+1 by checked arithmetic, and requires consumption of K_{anchor}, which is derived from the parent core root and x_{anchor} rather than supplied by the branch. Two distinct successors therefore present (B, h', u_i+1) and (B, h'', u_i+1) with $h' \neq h''$: two claims to one position under one bundle, decided by direct leaf comparison. □

Theorem 10 (Offline Uniqueness in Software)**.** *For every adversary, no realized history contains two distinct accepted offline successors of the same offline origin. The statement and its proof contain no hardware term.*

Proof. All offline candidates from the origin consume K_{anchor} by Definition 66, hence conflict. Realized History Uniqueness (Theorem 3) excludes two accepted successors consuming K_{anchor} from one parent occurrence. Every object examined, adopted frontiers, SMT leaves, committed counter positions, derived keys, and signatures, is software state. □

Corollary 4 (Hardware Independence of Uniqueness)**.** *Delete the anchor hardware entirely and Theorem 10 holds unchanged. Hardware evidence enters only the mode validity predicate below, where it decides device identity. It neither strengthens nor weakens exclusivity.*

30.4 Three Factor Release Witness

Let Δ°_{i+1} be the canonical transition core: action, recipient identity, object identifier, payload, old state proofs from R_i, the counter pair $(u_i, u_i + 1)$, and the receiver challenge r_R, deliberately excluding the successor root. The transition digest is

$$D_{i+1} = \mathrm{H}(\texttt{"DSM/transition-digest/v2"} \parallel \mathrm{enc}(\Delta^{\circ}_{i+1})),$$

and the dependency order is a directed acyclic graph:

$$D_{i+1} \to h_{i+1} \to L_{i+1} \to R_{i+1} \to M_{i+1} \to \sigma^{\mathrm{chip}}_{i+1},\ \sigma^{\mathrm{host}}_{i+1}.$$

The root advance message binds both roots on each side:

$$\begin{aligned} M_{i+1} = \mathrm{H}\big(&\texttt{"DSM/root-advance/v2"} \parallel B \parallel R_i \parallel R_{i+1} \parallel h_i \parallel h_{i+1} \\ &\parallel \mathrm{le64}(u_i) \parallel \mathrm{le64}(u_i + 1) \parallel D_{i+1} \parallel \mathrm{recipient} \parallel r_R\big). \end{aligned}$$

The hardware witnesses are $\sigma^{\mathrm{chip}}_{i+1}$ under the non exportable chip key and $\sigma^{\mathrm{host}}_{i+1}$ under the measurement gated partition sealed host key, both over M_{i+1}. Both hardware witnesses are minted during commit, after the counter decrement of Definition 65, not during prepare, so a signed release for an origin comes into existence only once the origin has been consumed. Factor one is the DSM transition itself, signed under seed derived device keys according to the ordinary step predicate. A release therefore proves live cooperation of phone, chip, and host over this exact transition, this exact recipient, and this exact challenge.

Definition 67 (Offline Mode Validity)**.** $\mathrm{ModeOK}_{\mathrm{off}}(s, s')$ holds iff:

(1) all encodings are canonical;

(2) the release chains from the receiver's adopted offline frontier for the holder, directly or through a branch proof of prior witnessed hops; genesis adoption additionally requires the upgrade certificate binding I_{off} to I_{on};

(3) R_i commits (B, h_i, u_i) through the anchor state leaf, with a verifying inclusion proof;

(4) D_{i+1} recomputes from Δ°_{i+1} and h_{i+1} recomputes from h_i and D_{i+1};

(5) the claimed next anchor counter is $u_i + 1$, using checked arithmetic;

(6) r_R is the challenge supplied by this receiver, and the recipient field names this receiver;

(7) M_{i+1} recomputes from the bound fields;

(8) the chip witness verifies over M_{i+1} under the chip key pinned in B;

(9) the host witness verifies over M_{i+1} under the host key pinned in B;

(10) the DSM transition proof verifies $R_i \to R_{i+1}$, including the relationship advance and the anchor leaf update;

(11) R_{i+1} commits $(B, h_{i+1}, u_i + 1)$ through the anchor state leaf;

(12) the branch consumes K_{anchor};

(13) no compromise or policy event invalidates the anchor. Where verify live policy applies, an authenticated audit read of the physical counter satisfies $H = H_0 - (u_i + 1)$ after commit; this is an audit option, not an acceptance requirement.

Theorem 11 (Offline Clone Exclusion)**.** *A party lacking any one of the three identity factors cannot produce an accepted offline release for an enrolled identity, except by forging a signature or breaking hash soundness. In particular, a software clone holding every host readable byte produces no offline release at all.*

Proof. $\text{ModeOK}_{\text{off}}$ requires the seed rooted DSM transition signatures, the chip witness under the non exportable chip key pinned in B, and the host witness under the measurement gated partition sealed key pinned in B, all over the same M_{i+1}. Absent a factor, the corresponding signature cannot be produced under Assumption 5. □

Theorem 12 (Stale Image Rejection)**.** *An appliance image restored to an earlier state cannot resume offline bearer operation.*

Proof. The restored image commits $u < H_0 - H$, since the physical counter cannot rewind below its enrolled bound. Local recovery fails closed on the mismatch, any counterparty holding a later frontier rejects its releases at condition (2) of $\text{ModeOK}_{\text{off}}$, and any authenticated audit read exposes the desynchronization. □

Theorem 13 (Full Stack Control Bound)**.** *Assume the offline evidence conditions of Assumption 5: the chip and host keys are non exportable, host signing is available only under the enrolled firmware measurement, the chip and host witnesses are minted only during commit after the counter decrement, and no perfect live state clone exists. Then an adversary holding all three factors live, including the malicious owner of an intact appliance, can choose the next valid offline successor but cannot fork one SMT anchor origin. A successful offline commit consumes the origin (R_i, h_i, u_i) and atomically advances to $(R_{i+1}, h_{i+1}, u_i + 1)$, and the same appliance cannot produce a second accepted release from (R_i, h_i, u_i), even to a disconnected first contact receiver. Conflicting accepted offline releases from one origin require a measurement boundary breach or a perfect live state clone in the sense of Assumption 5.*

Proof. A valid offline package must satisfy $\text{ModeOK}_{\text{off}}$: it must prove the predecessor anchor leaf (B, h_i, u_i), prove the successor anchor leaf $(B, h_{i+1}, u_i + 1)$, consume the derived anchor step key K_{anchor}, verify the DSM transition proof $R_i \to R_{i+1}$, and carry the chip and host witnesses over the same M_{i+1}. A second distinct release from the same origin needs a second host witness over a distinct M'_{i+1} that still binds R_i, h_i, u_i.

Under enrolled firmware, no host witness for the origin exists until commit, and commit performs the one counter decrement that moves the live counter from u_i to $u_i + 1$; afterward re pinning $H_0 - H = u_i$ fails, so a second commit at the same origin is refused and no second witness at position u_i is minted. Preparing and aborting mints nothing, so prepare, abort, and re prepare cannot harvest a second witness. Under rogue firmware, the measurement gate refuses host signing, so no host witness is produced at all and the package fails $\text{ModeOK}_{\text{off}}$. The chip witness alone is not measurement gated and can be produced by rogue firmware, but a package lacking the host witness is not accepted. The remaining routes, export of a non exportable key, extraction of a minted witness before its committing counter step, or perfect live emulation, are exactly the boundary excluded by Assumption 5. Hence a single enrolled appliance yields at most one accepted release per origin. Independently, Theorem 10 excludes folding two successors of one origin into any one realized history. □

Remark 11 (Correction of the earlier claim)**.** Earlier drafts left open a disconnected genesis fork under full stack compromise. That was too broad for the corrected offline bearer design. The correction has two load bearing parts, both carried by Assumption 5. First, host signing is gated on the enrolled firmware measurement, so replacing firmware does not yield a programmable signing oracle. Second, the chip and host witnesses are minted only at commit, after the one way counter decrement, so no witness for an origin exists while that origin is still spendable and prepare, abort, and re prepare harvests nothing. Together with the single active state, one prepared record, frontier advance, and physical counter tracker, these prevent a correct single enrolled appliance from originating two accepted releases from one SMT anchor origin, including to a disconnected first contact receiver. Full stack control of the enrolled device still gives the attacker the power to spend once, choose the valid successor, refuse service, destroy availability, or exhaust policy windows. The receiver witnessed counter reads were still wrong and remain removed: a scalar counter read is not transition authority. Only a measurement boundary breach or a perfect live state clone reopens the fork, and those are outside the model.

Remark 12 (Counter Demotion)**.** The physical counter retains three jobs, none of which is acceptance authority: a non rewind floor, a stale image tripwire, and the offline exposure cap. Prepare moves nothing and mints nothing. Commit performs exactly one counter update after re pinning $H_0 H = u_i$ with checked arithmetic, then mints the chip and host witnesses over M_{i+1}. Recovery re emits a committed release and never signs a second release for one counter step. Exhaustion at $H = 0$ forces online only operation until authority rotation.

31 Storage Nodes and Non Authority

Storage nodes do not decide DSM validity.

Definition 68 (Storage Node)**.** A storage node is a byte availability participant. It may store, mirror, index, and return DSM objects, proof material, byte commitments, and routing payloads. It is not a signer of DSM transitions and is not an acceptance authority.

Rule 7 (Storage Node Non Authority)**.** A DSM verifier must not accept a state transition merely because a storage node served it, ordered it, signed it, ranked it, or returned it first. Acceptance depends only on DSM proof material and the relevant local mode predicate.

Remark 13. This boundary is important. Storage node economics and auditability can improve availability, but they do not replace the local acceptance predicate and do not become a consensus layer.

32 Claim Boundary

The claims of this paper are scoped.

(1) Core DSM claims are claims about the guarded linear state predicate, canonical resource keys, consumed parent sets, and public verification.

(2) Online DSM claims do not require hardware anchor evidence.

(3) Offline bearer claims apply only when the optional identity anchored mode is used and the receiver verifies the three factor identity witness and the SMT committed counter positions. Offline uniqueness claims require no hardware evidence at all.

(4) Storage nodes are never authority.

(5) DSM safety is not unconditional liveness. A valid branch may fail to realize if a guard is never fulfilled, a counterparty refuses cooperation, storage is unavailable, or required mode evidence is missing.

(6) No offline only protocol distinguishes a perfect live state clone from the original device.

(7) Full stack control of the enrolled offline device, including by the malicious owner of an intact appliance, gives authority to choose and execute the next valid offline successor. Under Assumption 5, meaning host signing gated on the enrolled firmware measurement and release witnesses minted only at commit after the counter decrement, it does not let a correct single enrolled appliance originate two accepted releases from one SMT anchor origin, even to a disconnected first contact receiver. Same origin offline release duplication is outside the valid model except at the measurement boundary breach or perfect live clone boundary.

(8) Machine checked artifacts prove the abstract fork exclusion model and the specified bounded concrete models. They do not prove every production implementation detail unless that detail is represented in the artifact.

32.1 Machine Checked Scope

The Lean development proves key scoped uniqueness and Tripwire over an abstract guarded model. The TLA$^+$ development checks concrete guarded families, malformed family falsification, realized history uniqueness, and relationship scoped online no fork behavior. The displayed TLA$^+$ module treats resource keys as given inputs and guard well formedness as an opaque flag; it exercises the consumed set threading and linearity layer, but not resource key derivation or descriptor injectivity (Assumption 4), which are discharged by Rule 2 and the implementation enforcement skeleton rather than by the model checker. Offline bearer implementation safety depends on wiring the three factor witness verification and the SMT committed counter checks into the production mode predicate. No offline specific uniqueness artifact is required: offline steps are accepted through the same realized history fold that the artifacts already check.

33 Architectural Consequences

33.1 No Global Ordering Requirement

DSM does not require a global ordering service for ordinary state evolution. The verifier checks adjacency, guard fulfillment, resource key consumption, policy, structure, and mode evidence locally.

33.2 No Replace By Fee at Protocol Level

A DSM candidate is not replaced by paying a higher fee to an ordering authority. A conflicting candidate must satisfy the same resource consumption rules. If the resource is already consumed, it is invalid.

33.3 No Protocol Level Global Ordering MEV Surface

Since DSM does not expose a global mempool ordering rule for ordinary state transitions, protocol level extraction by reordering unrelated global transactions is not part of the core model.

33.4 No Consensus Inclusion Fee

DSM acceptance does not require purchase of inclusion in a block or ordering batch. Availability may have costs, but those costs do not become validity.

33.5 No Consensus Capture Attack Class

A validator set, miner majority, sequencer, or ordering cartel cannot select a different branch as the valid branch for a consumed resource because the branch is accepted by local deterministic verification.

33.6 No Protocol Level Censorship Gate

A storage node or relay can withhold bytes, but withholding is an availability failure, not a validity decision. A verifier can accept valid proof carrying bytes from any delivery path.

34 Structural Privacy

DSM does not require public broadcast of every state transition. A relationship may exchange proof carrying objects privately. Storage nodes may see encrypted or opaque bytes. A counterparty sees the proof material required for its acceptance decision. Offline bearer mode deliberately discloses additional identity evidence to the receiver because the receiver is accepting without online reconciliation.

35 Comparison to Blockchain Systems

Property	**Blockchain model**	**DSM model**
Conflict selection	Global ordering and consensus select one executed history.	Conflict local resource linearity prevents co realized branches.
Candidate branches	Conflicting transactions may be broadcast and ordered later.	Multiple futures may be pre-committed, but only one may realize for a consumed resource.
Double spend prevention	Prevented by consensus assumptions and finality rules.	Prevented in software by local deterministic verification and resource consumption, online and offline alike.
Offline transfer	Not native in base ledger model.	Optional identity anchored mode. Uniqueness is the same software theorem; hardware witnesses device identity.
Storage layer	Often part of consensus or full node replication.	Storage nodes are byte availability participants, not authority.
MEV surface	Ordering can be economically valuable.	No protocol level global ordering surface for ordinary DSM transitions.

Finality	Often probabilistic or committee dependent.	Conflict local derivability and local acceptance. Offline bearer adds identity evidence, not a second uniqueness mechanism.

36 Mapping Summary

DSM object	**Role**	**Consensus free effect**
Guard family	Defines candidate future space and fulfillment rules.	Makes possible branches explicit before realization.
Resource consumption key	Canonical lock for a consumed resource.	Conflicting futures fight over the same derived key.
Consumed parent set	Records realized consumption.	Blocks contraction and reuse.
Relationship projection	Local chain for a bilateral relationship.	Prevents cross relationship parent replay.
Policy object	Deterministic admissibility rules.	Cannot select by external ordering.
Mode evidence	Extra evidence for selected mode.	Offline bearer adds identity evidence without changing DSM core.
SMT state counter	Committed position of the offline origin.	Staleness and forks are decidable on sight by leaf comparison.
Offline bearer	Optional identity anchored mode.	Uniqueness stays in software. Hardware proves the device is the device.
Tripwire	Non derivability theorem.	Forks are invalid or exposed by reconciliation if invalid bytes are presented.

37 Main Result

Theorem 14 (Consensus Free Fork Exclusion)**.** *Under Assumptions 1 through 4, Rule 2, guard family well formedness, consumption validity, admissible policy, and valid mode evidence, DSM excludes conflict local realized forks without a global consensus layer. For any consumed resource key K, no realized history contains two distinct accepted successors from the same parent occurrence consuming K.*

Proof. The accepted transition predicate requires candidate validity, guard validity, structural validity, linearity validity, policy validity, and mode validity. Guard family well formedness ensures conflicting candidates are either exclusive at fulfillment or share the derived key of the resource they contend for, which by Assumption 4 and Lemma 1 is forced rather than assumed. Consumption validity requires the key to be absent before acceptance and present after acceptance. The realized history threads the consumed parent set forward. Therefore after one accepted successor consumes K, every later conflicting successor consuming K fails linearity. Storage nodes, message ordering, and transport do not enter the validity predicate. Hence conflict local fork exclusion is obtained without consensus. □

38 Conclusion

DSM separates possibility from realization. It permits precommitment forking, but it prohibits realized forking for a consumed resource. The mechanism is not global ordering. The mechanism is local deterministic verification over guarded candidate futures, canonical resource consumption keys, consumed parent sets, structural proofs, admissible policy, and mode specific evidence.

Offline bearer mode fits this model as an optional mode predicate, and its corrected form is simpler than earlier drafts rather than more elaborate. The SMT committed anchor counter writes the offline origin into the committed state, every valid offline release consumes the derived anchor step key, and realized history uniqueness excludes co realization offline exactly as it does online. What prevents Alice and Bob from both accepting the same offline bearer state is not a hardware read and not later network comparison. A correct enrolled appliance cannot present that state twice as two accepted releases: one atomic commit advances the SMT anchor leaf, frontier, active root, and physical counter tracker, and by the measurement gate and post decrement minting of Assumption 5 no signed release for the old origin exists to reuse, so the next attempt from the old origin is stale before it reaches receiver acceptance. If conflicting same origin offline releases exist, the anti clone or implementation assumptions have already been broken; once compared, they collide immediately at the same anchor step key and counter position and cannot be folded into any realized history. Hardware answers the one question software cannot: whether the device is the physical device that was enrolled. It answers with a three factor witness, and it answers nothing else.

The result is a consensus free, conflict local state model: multiple futures may be described, but for a fixed consumed resource, at most one may be realized.

A Lean 4 Development

The Lean development formalizes the abstract guarded model. It contains:

- an abstract type of states;
- an abstract type of resource keys;
- a step relation scoped by consumed keys;
- a history fold threading consumed keys;
- uniqueness lemmas for accepted histories;
- a Tripwire theorem showing no key scoped realized fork exists in a valid history.

The Lean core proves the fork exclusion theorem without adding the desired result as an axiom. The theorem depends on resource key uniqueness, consumed set threading, and the absence check before each accepted step.

The Lean artifacts do not attempt to prove hardware behavior, firmware correctness, TROPIC behavior, or transport liveness. Those are outside the abstract guarded model and are represented in this paper as assumptions or implementation obligations.

B TLA$^+$ Development

The TLA$^+$ development models concrete guard families and realized history update.

```
------------------------------ MODULE DSM_Guarded ------------------------------

VARIABLES consumed, ledger

Init ==
  /\ consumed = {}
  /\ ledger = {}

CanAccept(step) ==
  /\ step.keys \cap consumed = {}
  /\ step.well_formed = TRUE

Accept(step) ==
  /\ CanAccept(step)
  /\ consumed' = consumed \cup step.keys
  /\ ledger' = ledger \cup {step}

Next ==
  \E step : Accept(step)

Spec == Init /\ [][Next]_<<consumed, ledger>>

RealizedHistoryUnique ==
  \A r1, r2 \in ledger :
    (r1.parent = r2.parent /\ r1.key = r2.key) => r1.succ = r2.succ

=============================================================================
```

Two families are checked against this module. A well formed instance, in which a conflict class at one key resolves to a single successor and a disjoint branch consumes a second key, satisfies both `Safety` and `RealizedHistoryUnique`. A key split instance confirms that guard family well formedness is load bearing. A relationship scoped module encodes Remark 6. Resource keys are derived from the pair of relationship identity and parent, and each relationship has a single acceptance locus, matching bilateral online acceptance.

C Offline Bearer Interface Skeleton

```
record OfflineRelease {
  bytes canonical_transition_core;
  bytes smt_proofs;
  ReleaseCertificate certificate;
  BranchHop[] branch_proof;      // frontier catch up, optional
  UpgradeCertificate upgrade;    // genesis adoption only
}

record ReleaseCertificate {
  bytes anchor_bundle;
  bytes sender_state_before;
  bytes sender_state_after;
```

```
  bytes frontier_root_before;
  bytes frontier_root_after;
  uint64 anchor_counter;
  uint64 next_anchor_counter;
  bytes transition_digest;
  bytes root_advance_message;
  bytes chip_signature;    // non exportable, PUF protected key
  bytes host_signature;    // measurement gated partition sealed key
  bytes receiver_challenge;
  bytes recipient;
}

VerifyOfflineAdvance(x):
  require canonical(x)

  require chains_from_adopted_frontier(x)
    // direct match, or branch_proof of prior witnessed hops;
    // genesis adoption requires x.upgrade to verify

  require anchor_leaf_before_commits(
    x.sender_state_before,
    x.certificate.anchor_bundle,
    x.certificate.frontier_root_before,
    x.certificate.anchor_counter
  )

  require x.certificate.next_anchor_counter ==
          x.certificate.anchor_counter + 1

  require transition_digest_recomputes(x)

  require frontier_advance_recomputes(
    x.certificate.frontier_root_before,
    x.certificate.transition_digest,
    x.certificate.frontier_root_after
  )

  require verify_receiver_challenge(
    x.certificate.receiver_challenge,
    x.certificate.transition_digest
  )

  require root_advance_message_recomputes(x)

  require chip_verify(
    chip_key(x.certificate.anchor_bundle),
    x.certificate.root_advance_message,
    x.certificate.chip_signature
  )

  require host_verify(
    host_key(x.certificate.anchor_bundle),
    x.certificate.root_advance_message,
    x.certificate.host_signature
```

```
  )

  require verify_dsm_transition(
    x.sender_state_before,
    x.sender_state_after,
    x.certificate.transition_digest
  )

  require anchor_leaf_after_commits(
    x.sender_state_after,
    x.certificate.anchor_bundle,
    x.certificate.frontier_root_after,
    x.certificate.next_anchor_counter
  )

  require consumes_offline_anchor_step_key(x)

  accept
```

D Implementation Enforcement Skeleton

The following skeleton states the production enforcement rule for resource keys. Derived keys are computed from the parent core root and the canonical resource descriptors, and a branch supplied key is never authoritative unless it equals the derived key.

```
fn verify_candidate(parent: State, candidate: Candidate) -> Result<(), Error> {
    let derived_keys = derive_resource_keys(parent, candidate.resource_descriptors)?;
    let committed_keys = candidate.committed_consumed_keys();

    if derived_keys != committed_keys {
        return Err(Error::ResourceKeyMismatch);
    }

    if !guard_family_well_formed(parent.guard_family()) {
        return Err(Error::MalformedGuardFamily);
    }

    if conflict_class_has_shared_key_disagreement(parent.guard_family()) {
        return Err(Error::ConflictKeyDisagreement);
    }

    if consumed_set_contains_any(parent.consumed_set(), derived_keys) {
        return Err(Error::AlreadyConsumed);
    }

    verify_candidate_digest(parent, candidate)?;
    verify_guard(parent, candidate)?;
    verify_structure(parent, candidate)?;
    verify_policy(parent, candidate)?;
    verify_mode(parent, candidate)?;

    Ok(())
}
```

```
fn apply_candidate(parent: State, candidate: Candidate) -> Result<State, Error> {
    verify_candidate(parent, candidate)?;

    let keys = derive_resource_keys(parent, candidate.resource_descriptors)?;
    let next_consumed = consumed_set_insert_all(parent.consumed_set(), keys)?;

    let next = recompute_successor(parent, candidate, next_consumed)?;

    if hash(next.canonical_bytes()) != candidate.successor_digest {
        return Err(Error::SuccessorDigestMismatch);
    }

    Ok(next)
}

fn verify_offline_identity_witness(x: OfflineRelease) -> Result<(), Error> {
    let m = x.certificate.root_advance_message;

    if !chip_verify(chip_key(&x.certificate.anchor_bundle), m,
                    &x.certificate.chip_signature) {
        return Err(Error::BadChipWitness);
    }

    if !host_verify(host_key(&x.certificate.anchor_bundle), m,
                    &x.certificate.host_signature) {
        return Err(Error::BadHostWitness);
    }

    if x.certificate.next_anchor_counter != x.certificate.anchor_counter + 1 {
        return Err(Error::BadCounterStep);
    }

    // Uniqueness is not decided here. It is the kernel's consumed key
    // check over the derived anchor step key, identical to online.
    Ok(())
}
```

E Terminology

Candidate future. A precommitted possible successor state.

Candidate fork. Multiple candidate successors from the same parent. Candidate forks are permitted.

Realized fork. Multiple realized successors from the same consumed resource, folded into one realized history. Realized forks are prohibited.

Resource. The thing that produces exactly one realized successor in a conflict class.

Generation. A logical, non temporal version of a resource family.

Step. One valid realized DSM transition, evaluated against the consumed parent set of its parent.

Realized history. A fold of accepted steps threading the consumed parent set forward.

Lineage. A forward only sequence of derivable resource states, indexed independently of any single per step key.

Guard. A deterministic predicate over canonical witness and parent bytes.

Guard family. The committed collection of guards and resource key sets for a candidate future space.

Conflict class at a key. The branches from a parent whose consumption set contains a given derived key. Conflict at a fixed key is an equivalence, even though pairwise conflict need not be transitive.

Canonically resolving conflict class. A conflict class whose fulfilled branches recompute a single canonical successor, so that static uniqueness applies.

Resource descriptor. Canonical description of the linear object being consumed, injective over the resources committed by a state.

Resource consumption key. The derived per step key that marks a resource as consumed, binding the parent core root.

Branch local key. A branch specific key used for audit or indexing. It is not the exclusion key.

Consumed parent set. The canonical set of consumed resource keys committed by the DSM state.

Core root. The Sparse Merkle Tree root over the components that do not depend on the candidate future space or guard family; the layer that guards, keys, and candidate digests bind, resolving the commitment cycle.

DLV. Deterministic Limbo Vault. A concrete guarded branch system.

CPTA. Context Policy and Token Anchor. A deterministic policy object whose operations are guarded DSM candidates.

Tripwire. The non derivability theorem for co realized same parent forks in any realized history.

Offline bearer mode. Optional DSM mode for accepting a transfer without online reconciliation at the moment of exchange.

Offline origin. The tuple (R_i, h_i, u_i): sender device root, offline frontier root, and committed anchor counter coordinate. In a fused implementation, the fused state root plus u_i.

SMT state counter. The anchor counter committed as an SMT leaf, consumed with the whole state, and tracked one to one by the physical counter.

Offline frontier root. The forward only lineage advanced once per offline bearer transfer; the object a counterparty durably tracks for a holder.

Identity domain. The online, seed only, or offline, three factor fused, identity scope of a relationship.

Three factor witness. The seed rooted DSM signatures, the chip signature, and the measurement gated host signature over one root advance message.

Measurement gate. The condition that host signing is available only under the enrolled firmware measurement, so rogue firmware cannot mint the host witness.

Exposure cap. The policy bound W on offline commits between reconciliations.

Storage node. A byte availability participant. It is not a DSM authority.

Device ID. The device identifier used for device binding.

Node ID. The storage node identifier used for storage infrastructure.

F References

1. Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.
2. David Chaum. Blind Signatures for Untraceable Payments. Advances in Cryptology, Proceedings of CRYPTO 1982. Springer, 1983, pp. 199 to 203.
3. David Chaum, Amos Fiat, and Moni Naor. Untraceable Electronic Cash. Advances in Cryptology, CRYPTO 1988. Lecture Notes in Computer Science 403. Springer, 1990, pp. 319 to 327.
4. David Chaum and Torben Pryds Pedersen. Wallet Databases with Observers. Advances in Cryptology, CRYPTO 1992. Lecture Notes in Computer Science 740. Springer, 1993, pp. 89 to 105.
5. Stefan Brands. Untraceable Off-Line Cash in Wallet with Observers. Advances in Cryptology, CRYPTO 1993. Lecture Notes in Computer Science 773. Springer, 1994, pp. 302 to 318.
6. Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. 2016.
7. Raphael Auer and Rainer Boehme. The Technology of Retail Central Bank Digital Currency. BIS Working Paper, Bank for International Settlements. 2020.
8. Leon Kempen and Johan Pouwelse. Offline Digital Euro: a Minimum Viable CBDC using Groth-Sahai proofs. arXiv:2407.13776. 2024.
9. Leslie Lamport. The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems. 1994.
10. Jean-Yves Girard. Linear Logic. Theoretical Computer Science. 1987.
11. Daniel J. Bernstein et al. SPHINCS+ Submission to the NIST Post-Quantum Cryptography Project.
12. Roberto Avanzi et al. CRYSTALS Kyber Algorithm Specifications and Supporting Documentation.
13. Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. BLAKE3.

14. Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. 1987.

15. Brandon Ramsay. Software Authority, Hardware Identity: Double Spend Exclusion in Software for DSM Offline Bearer State. Irrefutable Labs Inc. 2026.

16. DSM project specifications and implementation notes. Irrefutable Labs Inc. 2025 to 2026.