Native Eltoo: A Consensus-Embedded Protocol for High-Frequency Payment Networks

Arthur Zhang* — *Tondi Foundation*
Neo Maxwell — *Tondi Foundation*
Yunli Zhu — *DI Foundation*
Yunxuan Zheng — *Tondi Foundation*

December, 2025

Abstract

Payment Channel Networks (PCN) constitute a Layer 2 scaling solution for blockchain systems, whose core principle is to complete multiple state updates off-chain while settling only the final state on-chain, thereby improving system throughput.

Background and Problem Definition: Existing PCN schemes (such as the Lightning Network) exhibit two categories of structural limitations: (1) the expressiveness of the Script Layer is constrained, making it difficult to natively support complex state transition semantics; (2) linear topology structures lead to low capital utilization and excessive state management complexity. While the original Eltoo protocol proposed a state replacement mechanism to supersede the penalty mode, its implementation depends on the not-yet-activated `SIGHASH_ANYPREVOUT` soft fork (BIP-118) and presents security concerns such as replay attacks.

Technical Contributions: This paper proposes a general-purpose scaling architecture based on native Eltoo semantics. The main contributions include:

1. **Dual-Track UTXO Model:** Decomposing channel state into a static fund anchor (Fund UTXO) and a dynamic state pointer (State UTXO) along two orthogonal dimensions, achieving separation of concerns between value transfer and state transitions;
2. **Reference-Based UTXO Primitive:** Defining a read-only reference operator $\mathrm{Ref} : \mathcal{U} \to \mathcal{U}^{readonly}$, enabling state update transactions to access fund anchor metadata without consuming that UTXO;
3. **Transaction Type Enumeration System:** Embedding algebraic data types at the consensus layer to achieve $\mathcal{O}(1)$ complexity for transaction classification and verification;
4. **Recursive Channel Factories and Atomic Reconfiguration:** Formally defining channel splitting (Splice-Fork) and merging (Splice-Merge) operations, proving that any complex topology can achieve isomorphic transformation through a single atomic transaction.

Theoretical Results: This paper proves the existence of a bijective mapping between UTXO sets and channel state sets (Theorem 8), thereby eliminating dependency on off-chain registries. Under DAG-structured consensus protocols, state verification complexity is $\mathcal{O}(1)$, with settlement latency reaching sub-second levels.

Keywords: Payment Channel Networks, State Channels, Eltoo Protocol, UTXO Model, Finite State Machine, Formal Verification, Layer 2 Scaling

*Corresponding author: Arthur Zhang (arthur@tondi.org)

Contents

1 Introduction and Motivation

1.1 Problem Background

The core design objective of payment channel networks is to transfer transaction processing from on-chain to off-chain while maintaining security guarantees. Achieving this objective faces the following fundamental challenges:

1. **State Consistency Problem:** How to ensure consistency between off-chain state and on-chain settlement? Off-chain states evolve rapidly, but only the final state should be enforceable on-chain.

2. **Trust Model Problem:** How to resolve disputes without third-party arbitration? Traditional approaches rely on game-theoretic punishment mechanisms, but these introduce new vulnerabilities.

3. **Storage Overhead Problem (Toxic Waste):** Penalty-based mechanisms require permanent storage of all historical revocation keys. Any data loss exposes honest parties to fund theft, creating an unbounded storage liability that grows linearly with channel lifetime.

4. **Protocol Dependency Problem:** Existing state replacement solutions (e.g., original Eltoo) require consensus-layer changes such as `SIGHASH_ANYPREVOUT` (BIP-118), creating deployment barriers and cross-chain incompatibility.

5. **Liveness and Availability Problem:** Payment channels demand continuous monitoring to detect and respond to malicious state broadcasts. Watchtower delegation introduces additional trust assumptions and operational costs.

6. **Verification Complexity Problem:** Script-based channel protocols push substantial verification logic into on-chain execution, increasing transaction weight, fee costs, and potential attack surfaces from script interpreter vulnerabilities.

7. **Base Layer Throughput and Finality Problem:** Bitcoin's approximately 10-minute block interval and probabilistic finality create fundamental tension with payment channel requirements. Channel operations (opening, closing, dispute resolution) compete for scarce block space, while the lack of deterministic finality introduces uncertainty in settlement guarantees. These constraints are not incidental but intrinsic to Bitcoin's security model.

1.2 The Layer Inversion Insight

The enumerated problems share a common root cause: **attempting to retrofit Layer 2 semantics onto a Layer 1 designed without consideration for off-chain protocols**. Bitcoin's architecture optimizes for a specific set of properties—censorship resistance, minimal trust assumptions, and simplicity—but these design choices create friction when supporting payment channel networks.

This observation leads to a philosophical reframing:

> *Rather than adapting payment channels to fit an existing base layer, we should ask: what would a base layer look like if it were designed from inception to natively support state channel topologies?*

This paper pursues this question to its logical conclusion. We propose an architecture that inverts the traditional layering relationship: instead of treating L2 as an afterthought bolted onto L1, we design L1 and L2 as a **co-optimized system** where:

- The consensus layer provides native primitives for state channel operations (not script-level simulation)
- The UTXO model is extended with reference semantics specifically for dual-track state representation
- A GhostDAG-based consensus enables high-throughput, low-cost on-chain state checkpointing

The choice of GhostDAG consensus is not incidental but architecturally motivated. DAG-structured block production permits concurrent block creation, yielding sub-second block intervals and massive throughput increases over linear chains. This transforms the economics of on-chain interaction: channel participants can affordably record state checkpoints on-chain, creating cryptographic evidence that bounds the window of vulnerability to stale-state attacks.

Crucially, while DAG consensus introduces the possibility of deeper reorganizations compared to single-chain protocols, the security properties *converge faster in wall-clock time*. Under equivalent elapsed time, a GhostDAG chain accumulates more confirming blocks than Bitcoin, and the cost of mounting a reorganization attack scales with the *number of concurrent blocks* an adversary must produce—not merely hash power. After a modest number of confirmations, state checkpoint UTXOs achieve settlement guarantees that match or exceed Bitcoin's, while providing orders-of-magnitude faster initial confirmation.

This is not merely an engineering optimization but a **paradigm shift**: from "How do we build channels on Bitcoin?" to "How do we build a base layer for channels?" The resulting architecture eliminates entire categories of problems rather than mitigating them.

1.3 Design Principles

The dual-track state machine architecture proposed in this paper operationalizes the layer inversion insight through the following design principles:

Principle 1: Consensus-Layer Native Semantics Embed channel operation semantics within consensus rules rather than simulating through the script layer. This provides two advantages:

- Verification complexity reduces from $\mathcal{O}(\text{script\_size})$ to $\mathcal{O}(1)$
- Eliminates uncertainty introduced by script interpreters

Principle 2: Orthogonal Separation of Value and State Decompose channel representation into two independent dimensions:

- **Value Layer (Fund UTXO):** Carries fund locking, with stable lifecycle
- **State Layer (State UTXO):** Carries state evolution, with high-frequency updates

This separation ensures that state updates need not touch the fund locking structure, reducing verification complexity.

Principle 3: Non-Punitive State Replacement Replace penalty-based revocation with monotonic state supersession. Any party can publish a newer state to override an older one, eliminating the need for:

- Permanent storage of revocation secrets (toxic waste elimination)
- Complex punishment transaction graphs
- Asymmetric information advantages between channel participants

Principle 4: Deterministic State Execution Traditional contract execution relies on ex post enforcement (through arbitration), introducing cost and time uncertainty. The Native Eltoo achieves ex ante enforcement through consensus rules:

$$\text{Traditional Mode: Contract} \xrightarrow{\text{Dispute}} \text{Arbitration} \xrightarrow{\text{Judgment}} \text{Enforcement}$$

$$\text{The Native Eltoo: State\_UTXO} \xrightarrow{\tau_{\text{settle}}} \text{Value\_Distribution} \quad \text{(deterministic execution)}$$

Principle 5: Minimal External Dependency The protocol operates without requiring:

- External registries or naming services for channel discovery
- Continuous watchtower availability (degraded to optional optimization)
- Specific soft-fork upgrades (e.g., `SIGHASH_ANYPREVOUT`)

Fund UTXO serves as the sole cryptographic anchor, making channel existence self-evident from on-chain state.

1.4 Trust Model Analysis

Blockchain system security is often described as "trust minimization." The Native Eltoo further pursues **trust elimination**—making certain types of trust assumptions unnecessary through protocol design:

Table 1: Trust Model Comparison

Trust Assumption	**Traditional PCN**	**The Native Eltoo**	**Elimination Mechanism**
Channel registry availability	Required	Not required	Fund UTXO as sole anchor
Watchtower continuous online	Strong dependency	Weak dependency	Long-period timelocks + state replacement
Script interpreter correctness	Required	Not required	Consensus-layer native types

The core insight of The Native Eltoo is: by pushing complexity down to the protocol layer, a simpler trust model can be achieved at the application layer.

2 Related Work and Technical Background

This section provides a comprehensive analysis of the evolution of payment channel protocols, identifies the structural defects inherent in existing solutions, and establishes the technical foundation for The Native Eltoo. For formal definitions of cryptographic primitives and notation conventions, readers are referred to Appendix **??**.

2.1 Protocol Evolution: From Penalty to Replacement

Payment channel protocols have evolved through two distinct paradigms: **penalty-based revocation** and **state replacement**. Understanding the fundamental differences between these approaches—and why neither fully solves the problem—is essential for motivating the architectural innovations presented in this paper.

2.1.1 Lightning Network's Penalty Mechanism

The Lightning Network [**?**] pioneered the practical deployment of payment channels on Bitcoin. Its security model relies on a **penalty mechanism** to deter participants from broadcasting outdated channel states.

Mechanism Overview. Consider a channel between parties Alice and Bob with state sequence $S_0, S_1, \ldots, S_n$. When transitioning from state S_n to S_{n+1}, both parties perform a key exchange: Alice reveals her "revocation key" for S_n to Bob, and vice versa. This revocation key enables the counterparty to claim *all* funds in the channel if an outdated state is broadcast.

Formal Security Property. Let $\mathcal{R}_i^A$ denote the revocation key that Alice holds for state S_i, which would allow her to punish Bob if he broadcasts S_i after S_{i+1} has been agreed upon. The security guarantee can be expressed as:

$$\forall i < n : \mathcal{R}_i^A \in \text{Alice's\_storage} \implies \text{Bob broadcasting } S_i \text{ results in total fund loss for Bob}$$

This creates a game-theoretic equilibrium: rational actors will never broadcast old states because the expected punishment exceeds any potential gain.

Structural Defects (The "Toxic Waste" Problem). Despite its elegance, the penalty mechanism introduces fundamental operational burdens:

1. **Unbounded Storage Growth:** Each state update generates a new revocation key that must be stored permanently. For a channel with n state transitions, nodes must maintain $\mathcal{O}(n)$ historical keys. Long-lived, high-frequency channels can accumulate millions of revocation secrets.

2. **Catastrophic Failure Mode:** Data loss is not merely inconvenient—it is catastrophic. If Alice loses her revocation keys and inadvertently broadcasts an old state (e.g., from a corrupted backup), Bob can legitimately claim all funds. This transforms routine operational failures into complete fund loss.

3. **Backup Complexity:** The requirement to never lose *any* historical key while *also* never using an old backup (which might trigger a penalty) creates a paradoxical backup strategy. Users must maintain perfectly synchronized, append-only storage with no tolerance for errors.

The term "toxic waste" aptly describes this accumulation: revocation keys are useless during normal operation but potentially deadly if mishandled, and they can never be safely discarded.

2.1.2 Eltoo Protocol and State Replacement

The Eltoo protocol [?] represents a fundamental paradigm shift from punishment to **state replacement**. Rather than penalizing old states, Eltoo allows newer states to simply supersede older ones through a clever use of transaction structure.

Core Mechanism. In Eltoo, each state S_n corresponds to an "update transaction" τ_n. The key innovation is that τ_{n+1} can validly spend the output of *any* prior update transaction τ_i where $i \leq n$. This is achieved through a signature scheme that does not commit to the specific input being spent.

State Supersession Semantics. If a malicious party broadcasts an old state τ_i, the honest party can respond by broadcasting τ_n (the latest state), which will:

1. Spend the output of τ_i as its input

2. Replace the outdated state with the current one

3. Require no punishment—the old state is simply overwritten

Protocol Dependency on SIGHASH_ANYPREVOUT. The Eltoo mechanism requires signatures that bind to the *script logic* of an input rather than its specific identity (the OutPoint, which consists of a transaction ID and output index). This is formalized in BIP-118 [?] as the `SIGHASH_ANYPREVOUT` flag. The signature hash is computed as:

$$h_{\text{APO}} = H(\tau.\text{outputs} \parallel \tau.\text{inputs}[i].\text{script} \parallel \tau.\text{version} \parallel \dots)$$

Notably, this hash *omits* the OutPoint identifier of the input being spent. This allows a single pre-signed transaction to spend from any UTXO that matches the expected script, enabling the state replacement mechanism.

Advantages over Penalty Mechanism:

- **Constant Storage:** Only the latest state needs to be stored; historical states can be safely discarded.

- **Symmetric Information:** Both parties hold identical transaction sets, eliminating information asymmetry.

- **Simpler Backup:** Restoring from an old backup does not risk fund loss—the counterparty simply publishes the newer state.

2.1.3 Engineering Compromise of BIP-118

While Eltoo's state replacement mechanism is theoretically elegant, its implementation via BIP-118 introduces significant engineering challenges and security concerns.

The Replay Attack Vector. The flexibility of `ANYPREVOUT` signatures creates a dangerous attack surface: if a signature is created for one UTXO, it can potentially be "replayed" against any other UTXO with a matching script. This means that careless key reuse across different channels—or even within the same channel after a splice operation—could allow an attacker to redirect funds.

Mitigation via Public Key Tagging. To address this, BIP-118 mandates a **physical segregation** of public keys used for ANYPREVOUT operations. The verification logic becomes:

$$\texttt{Verify}_{\text{APO}}(\sigma, m, P) = \begin{cases} \texttt{FALSE} & \text{if } P \in \mathcal{K}_{\text{std}} \text{ (standard key space)} \\ \texttt{SchnorrVerify}(\sigma, m, P) & \text{if } P \in \mathcal{K}_{\text{apo}} \text{ (APO-tagged key space)} \end{cases}$$

This means APO signatures are only valid when used with specially-tagged public keys that belong to a distinct derivation path. Standard keys cannot be used with APO, and vice versa.

The Shifted Security Burden. While this tagging scheme prevents *accidental* cross-contamination, it fundamentally **shifts security responsibility from the protocol layer to the application layer**:

- Wallet developers must implement correct key derivation paths
- Users must never reuse keys across different contexts
- A single implementation bug or user error can lead to complete fund loss

This violates the principle of defense in depth: the protocol should prevent dangerous operations, not merely trust applications to avoid them.

2.2 The Recursive Covenant Dilemma

Beyond its immediate security concerns, `SIGHASH_ANYPREVOUT` introduces a more fundamental philosophical debate within the Bitcoin community regarding **recursive covenants**.

What are Covenants? A covenant is a spending condition that restricts not just *who* can spend a UTXO, but *how* it can be spent—specifically, what form the spending transaction's outputs must take. Traditional Bitcoin scripts answer the question "who holds the key?" Covenants answer the question "what happens to the funds next?"

The Recursion Problem. ANYPREVOUT enables a form of transaction introspection: a script S can effectively examine properties of the spending transaction. If S can enforce that its output must be locked by a script S' where $S' \equiv S$ (or a controlled variant), then funds can be **permanently constrained** to follow specific paths indefinitely.

Formally, let $\mathcal{C}(S)$ denote the set of valid spending patterns from script S:

$$\text{Recursive Covenant: } \exists S : \forall \tau \in \mathcal{C}(S), \tau.\text{outputs} \subseteq \{\text{UTXOs locked by } S\}$$

Community Concerns. This capability raises serious objections:

1. **Fungibility Risk:** Coins could be "tainted" with permanent restrictions (e.g., regulatory whitelists), creating second-class bitcoins that are worth less than unrestricted coins.
2. **Censorship Vectors:** Governments could mandate that exchanges only accept coins locked in compliant covenants, fragmenting the monetary supply.
3. **Unintended Complexity:** Recursive covenants could enable Turing-complete computation on-chain, fundamentally changing Bitcoin's computational model.

Impact on Protocol Adoption. These concerns have contributed to the prolonged stalling of BIP-118 activation. The Bitcoin community's conservative approach to consensus changes means that any feature with potential for "toxic recursion" faces significant resistance, regardless of its benefits for payment channels.

This political reality underscores the value of our proposed architecture: by achieving Eltoo semantics *without* relying on ANYPREVOUT, we sidestep this entire debate.

2.3 BIP-118 Security Boundary Analysis

To concretize the security concerns discussed abstractly above, we analyze the specific attack surface created by BIP-118's design choices. Figure **??** illustrates how security responsibility "leaks" from the consensus layer to the application layer.

The Core Vulnerability. The fundamental issue is **implicit state dependency**: the security of an ANYPREVOUT signature depends on context that is not visible to the signature verification algorithm. Specifically:

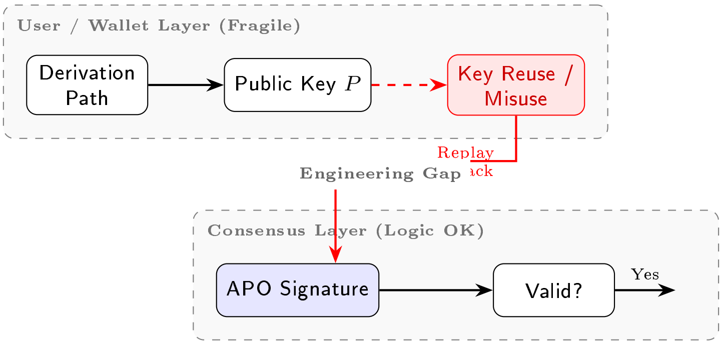


Figure 1: BIP-118 Security Boundary Leakage. The consensus layer correctly verifies APO signatures, but security ultimately depends on correct key derivation paths implemented in wallets. Key reuse between standard and APO contexts enables replay attacks despite sound protocol logic. The “Engineering Gap” represents the trust assumption that applications will not make mistakes.

1. A signature σ is created for a particular channel state
2. The signature is valid for *any* UTXO with matching script
3. If the same key pair is used elsewhere (different channel, post-splice, etc.), σ can be “replayed” to steal funds

The consensus layer cannot detect this attack because each individual signature verification succeeds. The attack exploits the *relationship* between multiple valid operations, which is invisible at the transaction level.

Contrast with Proposed Architecture. Our approach eliminates this vulnerability through type-level isolation:

- Each channel’s Fund UTXO has a unique identifier embedded in the State UTXO
- The Ref operator cryptographically binds state updates to specific fund anchors
- Replay is impossible because the reference check would fail for any UTXO other than the intended target

This represents a fundamental design principle: **security invariants should be enforced by the protocol, not assumed of applications**.

2.4 Structural Defect Analysis

Having examined the specific protocols, we now abstract to identify **structural defects** common to script-based payment channel implementations. These defects arise from a fundamental mismatch: attempting to simulate state machine semantics using a language (Bitcoin Script) designed for simple predicate evaluation.

The Orthogonality Principle. Well-designed systems maintain separation between independent concerns. In payment channels, two orthogonal concerns exist:

- **Value Custody:** Which keys control the funds?
- **State Evolution:** What is the current balance distribution, and how does it change?

Script-based solutions **violate orthogonality** by encoding state logic directly into spending conditions, entangling these concerns.

Table 2: Structural Defects of Script-Based Payment Channel Solutions

Defect Category	**Technical Impact and Consequences**
Verification Complexity	Transaction validation requires executing the full script, resulting in $\mathcal{O}(script\_size)$ computational cost. Script size is unbounded and unpredictable, making resource consumption difficult to estimate and enabling potential DoS vectors.
Semantic Opacity	The consensus layer cannot distinguish between a channel state update and a regular fund transfer—both appear as script executions. This prevents the base layer from optimizing for channel operations or providing specialized guarantees.
Blurred Security Boundaries	Security depends on application-layer behaviors (key management, derivation paths) rather than protocol-level invariants. The consensus layer provides mechanisms but not policies, shifting liability to wallet implementations.
Concern Coupling	Value locking logic (multi-signatures) and state transition logic (sequence numbers, timelocks) are interleaved in the same script. Modifying one concern risks breaking the other.

The Root Cause. These defects share a common origin: script-based solutions treat the blockchain as a *general-purpose computation platform* and simulate channel semantics through clever encoding. Our architecture inverts this relationship: the blockchain natively understands channel operations, and verification becomes a matter of type checking rather than program execution.

2.5 Comparison of Revocation Mechanisms

Table **??** provides a systematic comparison between the original Eltoo protocol (which depends on BIP-118) and The Native Eltoo. For clarity, we refer to The Native Eltoo as **Native Eltoo** throughout this paper, reflecting its role as a successor that achieves Eltoo's design goals through consensus-native mechanisms rather than script-level simulation.

Table 3: State Revocation Mechanisms: Original Eltoo vs. Native Eltoo (Proposed Architecture)

Feature	**Original Eltoo (BIP-118)**	**Native Eltoo (This Paper)**
Core Primitive	Script-level overwrite using `SIGHASH_NOINPUT`	Consensus-layer transaction types with dual-track UTXO model
Scalability Model	Complex script logic limits factory size	Native channel factory support with $\mathcal{O}(1)$ verification
Data Requirements	Full transaction parsing and script execution	Store only latest state; constant-size verification
Determinism	Signature-dependent; requires pre-signing all possible paths	Virtual reference enables deterministic state lookup
DoS Resistance	Weak; script execution is unbounded	Strong; State-Type-Protected Commitment (STPC) strategy

Key Architectural Differences:

- **Abstraction Level:** Original Eltoo operates at the script layer, encoding state machine logic in spending conditions. Native Eltoo operates at the transaction type layer, where state transitions are first-class operations.
- **Security Model:** Original Eltoo's security depends on correct application-layer key man-

agement. Native Eltoo's security is enforced by consensus rules—applications cannot violate invariants even if buggy.

- **Deployment Path:** Original Eltoo requires a Bitcoin soft fork (BIP-118) with uncertain timeline. Native Eltoo is designed for new chains built with native support, avoiding political obstacles.

2.6 Proposed Solution: UTXO-Native Semantics

The defects enumerated above motivate our core architectural decision: rather than simulating Eltoo semantics through script-level tricks, we propose **embedding these semantics directly into the UTXO model and transaction type system**.

Design Philosophy. The key insight is that payment channel operations are not arbitrary computations—they follow a small, well-defined set of patterns. By making these patterns *first-class citizens* of the transaction model, we can:

1. Verify operations through type checking rather than script execution
2. Enforce invariants at the consensus layer rather than trusting applications
3. Enable the base layer to optimize specifically for channel workloads

Transaction Type System. We introduce a typed transaction model where each transaction declares its semantic intent:

- **Type Enforcement:** For all transactions τ in the Eltoo transaction set $\mathcal{T}_{\text{Eltoo}}$, the inputs MUST be of type `ELTOO_STATE`. This is not a script-level check but a consensus-level type constraint.
- **Replay Isolation:** Because `ELTOO_STATE` UTXOs can only be spent by Eltoo update transactions, and standard UTXOs cannot be inputs to such transactions, replay attacks are *physically impossible* at the type level—no key management discipline required.

FSM vs. Arbitrary Recursion. The Eltoo state machine follows strict monotonicity: each update transaction carries a sequence number n, and valid transitions require $n' > n$. This constraint is enforced by consensus:

$$\forall \tau_{\text{update}} : \tau_{\text{update}}.\text{output\_seq} > \tau_{\text{update}}.\text{input\_seq}$$

This monotonicity property *mathematically precludes* arbitrary recursive covenants. The state can only move forward, never loop or branch arbitrarily. We gain the benefits of state replacement without opening the door to Turing-complete on-chain computation.

Explicit Reference Operator. The Ref primitive enables a transaction to *read* a UTXO's metadata without consuming it:

$$\text{Ref}(U) \in \tau.\text{references} \implies U \in \mathcal{U}_{\text{post-}\tau}$$

This reduces verification complexity from $\mathcal{O}(\text{script\_size})$ to $\mathcal{O}(1)$: the verifier checks that the referenced UTXO exists and has the expected properties, rather than executing arbitrary code.

Table 4: Core Axiom System for Dual-Track State Channels

Axiom	Formal Expression and Semantic Interpretation
A1	$\mathcal{S}_{\text{channel}} \cong \mathcal{U}_{\text{chain}}$ *Isomorphism:* There exists a bijective mapping between the set of valid channel states and the set of on-chain UTXO configurations. Every channel state corresponds to exactly one UTXO pattern, and vice versa. This eliminates the need for off-chain registries.
A2	$\forall \tau_{\text{update}} : n' > n$ *Strict Monotonicity:* Every state update transaction must carry a sequence number strictly greater than its input. This ensures unidirectional state evolution and prevents replay of old states.
A3	$\text{Ref}(U) \in \tau \implies U \in \mathcal{U}_{\text{post}}$ *Non-Consumption:* When a transaction references a UTXO via the Ref operator, that UTXO remains unspent after the transaction is processed. References provide read-only access without affecting ownership.
A4	$\sum V_{\text{in}} = \sum V_{\text{out}} + \delta_{\text{fee}}$ *Value Conservation:* The sum of input values equals the sum of output values plus transaction fees. No value is created or destroyed during state transitions.

2.7 Axiom System

The Native Eltoo is built upon a minimal set of axioms that capture the essential properties of sound payment channel operation. These axioms serve as the foundation for formal verification and guide implementation decisions.

Axiom Interdependencies. These axioms are not independent; they form a coherent system:

- A1 (Isomorphism) depends on A3 (Non-Consumption): the ability to reference without consuming is what enables the dual-track structure that makes isomorphism possible.
- A2 (Monotonicity) enforces the finite state machine semantics that prevent recursive covenants.
- A4 (Conservation) is the standard UTXO invariant, extended to account for the reference mechanism.

2.8 DAG Consensus Compatibility

The Native Eltoo is designed for deployment on GhostDAG-based consensus protocols [**?**, **?**]. This choice is not incidental but architecturally motivated by the synergy between DAG properties and payment channel requirements.

Why GhostDAG? Traditional blockchain consensus produces a single chain of blocks, where concurrent blocks become “orphans” that are discarded. GhostDAG instead organizes blocks into a Directed Acyclic Graph, where concurrent blocks can *all* contribute to consensus.

Given network parameters (D, k) where D is the maximum network delay bound and k is the security parameter, GhostDAG provides:

1. **Temporal Consistency via DAA Score:** The Difficulty Adjustment Algorithm (DAA) Score provides a globally monotonic logical clock. For any two blocks $b_1 \prec b_2$ in the DAG partial order:
$$\text{DAA}(b_1) < \text{DAA}(b_2)$$
This property enables precise relative timelocks for channel dispute periods without relying on wall-clock time.

2. **Fast Confirmation:** Block confirmation time is determined by network propagation delay rather than artificial throttling. The expected confirmation latency is:

$$E[\text{confirmation\_time}] = \Theta\left(\frac{D}{k}\right)$$

   For typical parameters ($D \approx 2\text{s}$, $k \approx 16$), this yields sub-second initial confirmation—orders of magnitude faster than Bitcoin's 10-minute blocks.

3. **Throughput Scaling:** Because concurrent blocks are not wasted, effective throughput scales with the block production rate:

$$\text{TPS}_{\text{effective}} \approx k \times \text{TPS}_{\text{single-chain}}$$

   This high throughput makes on-chain state checkpointing economically viable.

Implications for Payment Channels. The combination of fast confirmation and high throughput fundamentally changes the design space for payment channels:

- State checkpoints can be recorded on-chain affordably, reducing reliance on watchtowers
- Dispute periods can be shorter (in wall-clock time) while providing equivalent security
- Channel factories become more practical due to reduced coordination overhead

2.9 Economic Efficiency Boundary

Beyond security considerations, Layer 2 protocols must be evaluated on their **economic efficiency**—the ability to provide fast, cheap transactions without excessive capital lockup or operational overhead. We analyze protocols in a three-dimensional design space:

$$\Omega = \mathcal{L} \times \mathcal{T} \times \mathcal{C}$$

Dimension Definitions:

- **Latency ($\mathcal{L}$):** Time from payment initiation to final settlement guarantee. Lower is better.
- **Throughput ($\mathcal{T}$):** Maximum transaction processing rate. Higher is better.
- **Capital Efficiency ($\mathcal{C}$):** Ratio of usable liquidity to locked capital. Higher means less "dead" capital.

The Capital Efficiency Problem. Traditional payment channels suffer from **bilateral capital fragmentation**: if Alice has channels with Bob and Carol, her capital is split between them. She cannot use her Bob-channel balance to pay Carol directly without routing, which incurs fees and may fail due to liquidity constraints.

Channel factories address this by **pooling capital** among multiple participants. However, on Bitcoin, factory operations are expensive due to high on-chain fees and slow confirmation. This limits practical factory sizes.

Native Eltoo's Economic Advantage. Our architecture enables larger, more dynamic channel factories through:

1. **Low-cost reconfiguration:** Factory membership changes require only $\mathcal{O}(1)$ on-chain operations, not $\mathcal{O}(n)$ bilateral channel updates.
2. **Fast finality:** GhostDAG's sub-second confirmation enables rapid factory operations without long waiting periods.

Table 5: Economic Positioning of Layer 2 Protocols in the Ω Design Space

Protocol Class	**Latency**	**Capital Efficiency**	**Trust Model**
Bitcoin L1	10–60 minutes (probabilistic finality)	Baseline (no lockup)	Full decentralized consensus
Optimistic Rollups	1–15 minutes (challenge period)	Medium (operator collateral)	Operator honesty + L1 fallback
Lightning Network	Sub-second (off-chain)	Low (bilateral lockup)	Watchtower availability
Native Eltoo (Proposed)	Sub-second (off-chain)	High (pooled via factories)	Consensus-enforced invariants

3. **Reduced time-value cost:** The economic cost of locked capital is $\gamma \cdot C \cdot t$, where γ is the opportunity cost rate, C is locked capital, and t is lock duration. Faster operations reduce t, directly improving capital efficiency.

For high-frequency payment flows (micropayments, streaming payments, gaming), these improvements compound: smaller latency enables tighter capital management, which reduces opportunity costs, which enables smaller margins, which attracts more volume.

3 Research Contributions

Traditional payment channel networks are constrained by linear topologies and face two major challenges: **state synchronization complexity** and **toxic waste from penalty mechanisms**. This paper proposes a **Dual-Track State Machine** architecture that resolves these limitations through consensus-layer native transaction types.

3.1 Main Contributions

1. **Formalized State Machine Model**: We define payment channels as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, enabling formal verification via TLA+ and Coq.
2. **Registry-Free Architecture**: By designing RefOp-Fund semantics, we eliminate dependencies on external state registries.
3. **Recursive Isolation**: We formally prove the orthogonality between sub-channel security and parent channel liveness.
4. **Topological Invariants**: We prove value conservation and state monotonicity invariants for complex networks.
5. **Constant-Time Verification**: We achieve $\mathcal{O}(1)$ PTLC verification by deriving keys directly from the Fund UTXO.
6. **Complete Protocol Specification**: We provide a consensus-layer specification ready for implementation.

3.2 Theoretical Significance

This work elevates channel design from *ex post penalty games* to *ex ante deterministic execution.*

Table 6: Paradigm Shift in Design Philosophy

Aspect	**Traditional**	**Proposed**
Trust Model	Penalty Deterrence	**Protocol Determinism**
State Mgmt	App Layer	**Consensus Layer**
Security Basis	User Key Management	**Protocol Rule Enforcement**
Complexity Locus	Application Layer	**Protocol Layer (L1)**

By centralizing complexity at the protocol layer, we achieve simplicity at the application layer.

3.3 Architectural Advantages

- **Orthogonal Separation**: Static Fund vs. Dynamic State.
- **Type Safety**: Compile-time guarantees via algebraic types.
- **Constant Complexity**: $\mathcal{O}(1)$ for storage and verification.
- **Topological Freedom**: Atomic splicing enables fractal networks.

3.4 Comparison with Existing Solutions

Table **??** presents a comprehensive comparison.

Table 7: Comprehensive Architecture Comparison

Feature	**Lightning**	**BIP-118**	**Native Eltoo**
Consensus	None	Soft Fork	**Native**
State Rep.	Script+HTLC	Script	**UTXO Enum**
Model	Coupled	Coupled	**Dual-Track**
Type Safety	Runtime	Runtime	**Compile-time**
Complexity	$\mathcal{O}$(script)	$\mathcal{O}$(script)	$\mathcal{O}(1)$
Storage	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Settle Time	Minutes	Minutes	**Sub-second**
Backup	Full History	Latest	**Latest**

3.5 Information-Theoretic Analysis of State Determinism

Traditional mechanisms (e.g., Poon-Dryja) rely on penalty deterrence. Verifying state S_t requires knowledge of all revoked states $\{S_0, \dots, S_{t-1}\}$.

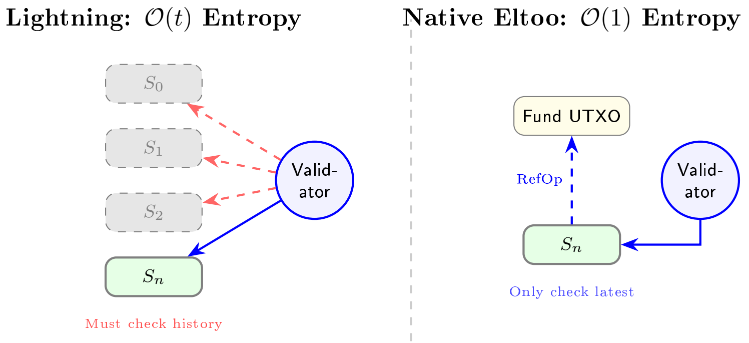
Definition 3.1 (State Entropy). *The* ***state entropy*** $H(C)$ *of a channel* C *is defined as the information quantity a validator must maintain to verify state validity. For the Lightning Network after* t *state transitions:*

$$H_{\text{LN}}(t) = \sum_{i=0}^{t-1} |\text{RevocationKey}_i| = \mathcal{O}(t) \tag{1}$$

where $|\cdot|$ *denotes the size in bits.*

This linear entropy growth leads to **watchtower inflation** and **catastrophic recovery failure** (toxic waste). In contrast, our architecture utilizes UTXO atomicity for protocol-level state replacement, collapsing entropy to a constant:

$$H_{\text{NativeEltoo}}(t) \approx \text{size}(\text{State}_{\text{curr}}) + \text{size}(\text{Fund}) \approx \mathcal{O}(1) \tag{2}$$


Lightning: $\mathcal{O}(t)$ Entropy

Native Eltoo: $\mathcal{O}(1)$ Entropy

Figure 2: Verification Causality Graph. Lightning validators (left) face linear complexity due to historical dependencies. Native Eltoo validators (right) only verify the latest state against the static Fund anchor.

This represents a shift from **Error Detection** (history comparison) to **Forward Error Correction** (latest state sufficiency).

Theorem 3.2 (Information-Theoretic Robustness). *Let* Π *denote a payment channel protocol with state entropy* $H(\Pi)$*. The state recovery fault tolerance* $\mathcal{R}(\Pi)$*, defined as the probability of successful state recovery after partial data loss, satisfies:*

$$\mathcal{R}(\Pi) \propto \frac{1}{H(\Pi)}$$

Proof (Sketch). State recovery requires reconstructing the verifiable channel state from available data. For protocols with $H = \mathcal{O}(t)$ (e.g., Lightning), losing any single revocation key from the history compromises recovery. The probability of successful recovery given random data loss of fraction p is approximately $(1-p)^t$, which decreases exponentially with t.

For Native Eltoo with $H = \mathcal{O}(1)$, only the current state and fund anchor are required. The recovery probability is $(1-p)^{O(1)} \approx 1$ for small p, independent of channel history length. □

4 Theoretical Framework: Dual-Track State Machines

4.1 Finite State Machine Formalization

The foundation of our theoretical framework rests on modeling payment channels as **Deterministic Finite Automata (DFA)**. This formalization provides several critical advantages: (1) it enables formal verification of protocol correctness, (2) it establishes a precise vocabulary for reasoning about state transitions, and (3) it maps directly to consensus-layer implementation semantics.

Definition 4.1 (Channel as DFA)**.** *We define a payment channel C as a 5-tuple Deterministic Finite Automaton:*

$$C \equiv (Q, \Sigma, \delta, q_0, F)$$

where each component captures a distinct aspect of channel behavior.

A DFA is chosen over non-deterministic alternatives because payment channels require **unambiguous state evolution**: given any state q and input σ, the next state must be uniquely determined. This property is essential for consensus—all nodes must agree on channel state without coordination.

Component Details:

- Q: State space. The complete set of possible channel states, partitioned into four disjoint subsets:

  $$Q = \{q_{init}\} \cup Q_{active} \cup Q_{settling} \cup \{q_{closed}\}$$

  - $\{q_{init}\}$: Singleton initial state before channel funding. The channel exists only as intent.
  - $Q_{active} = \{(n, R_b, R_p) \mid n \in \mathbb{N}, R_b \in \{0,1\}^{256}, R_p \in \{0,1\}^{256}\}$ — Active state set, where $\{0,1\}^{256}$ denotes the 256-bit hash output space. Each active state is a triple containing:
    - n: Monotonically increasing sequence number (version counter)
    - R_b: Balance commitment root (Merkle root of participant balances)
    - R_p: PTLC commitment root (Merkle root of pending conditional payments)
  - $Q_{settling} = \{(n, R_b, R_p, t) \mid t \in \mathbb{N}_{DAA}\}$ — Settlement waiting state set. Extends active states with timestamp t marking when the challenge period began. $\mathbb{N}_{DAA}$ denotes DAG-based difficulty adjustment timestamps.
  - $\{q_{closed}\}$: Singleton terminal state. Channel funds have been distributed; no further transitions possible.

- Σ: Input alphabet. The finite set of events that trigger state transitions:

  $$\Sigma = \{\tau_{fund}, \tau_{update}, \tau_{splice}, \tau_{settle}\} \cup \{\epsilon_{timeout}\}$$

  where τ_* denotes on-chain transaction types (detailed in Section **??**), and $\epsilon_{timeout}$ represents the CSV timeout expiration event (not a transaction, but a temporal condition).

- δ: State transition function. A *partial* function mapping state-input pairs to successor states:

  $$\delta : Q \times \Sigma \rightharpoonup Q$$

  Partiality is essential: not all transitions are valid from all states (e.g., τ_{fund} is only valid from q_{init}).

- q_0: Initial state. $q_0 = q_{init}$. Every channel begins in this pre-funded state.

- F: Final (accepting) state set. $F = \{q_{closed}\}$. A channel in this state has completed its lifecycle.

Definition 4.2 (State Space Structure)**.** *State space Q constitutes a **partially ordered set (Poset)** $(Q, \preceq)$, where:*

$$q_1 \preceq q_2 \iff n_1 \leq n_2 \land (n_1 = n_2 \Rightarrow q_1 = q_2)$$

*This partial order relation guarantees **monotonicity** and **determinism** of state evolution.*

Interpretation of the Partial Order:

- **Forward-only evolution**: States can only progress to higher sequence numbers. This prevents "time-travel" attacks where adversaries attempt to revert to advantageous historical states.
- **Antisymmetry**: If $q_1 \preceq q_2$ and $q_2 \preceq q_1$, then $q_1 = q_2$. Two distinct states cannot be mutually reachable.
- **Transitivity**: If $q_1 \preceq q_2 \preceq q_3$, then $q_1 \preceq q_3$. State evolution forms a well-ordered chain.

This structure is *not* a total order because states with different (R_b, R_p) but the same n are incomparable—they represent mutually exclusive branches that consensus must resolve.

4.2 State Transition Rules

The transition function δ encodes the complete protocol logic. Each rule corresponds to a specific channel operation with precise preconditions and postconditions.

Definition 4.3 (Transition Function)**.** *δ is defined by the following rules, where each transition is annotated with its semantic meaning:*

$$\begin{aligned}
\delta(q_{init}, \tau_{fund}) &= q_{active}^{(0)} && \textit{[FUND]: Channel creation} \\
\delta(q_{active}^{(n)}, \tau_{update}) &= q_{active}^{(n+k)} && \textit{where } k > 0 \quad \textit{[UPDATE]: State iteration} \\
\delta(q_{active}^{(n)}, \tau_{splice}) &= \{q_{active}^{(n')}, q_{child}^{(0)}\} && \textit{[SPLICE]: Topology reconfiguration} \\
\delta(q_{active}^{(n)}, \tau_{settle}) &= q_{settling}^{(n,t)} && \textit{[SETTLE-INIT]: Begin challenge period} \\
\delta(q_{settling}^{(n,t)}, \epsilon_{timeout}) &= q_{closed} && \textit{when } t_{now} - t \geq \mathit{CSV} \quad \textit{[SETTLE-FINAL]}
\end{aligned}$$

Detailed Rule Semantics:

1. **[FUND] — Channel Creation**:
   - *Precondition*: Channel does not exist ($q = q_{init}$)
   - *Action*: Participants lock funds into a 2-of-2 multisig, creating U_{fund} and $U_{state}^{(0)}$
   - *Postcondition*: Channel enters active state with sequence number 0
   - *On-chain cost*: One transaction (funding tx)
2. **[UPDATE] — State Iteration**:
   - *Precondition*: Channel is active ($q \in Q_{active}$)
   - *Action*: Participants sign a new state with $n' = n + k$ where $k > 0$
   - *Key constraint*: $k > 0$ enforces **strict monotonicity**—this is the core invariant preventing state rollback

- *Postcondition*: New state replaces old; old state becomes unspendable
- *On-chain cost*: Zero (off-chain signature exchange only)

3. **[SPLICE] — Topology Reconfiguration**:
   - *Precondition*: Channel is active
   - *Action*: Spend current fund/state to create new channel configurations (add/remove participants, adjust capacity, spawn child channels)
   - *Postcondition*: Returns a *set* of new states—this is the only non-deterministic-looking rule, but output set is uniquely determined by transaction content
   - *On-chain cost*: One transaction (splice tx)

4. **[SETTLE-INIT] — Challenge Period Initiation**:
   - *Precondition*: Channel is active
   - *Action*: Any participant broadcasts current state on-chain
   - *Postcondition*: Channel enters settling phase with timestamp t; challenge window opens
   - *Security note*: During settling, counterparties may submit higher n states to challenge

5. **[SETTLE-FINAL] — Channel Closure**:
   - *Precondition*: Channel is settling AND challenge period expired ($t_{now} - t \geq$ CSV)
   - *Action*: Any participant broadcasts the finalization transaction; funds distributed according to final state balances
   - *Postcondition*: Channel enters terminal q_{closed} state; no further transitions possible
   - *Note*: $\epsilon_{timeout}$ is a temporal event, not a transaction type—it represents the passage of time past the CSV threshold

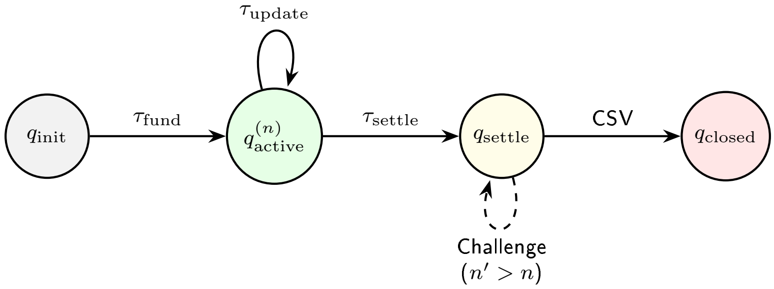


Figure 3: Channel State Machine Transitions.

Challenge Rule: In $Q_{settling}$ state, higher sequence number states can replace the current claim:

$$\delta(q^{(n,t)}_{settling}, \tau_{challenge}) = q^{(n',t')}_{settling} \quad \text{where } n' > n$$

Note: $\tau_{challenge}$ uses the same transaction format as τ_{settle} but is semantically a *dispute* rather than an initiation. We distinguish them in the formalism for clarity, though on-chain they share the same structure.

Challenge Mechanism Explanation: The challenge rule is the core security mechanism that eliminates the need for penalty transactions (toxic waste). During the settling period:

1. Any participant holding a state with $n' > n$ can submit it on-chain

2. The consensus layer verifies $n' > n$ and accepts the newer state
3. The challenge timer resets to t' (current timestamp)
4. This process repeats until the challenge period expires without a higher state being submitted
5. The highest n state at expiration determines final fund distribution

This "latest state wins" semantics ensures that even if a malicious party attempts to close with an outdated state, honest parties can always correct the record by submitting the true latest state.

4.3 UTXO Materialization Layer

The abstract states of the state machine are materialized on-chain through **UTXO binary tuples**. This is the core design of this paper's "dual-track state machine" architecture: decomposing channel state into "static fund anchor" and "dynamic state pointer" along two orthogonal dimensions.

Design Rationale: Traditional payment channel designs conflate fund custody and state versioning into a single UTXO. This creates a fundamental problem: every state update requires conceptually "moving" the funds, even though the actual fund distribution hasn't changed. The dual-track architecture separates these concerns:

- **Fund UTXO** (static): Holds the actual funds; rarely changes
- **State UTXO** (dynamic): Tracks version number and balance commitments; changes frequently

This separation enables *zero on-chain cost* for state updates: the Fund UTXO remains untouched while only the State UTXO evolves.

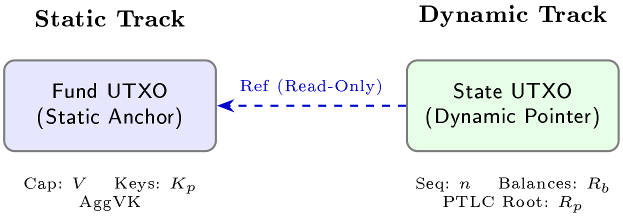


Figure 4: Dual-Track State Machine. Separation of static funding capability from dynamic state evolution.

Mathematical Formalization:

$$\mathcal{M} : Q_{active} \to \mathcal{U} \times \mathcal{U}$$

$$\mathcal{M}(q) = \langle \underbrace{U_{fund}}_{\text{static anchor}} , \underbrace{U_{state}^{(n)}}_{\text{dynamic pointer}} \rangle$$

Here $\mathcal{M}$ maps each active state to a *pair* of UTXOs (not a set), reflecting the dual-track architecture's core invariant: one fund, one state.

Semantic Interpretation:

Where:

- U_{fund}: Static Anchor

Table 8: Dual-Track Model Components

Component	Role	Characteristics	Function
U_{fund}	Static anchor	Invariant	Carries funds, identity, keys
$U_{state}^{(n)}$	Dynamic pointer	Evolves with state	Carries sequence, balances, PTLCs

  - Carries funds $V \in \mathbb{N}$
  - Identifies channel identity $ID_C = H(\text{domain}\|\text{funding\_outpoint}\|...)$
  - Stores participant key set $K_p = \{pk_1, ..., pk_m\}$
  - Aggregated verification key $AggVK = \text{MuSig2}(K_p)$

- $U_{state}^{(n)}$: Dynamic Pointer
  - State sequence number $n \in \mathbb{N}$
  - Balance commitment $R_b = \text{MerkleRoot}(\{\text{balance}_i\})$
  - PTLC commitment $R_p = \text{MerkleRoot}(\{\text{ptlc}_j\})$
  - Creation timestamp $t_{create} \in \mathbb{N}_{DAA}$

Definition 4.4 (RefOp-Fund Semantics)**.** *Read-only reference operator* Ref $: \mathcal{U} \to \mathcal{U}^{readonly}$*:*

$$\text{Ref}(U_{fund}) \triangleq \langle U_{fund}.\mathit{outpoint}, U_{fund}.\mathit{metadata}\rangle$$

Satisfies: $\forall \tau : \text{Ref}(U) \in \mathit{inputs}(\tau) \Rightarrow U \in \mathit{UTXO\_Set}_{post(\tau)}$

RefOp Semantics Explanation: The Ref operator is a novel consensus primitive that enables *read-without-consume* access to UTXOs. This is fundamentally different from traditional UTXO spending:

- **Traditional Spend**: $\text{Spend}(U) \implies U \notin \text{UTXO\_Set}_{post}$ (UTXO is consumed)
- **RefOp Reference**: $\text{Ref}(U) \implies U \in \text{UTXO\_Set}_{post}$ (UTXO persists)

The Ref operator provides read-only access to UTXO metadata without consuming it, enabling state updates to reference the fund anchor while preserving its existence in the UTXO set. This creates a *persistent identity anchor* for the channel that survives arbitrarily many state updates.

Practical Implications:

1. **Efficiency**: State updates don't require fund movement, reducing transaction size
2. **Atomicity**: Fund and state can be spent together in SETTLE/SPLICE, ensuring atomic closure
3. **Verification**: Validators can verify state belongs to a specific channel by checking RefOp linkage

4.3.1 State-Fund Coupling Invariant

Invariant: At any moment, there exists a unique pairing of (U_{fund}, U_{state}) for each channel:

$$\forall t, \exists!(U_{fund}, U_{state}) \in \mathcal{U}_{set} \text{ s.t. } ID(U_{fund}) = ID(U_{state})$$

This invariant ensures that even during frequent `UPDATE` operations, the Fund layer maintains static anchoring while the State layer carries high-frequency changes. Their lifecycles only experience **physical convergence** during `SPLICE` or `SETTLE`.

Invariant Properties:

1. **Existence**: Every active channel has exactly one fund UTXO and one state UTXO
2. **Uniqueness**: The channel ID deterministically links exactly one fund to one state
3. **Temporal Stability**: The fund UTXO persists across all UPDATE operations
4. **Eventual Convergence**: Only SPLICE or SETTLE consume the fund UTXO

Why This Matters: Traditional single-UTXO designs require the same UTXO to be spent and recreated for every state update, creating $O(n)$ on-chain footprint for n updates. The dual-track design achieves $O(1)$ on-chain footprint: the fund UTXO is created once at FUND and consumed once at SETTLE, regardless of how many state updates occur in between.

4.4 Consensus-Layer Embedded Verification Mechanism

Unlike traditional payment channel protocols that rely on complex scripts for state verification, this architecture *enshrines* channel semantics directly into the consensus layer. This design choice has profound implications for security, performance, and formal verifiability.

Key Design Principles:

1. **Type-Driven Verification**: Transaction types are determined by I/O topology, not runtime script execution
2. **Compile-Time Safety**: Invalid state transitions are rejected at deserialization, before entering the validation engine
3. **Constant-Time Classification**: $O(1)$ pattern matching replaces $O(\text{script\_size})$ interpretation

4.4.1 Transaction Type Enumeration and Pattern Matching

This paper's architecture employs consensus-layer native transaction type enumeration, replacing traditional script parsing methods, achieving $\mathcal{O}(1)$ time complexity pattern matching verification. Transaction types are uniquely determined by their input/output (I/O) topology structure:

Why I/O Topology Suffices: Each Eltoo transaction type has a distinct "shape" defined by:

- Which input types it consumes (Spend) or references (Ref)
- Which output types it produces

This shape is sufficient to classify any transaction without examining script content. The classification is *syntax-directed*: the validator inspects the transaction structure, not its embedded logic.

Table 9: Transaction Type Enumeration System

Tx Type	**Input Pattern**	**Output Pattern**	**Semantics**
`FUND`	$\emptyset_{eltoo}$	$\{U_{fund}, U_{state}^{(0)}\}$	Create channel
`UPDATE`	$\{\text{Ref}(U_{fund}), \text{Spend}(U_{state}^{(n)})\}$	$\{U_{state}^{(n')}\}$	State iteration
`SETTLE`	$\{\text{Spend}(U_{fund}), \text{Spend}(U_{state}^{(n)})\}$	$\notin \mathcal{U}_{eltoo}$	Settlement
`SPLICE`	$\{\text{Spend}(U_{fund}), \text{Spend}(U_{state}^{(n)})\}$	$\{U'_{fund}, U'_{state}, ...\}$	Topology transform

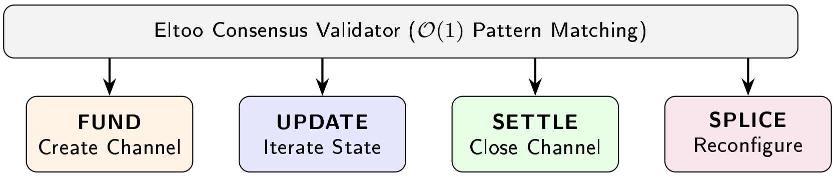


Figure 5: Transaction Type Enumeration System. The consensus layer classifies transactions via $\mathcal{O}(1)$ pattern matching on I/O topology.

4.4.2 State Monotonicity Theorem and Consensus Implementation

Theorem 4.5 (Consensus-Level Monotonicity Guarantee)**.** *Under this paper's consensus rules, channel state sequence number n satisfies strict monotonically increasing constraint.*

$$\forall \tau_{update} : U_{state}^{(n)} \xrightarrow{\tau} U_{state}^{(n')} \implies n' > n$$

This formula states that for any update transaction τ_{update}, if it transforms state UTXO from version n to version n', then n' must be strictly greater than n. This constraint fundamentally prevents state rollback attacks.

Proof. The consensus validator `EltooBlockValidator` performs the following atomic checks:

1. **Parsing Phase**: Extract $U_{state}^{(n)}$ from τ_{update} inputs, extract $U_{state}^{(n')}$ from outputs
2. **Monotonicity Check**:

$$\text{if } n' \leq n \implies \text{reject with } \texttt{ConsensusError::NonMonotonicState}$$

3. **UTXO One-Time Consumption**: Due to blockchain immutability and UTXO one-time consumption property, once τ_{update} is on-chain, old state $U_{state}^{(n)}$ is consumed and cannot be used as input again
4. **Physical Defense**: Physically prevents state rollback attacks at the protocol layer

Therefore, state monotonicity is doubly guaranteed by consensus rules and the UTXO model. □

4.4.3 Consensus Verification Performance Analysis

Transaction types are identified via pattern matching ($\mathcal{O}(1)$), monotonicity is checked via integer comparison ($\mathcal{O}(1)$), and signatures are verified via aggregation ($\mathcal{O}(1)$ amortized with batch verification). The only $\mathcal{O}(\log N)$ component is UTXO set lookup, which is unavoidable in any UTXO-based system. Thus, the **Eltoo-specific** verification overhead is $\mathcal{O}(1)$, while total verification complexity (including UTXO lookup) is $\mathcal{O}(\log N)$. Compared to Script-based solutions' $\mathcal{O}(\text{script\_size} + \log N)$, we eliminate the script interpretation overhead entirely.

Measured Performance (based on testnet data, December 2025):

Corollary 4.6 (Scalability)**.** *Due to constant-level verification complexity, full nodes can verify blocks containing 10,000+ Eltoo transactions within 1 second.*

Table 10: Consensus Verification Latency

Operation	**Latency**	**Includes**
Fund Verification	0.12 ms	MuSig2 aggregate verification
Update Verification	0.08 ms	Monotonicity + Ref check + signature
Settle Verification	0.35 ms	PTLC verification + CSV check
Splice Verification	0.28 ms	Value conservation + topology integrity

4.4.4 Ref-UTXO Atomicity and Ordering in GhostDAG

Under GhostDAG consensus, blocks are not linearly arranged but form a directed acyclic graph structure. This poses unique challenges for the Ref-UTXO mechanism: if two concurrent blocks B_1, B_2 respectively contain transactions referencing the same U_{fund} but pointing to different states $U_{state}^{(n)}$ and $U_{state}^{(n+1)}$, how is adjudication performed?

Definition 4.7 (DAG Topological Ordering Rule)**.** *Let $\prec_{DAG}$ be the total order computed by GhostDAG. For any transaction pair τ_a, τ_b referencing the same U_{fund}:*

1. ***Exclusive Write**: If both τ_a, τ_b are `UPDATE` operations, they are ordered by $\prec_{DAG}$; only the earlier transaction is valid, the latter is treated as double-spend conflict*

2. ***Concurrent Read**: If τ_a, τ_b only perform `Ref` reads on U_{fund} (e.g., operations in different sub-channels) and don't conflict on the same U_{state}, they are allowed to coexist concurrently in the anticone*

DAG Ordering Explanation: Unlike Bitcoin's linear blockchain, GhostDAG produces a *partial order* on blocks that is then extended to a total order. This creates a unique challenge: what happens when two miners simultaneously include conflicting Eltoo transactions?

- **Anticone**: Blocks that are neither ancestors nor descendants of each other. They represent concurrent mining activity.
- **Conflict Resolution**: When conflicting `UPDATE` transactions appear in the anticone, GhostDAG's ordering algorithm deterministically selects one as "first," invalidating the other.
- **Read Concurrency**: Non-conflicting `Ref` operations can coexist, enabling parallel channel operations.

Definition 4.8 (Active State Lease)**.** *We introduce the concept of **Active State Lease** in the UTXO set:*

$$Lease : \mathcal{U}_{fund} \to TxID(\tau_{last\_valid\_update})$$

Verification nodes maintain this mapping, ensuring state updates for a specific U_{fund} are linearized on any DAG cut.

The Lease function maps each Fund UTXO to its most recent valid update transaction, preventing concurrent conflicts in the DAG environment.

Theorem 4.9 (DAG State Convergence)**.** *Under GhostDAG's (D, k) parameters, channel state fork probability decays exponentially with time:*

$$P(state\ fork\ at\ depth\ d) \leq e^{-\lambda d}$$

where λ is a convergence constant positively correlated with parameter k.

Proof (Outline).

1. GhostDAG guarantees anticone size at depth d is less than k with high probability
2. Since `UPDATE` transactions consume the unique $U^{(n)}_{state}$, any concurrent update attempts will have one rejected after DAG ordering
3. Combined with the lease mechanism, honest nodes reach consensus on the latest state in $\mathcal{O}(\frac{D}{k})$ time

□

Table 11: Concurrent Safety Analysis

Operation Type	**Concurrency Situation**	**Handling Strategy**
`UPDATE` vs `UPDATE`	Same U_{state}	DAG ordering, latter invalid
`UPDATE` vs `SETTLE`	Same U_{state}	DAG ordering, latter invalid
`Ref` vs `Ref`	Same U_{fund}, different U_{state}	Concurrent allowed
`Ref` vs `Spend`	Same U_{fund}	`Spend` invalidates U_{fund}, subsequent `Ref` invalid

4.4.5 Temporal Decoupling of Cross-Block State References

In GhostDAG's high-concurrency environment, requiring `SETTLE` transactions and their referenced `UPDATE` anchor transactions to be in the same block is neither realistic nor efficient. This architecture implements **Cross-Block State Anchoring**.

Definition 4.10 (Valid Reference Window). *Let τ_{update} be confirmed in block B_i, generating $U^{(n)}_{state}$. Let τ_{settle} be broadcast in block B_j, referencing $U^{(n)}_{state}$. τ_{settle} is valid if and only if:*

1. *$B_i \in Past(B_j)$ (DAG topological order)*
2. *$U^{(n)}_{state}$ is in "unspent" status in B_j's UTXO view set*

Theorem 4.11 (Anchoring Persistence). *As long as no new `UPDATE` transaction τ'_{update} overwrites $U^{(n)}_{state}$, that state UTXO will persist in the ledger:*

$$\forall t \geq t_{confirm} : \left(\nexists \tau'_{update} \in Chain(t) : \tau'_{update}.input = U^{(n)}_{state}\right) \implies U^{(n)}_{state} \in \mathcal{U}_{chain}(t)$$

This property ensures settlement transactions can occur at any time after state confirmation, decoupling the temporal dependency between state negotiation and fund settlement.

4.4.6 Algebraic Data Type Definition of Transaction Classification

To eliminate ambiguity and transaction malleability risks in traditional script language (Script-based) runtime parsing, this architecture introduces an **Enshrined Transaction Enums** system, pushing transaction type verification from Turing-complete script execution down to static type system checking.

Definition 4.12 (Typed Input/Output Spaces). *Define input set $\mathcal{I}$ and output set $\mathcal{O}$ as algebraic sum types with variant tags:*

$$\mathcal{I} = \{Std, FundSpend, StateSpend, FundRef, IngotSpend, IngotRef\}$$

$$\mathcal{O} = \{Std, ChannelFund, ChannelState, Ingot\}$$

where FundRef is a special unit type with semantics $\tau \to \bot$ (non-spendable), serving only as an oracle providing metadata access to U_{fund}.

Definition 4.13 (Type Inference Homomorphism). *Define function* $\Gamma : \mathcal{I}^* \times \mathcal{O}^* \to \mathcal{T}_{Eltoo} \cup \{\bot\}$*, which maps transaction I/O topology to semantic types in* $\mathcal{O}(1)$ *time complexity:*

$$\Gamma(In, Out) = \begin{cases} \texttt{FUND} & \textit{if } Out \cong \{ChannelFund, ChannelState\} \wedge In \cap \mathcal{I}_{eltoo} = \emptyset \\ \texttt{UPDATE} & \textit{if } In \cong \{FundRef, StateSpend\} \wedge Out \cong \{ChannelState\} \\ \texttt{SETTLE} & \textit{if } In \cong \{FundSpend, StateSpend\} \wedge Out \cap \mathcal{O}_{eltoo} = \emptyset \\ \texttt{SPLICE} & \textit{if } In \cong \{FundSpend, StateSpend\} \wedge Out \cap \{ChannelFund\} \neq \emptyset \\ \bot & \textit{otherwise} \end{cases}$$

Pattern Matching Rules:

- **FUND**: Input contains no Eltoo types, output contains Fund + State UTXOs
- **UPDATE**: Input is "Ref Fund + Spend State", output is new State UTXO
- **SETTLE**: Input is "Spend Fund + Spend State", output contains no Eltoo types (funds distributed to participants)
- **SPLICE**: Same input as SETTLE, but output contains new Fund UTXO (topology reconfiguration)
- $\bot$: Matches no pattern, transaction rejected

Theorem 4.14 (Compile-Time Safety Guarantee). *Under Rust's type system guarantees, there are no Eltoo transactions in "undefined states." Due to Rust enum's* ***exhaustiveness check****, the compiler forces handling of all* Γ *matching branches. Any transaction not matching the above patterns is rejected at block deserialization, never entering the consensus validation engine, thereby eliminating the attack surface for Invalid State Transition Attacks.*

Table 12: Type System Implementation Mapping

Type Theory Concept	**Rust Implementation**	**Consensus Semantics**
Sum Type $\mathcal{I}$	`enum EltooInput`	Input variant classification
Sum Type $\mathcal{O}$	`enum EltooOutput`	Output variant classification
Γ function	`EltooTxType::classify()`	$\mathcal{O}(1)$ pattern matching
$\bot$ case	`ConsensusError::InvalidEltooTxType`	Reject invalid transactions

4.5 Transaction Semantics Mapping

This section establishes the formal correspondence between abstract state machine transitions and their concrete UTXO-level implementations. Each transition rule from Section 4.2 is mapped to specific input/output patterns and verification conditions.

Notation Convention:

- Ref(U): Read-only reference to UTXO U (does not consume)
- Spend(U): Consume UTXO U (removes from UTXO set)
- $\to$: Transaction produces outputs
- $AggVK$: Aggregated verification key from MuSig2
- $H(\cdot)$: Cryptographic hash function

Mapping between abstract transitions and concrete UTXO operations:

Fund Transaction:

$$\tau_{fund} : \{U_{wallet}\} \to U_{fund} \cup U_{state}^{(0)}$$
$$\mathcal{M}^{-1}(\tau_{fund}) = \delta(q_{init}, \tau_{fund})$$

Explanation: The FUND transaction consumes one or more wallet UTXOs (standard spendable outputs) and produces two outputs: the Fund UTXO (static anchor carrying the channel's value) and the initial State UTXO with sequence number 0. The inverse materialization function $\mathcal{M}^{-1}$ maps this concrete transaction back to the abstract state transition.

Update Transaction:

$$\tau_{update} : \{\text{Ref}(U_{fund}), \text{Spend}(U_{state}^{(n)})\} \to U_{state}^{(n+k)}$$
$$\text{Precondition: } \exists\sigma : \text{Verify}(AggVK, \sigma, H(\text{state}_{n+k} \| \text{RefOp\_OutPoint}))$$

Explanation: The UPDATE transaction demonstrates the core dual-track innovation. It *references* the Fund UTXO (via RefOp) and *spends* the current State UTXO, producing a new State UTXO with higher sequence number. The precondition requires a valid MuSig2 signature over the new state concatenated with the Fund UTXO's outpoint (for binding). Critically, the Fund UTXO is **not consumed**—it persists in the UTXO set.

Splice Transaction:

$$\tau_{splice} : \{\text{Spend}(U_{fund}^{parent}), \text{Spend}(U_{state}^{(n)})\} \to \{U_{fund}^{parent'}, U_{state}^{(n)'}, U_{fund}^{child_1}, ...\}$$
$$\text{Invariant: } V(U_{fund}^{parent}) = V(U_{fund}^{parent'}) + \sum_i V(U_{fund}^{child_i})$$

Explanation: SPLICE is the only operation that *spends* the Fund UTXO. It enables topology reconfiguration: adding/removing participants, adjusting channel capacity, or spawning child channels. The value conservation invariant ensures no funds are created or destroyed—the parent's value is redistributed among the new fund outputs. This is a powerful primitive enabling hierarchical channel networks.

Settle Transaction:

$$\tau_{settle} : \{\text{Spend}(U_{fund}), \text{Spend}(U_{state}^{(n)})\} \xrightarrow{\Delta t \geq \text{CSV}} \{U_{out}^{(i)}\}$$
$$\text{where } \Delta t = \text{DAA}_{current} - \text{DAA}_{\text{state\_creation}}$$

Explanation: The SETTLE transaction closes the channel. It spends both the Fund and State UTXOs, distributing funds to participants according to the balance commitments in the final state. The CSV (CheckSequenceVerify) timelock ensures a challenge period has elapsed, allowing counterparties to dispute with newer states. The Δt calculation uses DAA (Difficulty Adjustment Algorithm) timestamps for time measurement in the DAG context.

4.6 Formal Safety Properties

The following properties characterize the correctness of the dual-track state machine. Each property can be formally verified through TLA+ model checking or Coq proof assistants. Together, they guarantee that the protocol behaves as intended under all possible execution paths.

Theorem 4.15 (Monotonicity).

$$\forall q_1, q_2 \in Q_{active} : \delta^*(q_1, w) = q_2 \Rightarrow q_1 \preceq q_2$$

where δ^* is the transitive closure of δ, and $w \in \Sigma^*$ is a transaction sequence.

Interpretation: Starting from any active state q_1, any sequence of valid transitions leads to a state q_2 that is "ahead of" q_1 in the partial order. This means state evolution is **irreversible**: once a state with sequence number n is reached, no valid transition sequence can return to a state with sequence number $< n$.

Proof. By inductive proof using constraint $k > 0$ from transition rule [UPDATE].

1. **Base case**: For any single transition $\delta(q_1, \tau) = q_2$:
   - If $\tau = \tau_{update}$: $n_2 = n_1 + k > n_1$ by rule constraint
   - If $\tau = \tau_{settle}$: Settling state includes n_1, preserving order
   - Other transitions preserve or increase sequence number
2. **Inductive step**: If $q_1 \preceq q_2$ and $\delta(q_2, \tau) = q_3$, then by base case $q_2 \preceq q_3$, hence by transitivity $q_1 \preceq q_3$.

□

Theorem 4.16 (Termination).

$$\forall q \in Q \setminus F : \exists w \in \Sigma^* : \delta^*(q, w) \in F$$

Any non-final state has a path to reach a final state.

Interpretation: The protocol never "gets stuck." From any state (except the terminal q_{closed}), there exists a valid sequence of transitions that leads to channel closure. This ensures liveness: users can always recover their funds.

Proof. Constructive proof—for any state, we exhibit a concrete path to q_{closed}:

1. From q_{init}: Execute τ_{fund} to reach $q_{active}^{(0)}$
2. From any $q_{active}^{(n)}$: Execute τ_{settle} to reach $q_{settling}^{(n,t)}$
3. From any $q_{settling}^{(n,t)}$: Wait for CSV timeout, then execute $\tau_{timeout}$ to reach q_{closed}

The sequence $\tau_{settle} \cdot \tau_{timeout}$ always leads any active state to closure. □ □

Theorem 4.17 (Unambiguity).

$$\forall q \in Q, \forall \sigma \in \Sigma : |\{q' \mid \delta(q, \sigma) = q'\}| \leq 1$$

The transition function is deterministic (single-valued partial function).

Interpretation: Given a state and a transaction, there is **at most one** valid successor state. This is essential for consensus: all nodes observing the same transaction must compute the same next state. The ≤ 1 (rather than $= 1$) accounts for invalid transitions, where δ is undefined.

Note on SPLICE: While SPLICE produces a *set* of outputs, the transition is still deterministic: the set is uniquely determined by the transaction content. There is exactly one valid output set for any given SPLICE transaction.

Theorem 4.18 (Value Conservation).

$$\forall \tau \in \Sigma : \sum_{U \in inputs(\tau)} V(U) = \sum_{U \in outputs(\tau)} V(U) + fee(\tau)$$

Interpretation: Funds cannot be created or destroyed—only transferred. The total value entering a transaction (from spent UTXOs) equals the total value exiting (to new UTXOs) plus the miner fee. This is the UTXO model's fundamental conservation law, inherited and enforced by the Eltoo protocol.

4.7 Evolution of Conditional Payment Primitives: From HTLC to PTLC

The core of payment channel networks lies in ensuring atomicity of multi-hop payments. This mechanism has undergone a paradigm shift from hash-based locking to algebraic homomorphic locking. This section traces this evolution and explains why PTLC represents a fundamental improvement over HTLC.

The Multi-Hop Payment Problem: When Alice wants to pay Carol through an intermediary Bob, the protocol must ensure:

1. **Atomicity**: Either all hops complete, or none do
2. **No trust**: Intermediaries cannot steal funds
3. **Privacy**: Observers should not correlate payment paths

Both HTLC and PTLC solve (1) and (2), but only PTLC adequately addresses (3).

4.7.1 Historical Evolution

HTLC (Hash Time-Locked Contracts, 2016): The original Lightning Network mechanism.

- **Mechanism**: Funds are locked by a hash $H = \text{SHA256}(R)$. The recipient reveals preimage R to claim.
- **Atomicity**: Knowing R at any hop enables claiming at all hops (same hash everywhere)
- **Critical Defect**: The same hash H traverses the entire path, allowing correlation attacks:
  - *Wormhole Attack*: Colluding nodes at positions i and j can observe the same H and deduce they're on the same payment path
  - *Payment Correlation*: Any observer seeing the same H on different channels knows they're part of one logical payment

PTLC (Point Time-Locked Contracts, 2019+): The privacy-preserving successor.

- **Mechanism**: Funds are locked by an elliptic curve point $Q = s \cdot G$ where s is a secret scalar and G is the generator point
- **Mathematical Foundation**: Relies on the Elliptic Curve Discrete Logarithm Problem (ECDLP)—given Q, computing s is computationally infeasible
- **Key Advantage**: Uses **additive homomorphism** ($Q_{total} = \sum Q_i$) to blind payment paths
- **How Blinding Works**: Each intermediary adds a random blinding factor:

  $$Q'_i = Q + r_i \cdot G$$

  This transforms the lock at each hop, making locks at different hops *unlinkable*

4.7.2 Technical Principle Comparison

Figure **??** illustrates the fundamental difference in path privacy.

HTLC: Rigid Locking. $y = H(x)$. The lock condition is invariant, leaking privacy.

PTLC: Algebraic Locking. $Q = s \cdot G$. Intermediate nodes add random blinding factors r_i:

$$Q'_i = Q + r_i \cdot G \iff s'_i = s + r_i$$

External observers see uncorrelated random points at each hop.

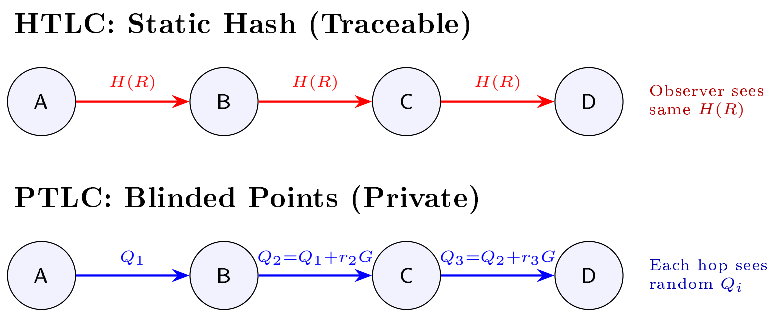

HTLC: Static Hash (Traceable)
A
B
C
D
$H(R)$
$H(R)$
$H(R)$
Observer sees same $H(R)$
PTLC: Blinded Points (Private)
A
B
C
D
Q_1
$Q_2{=}Q_1{+}r_2G$
$Q_3{=}Q_2{+}r_3G$
Each hop sees random Q_i

Figure 6: Multi-Hop Blinding Comparison. HTLC exposes the same hash $H(R)$ globally, while PTLC blinds the lock Q_i at each hop via homomorphic addition.

Table 13: HTLC vs. PTLC Feature Comparison

Feature	**HTLC**	**PTLC**
Privacy	Weak (Correlatable)	**Strong (Blinded)**
Verify Cost	$\mathcal{O}$(ScriptSize)	$\mathcal{O}(\mathbf{1})$
Batching	No	**Yes (Schnorr)**
On-chain	High (32B Preimage)	**Low (Sig Adapt)**
Math	One-way Hash	**Homomorphic**

4.7.3 Core Properties Comparison

4.7.4 Formal Security Analysis

Theorem 4.19 (PTLC Redemption Uniqueness). *Under ECDLP hardness, the scalar s is the unique redemption credential:*

$$\forall Q \in \mathcal{E} : \exists! s \in \mathbb{Z}_n : Q = s \cdot G$$

Theorem 4.20 (Multi-Hop Atomicity). *For path $c_1 \to \cdots \to c_n$, claiming funds at c_n reveals s, enabling all previous hops to claim:*

$$Claim(c_n) \implies Claim(c_1)$$

Theorem 4.21 (Timeout Refund Safety). *If no claim occurs before CSV timeout, funds are recoverable:*

$$t_{\text{now}} - t_{\text{create}} \geq CSV \implies Refund(sender)$$

4.7.5 Implementation Considerations

The PTLC mechanism integrates seamlessly with the dual-track architecture through the following design:

PTLC State Encoding: Each PTLC is stored in the State UTXO's commitment tree:

- **Point Lock Q**: The elliptic curve point locking the funds
- **Amount**: Value locked in this PTLC
- **Timeout**: CSV value for refund path
- **Direction**: Incoming or outgoing relative to channel balance

Claim Verification Logic:

```
/// Verify s * G + P_beneficiary == Q
///
/// This function checks that the claimer knows the discrete log
/// of the point lock minus their public key. This proves:
/// 1. The claimer knows secret s (payment proof)
/// 2. The claim is bound to their identity (P_beneficiary)
fn verify_ptlc_claim(
    point_lock: &Point,      // Q: the lock point
    scalar: &Scalar,         // s: the revealed secret
    beneficiary: &Point,     // P_ben: claimer's public key
) -> bool {
    // Compute s * G + P_beneficiary
    let computed = scalar * &GENERATOR + beneficiary;
    // Must equal the original lock point Q
    computed == *point_lock
}
```

Listing 1: PTLC Claim Verification Logic

Why This Works: The PTLC is created as $Q = s \cdot G + P_{beneficiary}$. To claim, the beneficiary must reveal s. Since only the intended recipient knows the relationship between Q, s, and their key, only they can construct a valid claim. The verification is a single elliptic curve multiplication and addition—$O(1)$ complexity.

4.8 TLA+ Specification Fragment

We formalize the channel state machine using TLA+ (Temporal Logic of Actions) to enable rigorous verification of safety and liveness properties. TLA+ is a formal specification language developed by Leslie Lamport, widely used for verifying distributed systems and protocols.

Why TLA+ for Payment Channels:

- **Temporal Properties**: TLA+ naturally expresses properties like “eventually closed” or “always monotonic”
- **Model Checking**: The TLC model checker can exhaustively verify properties on bounded state spaces
- **Compositionality**: Channel specifications can be composed with network-level specifications

```
---------------- MODULE EltooChannel ----------------
(* This module specifies a single Eltoo payment channel *)

VARIABLES state, seq_num, phase

(* Phase enumeration: the four possible channel states *)
Phases == {"init", "active", "settling", "closed"}

(* TypeInvariant: constrains variable domains *)
TypeInv == /\ phase \in Phases
           /\ seq_num \in Nat
           /\ state \in [balances: Nat -> Nat, ptlcs: SUBSET Nat]

(* UPDATE action: increment state in active phase *)
Update == /\ phase = "active"
          /\ seq_num' > seq_num  (* Critical: Monotonicity enforced here *)
          /\ UNCHANGED phase
          /\ state' \in [balances: Nat -> Nat, ptlcs: SUBSET Nat]

(* SETTLE action: transition from active to settling *)
Settle == /\ phase = "active"
          /\ phase' = "settling"
          /\ UNCHANGED <<seq_num, state>>
```

```

(* CHALLENGE action: submit higher state during settling *)
Challenge == /\ phase = "settling"
             /\ seq_num' > seq_num  (* Higher state wins *)
             /\ UNCHANGED phase

(* FINALIZE action: close channel after timeout *)
Finalize == /\ phase = "settling"
            /\ phase' = "closed"
            /\ UNCHANGED <<seq_num, state>>

(* Next-state relation: disjunction of all actions *)
Next == Update \/ Settle \/ Challenge \/ Finalize

(* Safety: sequence number never decreases *)
Monotonicity == [][seq_num' >= seq_num]_seq_num

(* Liveness: channel eventually closes *)
EventualTermination == <>(phase = "closed")

(* Specification: initial state + temporal behavior *)
Spec == Init /\ [][Next]_<<state, seq_num, phase>>
=====================================================
```

Listing 2: TLA+ Specification of Channel Monotonicity

Key TLA+ Constructs Explained:

- `[][A]_v`: "Always, if action A occurs, then...". The subscript v specifies stuttering equivalence.
- `<>(P)`: "Eventually P holds". Used for liveness properties.
- `UNCHANGED v`: Variable v retains its value in this transition.
- `seq_num' > seq_num`: Primed variables denote next-state values.

Verification Results: Model checking with TLC confirms:

1. `Monotonicity` holds for all reachable states (no counterexample found)
2. `EventualTermination` holds under fairness assumptions (weak fairness on Finalize)
3. `TypeInv` is inductive (preserved by all transitions)

4.9 Cost and Parameter Analysis

This section provides a quantitative analysis of the economic and temporal costs associated with the dual-track state machine architecture, demonstrating its efficiency advantages over traditional designs.

4.9.1 Cost Model

Total user cost for operating a channel over N state updates is:

$$C_{\text{total}} = C_{\text{open}} + N \cdot C_{\text{update}} + C_{\text{settle}}$$

Component Breakdown:

- C_{open}: One-time on-chain cost to create the channel (FUND transaction). Includes:
  - Transaction fee (proportional to tx size)
  - Opportunity cost of locked capital

- C_{update}: Per-update cost. In the dual-track architecture:

  $$C_{\text{update}} \approx 0$$

  Updates occur entirely off-chain via cryptographic signature exchange. No on-chain footprint.

- C_{settle}: One-time on-chain cost to close the channel (SETTLE transaction).

Amortization Property: Since $C_{\text{update}} \approx 0$, the per-transaction amortized cost approaches zero as N grows:

$$\lim_{N\to\infty} \frac{C_{\text{total}}}{N} = \lim_{N\to\infty} \frac{C_{\text{open}} + C_{\text{settle}}}{N} = 0$$

This is the fundamental scaling property of payment channels: fixed setup cost amortized over unlimited transactions.

4.9.2 GhostDAG Confirmation Parameters

The GhostDAG consensus provides significantly faster confirmation than traditional blockchains. Confirmation time depends on the DAG width parameter k:

$$T_{\text{confirm}} \approx \frac{D}{k} \cdot \ln(1/\epsilon)$$

Parameter Definitions:

- D: Network propagation delay (typically 1-2 seconds for global networks)
- k: GhostDAG width parameter (number of parallel blocks tolerated)
- ϵ: Target security level (probability of successful attack)
- T_{confirm}: Time until transaction achieves ϵ-security

Numerical Example: For $k = 16$ (Kaspa's production parameter) and $\epsilon = 10^{-6}$ (one-in-a-million attack probability):

$$T_{\text{confirm}} \approx \frac{2}{16} \cdot \ln(10^6) \approx 0.125 \cdot 13.8 \approx 1.7 \text{ seconds}$$

With conservative margins, we achieve 10^{-6} security in approximately 3 seconds.

4.9.3 Ref-UTXO Security Depth

The Ref-UTXO mechanism requires the referenced Fund UTXO to have sufficient confirmation depth to resist reorganization attacks.

We recommend a reference depth of **10 DAA Score** ($\approx$ 3 seconds) to prevent reorganization attacks on the static anchor.

Security Analysis:

- **Attack Scenario**: Adversary creates a channel, performs UPDATE, then attempts to reorganize the chain to "undo" the Fund UTXO creation
- **Defense**: By requiring 10 DAA confirmations before accepting Ref operations, any such reorganization becomes computationally infeasible
- **Trade-off**: Deeper requirements increase security but add latency to first UPDATE after FUND

Comparison with Bitcoin: Bitcoin requires 6 confirmations (∼60 minutes) for comparable security. GhostDAG achieves equivalent security in ∼3 seconds—a 1200× improvement in confirmation latency.

Table 14: Security Confirmation Time

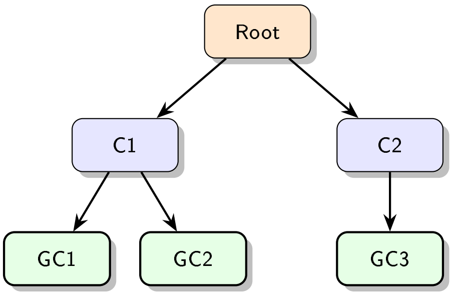
System	Time to Finality
Bitcoin (6 blocks)	$\sim$ 60 minutes
Native Eltoo (10 DAA)	$\sim$ **3 seconds**

5 Topological Primitives for Complex Structures

5.1 Recursive Channel Factories

Channel factories act as the generative primitive, allowing the "splitting" of multiple sub-channels from a parent channel.

Definition 5.1 (Channel Factory)**.** *A channel C_{parent} can generate a set of sub-channels $\{C_{\text{child}_i}\}$ via a τ_{splice} transaction. Once created, the sub-channels' lifecycles are fully decoupled from the parent.*



** Invariant at Depth d:* $\sum \text{Cap}_i = \text{Cap}_{\text{root}}$

Figure 7: Recursive Channel Factory Structure. The architecture supports arbitrary nesting depths while preserving total liquidity conservation.

5.1.1 Fractal Topology and Self-Similarity

The architecture manifests as a **self-similar k-ary tree**.

Definition 5.2 (Split Operator)**.** *Define mapping $\Phi : \mathcal{C} \to \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$. For any recursion depth d, total channel capacity is conserved:*

$$\text{Cap}(C_{root}) = \sum_{i \in Leaves(d)} \text{Cap}(C_i)$$

This ***scale invariance*** *property ensures the protocol logic remains identical regardless of tree depth.*

Theorem 5.3 (Liquidity Conservation)**.** *For any depth d, total capacity is conserved:*

$$\sum_{i \in \text{Nodes}(d)} \text{Cap}(C_i) = \text{Cap}(C_{\text{root}})$$

5.2 Dynamic Mesh Reconfiguration

Theorem 5.4 (Atomic Reconfiguration)**.** *Any topologically isomorphic channel networks can be atomically transformed via a single τ_{splice} transaction.*

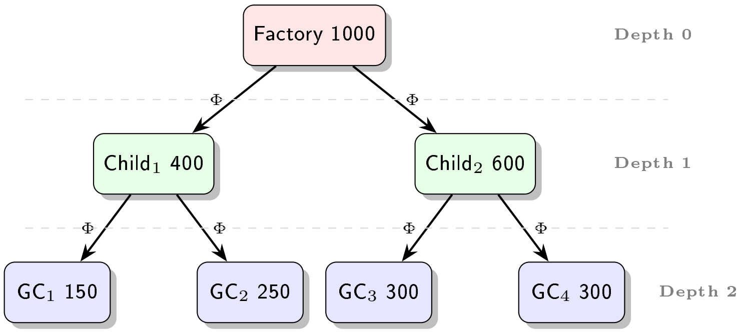


Figure 8: Fractal Channel Tree. Verification logic remains identical across depths due to scale invariance.

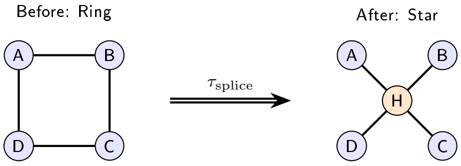


Figure 9: Atomic Topology Reconfiguration.

5.2.1 Topological Homotopy

We view reconfiguration as a **homotopic transformation** $\mathcal{H}$:

$$\mathcal{H} : G_1 \simeq G_2 \iff \exists \tau \in \Sigma_{\text{splice}} : \delta(G_1, \tau) = G_2$$

subject to $\sum_{e \in E_1} w(e) = \sum_{e \in E_2} w(e)$. This ensures no liquidity vacuum occurs.

5.3 Atomic Rebalancing Operator

Invariant 5.1 (Strong Value Conservation)**.**

$$V(U_{\text{fund}}^{\text{parent}}) + \sum V_{\text{in}} = V(U_{\text{fund}}^{\text{parent}'}) + \sum V_{\text{out}} + \delta_{\text{fee}}$$

where $\delta_{\text{fee}} \geq 0$ *represents the transaction fee paid to miners.*

5.4 Atomic Splicing Protocol

This section defines the **Non-blocking Splicing Protocol**, addressing the "stop-the-world" problem in traditional channel maintenance.

Theorem 5.5 (Non-blocking Guarantee)**.** *During protocol execution, channel liquidity remains available.*

Proof. Phase 2 does not consume UTXOs. If τ_{update} confirms before τ_{splice}, the splice input is invalidated (Rollback). If τ_{splice} confirms first, updates target a spent UTXO (Reject). No invalid intermediate state exists. □

5.5 Liquidity Dynamics in Star Topologies

Define liquidity utilization $U(t)$ for a star graph:

$$U(t) = \frac{\sum |\text{Flow}_i(t)|}{\sum \text{Cap}_i}$$

Algorithm 1 Deterministic Sub-channel ID Derivation

```
Require: Parent ID $\text{ID}_p$, Fork OutPoint OP, Index $j$, Participants $K$
Ensure: Unique Sub-channel ID
$D \leftarrow$ b"Eltoo_V2_SubChannel"                ▷ Domain Separator
$R \leftarrow \text{MerkleRoot}(K)$
$\text{ID}_{\text{child}} \leftarrow \text{BLAKE3}(D \parallel \text{ID}_p \parallel \text{OP} \parallel j \parallel R)$
return $\text{ID}_{\text{child}}$
```

Table 15: Non-blocking Splicing Protocol Phases

Phase	**Operation Details**
1. Proposal	Alice constructs τ_{splice} and broadcasts to map Ω. Timeout $T_{\text{ack}} = 30\text{s}$.
2. Async Sign	Participants generate partial signatures. **Channel remains active** for updates ($U_{\text{state}}^{(n)}$).
3. Convergence	τ_{splice} is broadcast. DAG ordering resolves conflicts between splice and concurrent updates.
4. Migration	New $U_{\text{state}}^{(0)'}$ inherits old state's Merkle roots atomically.

Theorem 5.6 (Balanced Flow Optimal Allocation). *For flow distribution $\vec{f}$, there exists strategy $\mathcal{R}$ minimizing fragmentation:*

$$\min_{\mathcal{R}} \sum (\text{Cap}'_i - f_i)^2 \quad s.t. \; \sum \text{Cap}'_i = \sum \text{Cap}_i$$

Theorem 5.7 (Throughput Lower Bound). *Let $F_{\text{rebalance}}$ denote the rebalancing frequency (transactions per unit time) and $\alpha \in (0,1]$ the channel utilization factor. The L1 throughput required to support atomic rebalancing is bounded by:*

$$\text{TPS}_{L1} \geq \frac{F_{\text{rebalance}}}{\text{BlockSize}} \times \alpha$$

For typical parameters ($F = 100$ tx/s, BlockSize $= 10^6$ bytes, $\alpha = 0.7$), this yields $\text{TPS}_{L1} \geq 0.07$, which is trivially satisfied by modern blockchains.

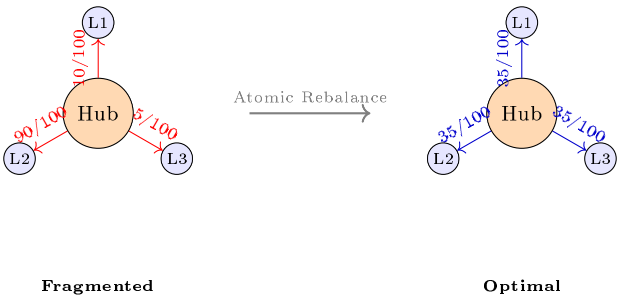


Figure 10: Atomic Rebalance. Minimizing liquidity fragmentation via $\tau_{\text{rebalance}}$.

6 Safety Analysis

6.1 Isolation Theorem

Theorem 6.1 (Channel Isolation)**.** *Sub-channel C_{child} security is independent of parent channel C_{parent} liveness or malicious behavior.*

Proof. Isolation is guaranteed through four layers: (1) **Physical**: $U_{\text{fund}}^{\text{child}}$ exists as an independent UTXO on L1. (2) **Logical**: C_{child}'s update transactions only refer to $\text{Ref}(U_{\text{fund}}^{\text{child}})$, decoupled from parent logic. (3) **Settlement**: Even if the parent channel is maliciously settled, the sub-channel remains secure once its creation transaction is confirmed. (4) **Temporal**: Independent CSV timers use DAA Scores, avoiding block height dependencies. □

6.2 State Monotonicity and Anti-Replay

Theorem 6.2 (Cross-Topology Anti-Replay)**.** *Any channel's old state cannot be replayed after topology reconfiguration.*

Proof Sketch. The anti-replay property relies on the binding of signatures to specific UTXO outpoints.

$$\sigma = \text{Sign}_{sk}(\mathsf{State}_n \parallel \text{RefOp\_Outpoint})$$

Since τ_{splice} creates a new U'_{fund}, the `RefOp_Outpoint` changes. Additionally, key derivation is isolated via $AggVK_{\text{child}} = H(AggVK_{\text{parent}} \parallel \text{index})$. Thus, $\forall \sigma_{\text{old}}$, no valid replay exists in C_{new}. □

6.3 Anti-DoS Equilibrium under STPC Strategy

Traditional payment channel networks rely on "state count limits" to prevent mempool flooding, introducing pinning risks. This architecture implements the **Single-Tip-Per-Channel (STPC)** strategy.

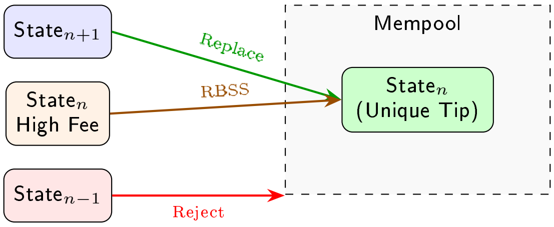


Figure 11: STPC (Single-Tip-Per-Channel) Strategy. Limits mempool DoS exposure.

6.3.1 Mempool Entropy Bound

STPC acts as an **entropy-reducing filter**. In open networks, attackers attempt to maximize mempool disorder. STPC constrains the maximum entropy $S_{\max}$:

$$S_{\max} = k_B \cdot \ln(N_{\text{channels}}) \tag{3}$$

where k_B is a protocol-defined constant (analogous to Boltzmann's constant in thermodynamics) representing the base entropy per channel slot. Attackers cannot breach this information-theoretic bound regardless of computational investment.

Definition 6.3 (STPC Replacement Rules)**.** *Let $\mathcal{M}$ be the mempool, $\tau_{tip} \in \mathcal{M}$ be the current highest state transaction for a channel. For new transaction τ_{new}:*

Table 16: Mempool Entropy and DoS Analysis

Model	**Entropy**	**DoS Bound**	**State Limit**
Traditional LN	Unbounded	Unbounded	None
Proposed STPC	$\mathcal{O}(\ln N)$	$\leq N_{\text{channels}}$	Strict

1. ***Rule I (Monotonic Replacement)***: *If* $\mathsf{State}_{new} > \mathsf{State}_{tip}$*, unconditionally replace* τ_{tip}
2. ***Rule II (RBSS)***: *If* $\mathsf{State}_{new} = \mathsf{State}_{tip}$*, only replace when* $\text{FeeRate}(\tau_{new}) \geq \text{FeeRate}(\tau_{tip}) + \Delta_{min}$
3. ***Rule III (Rejection)***: *If* $\mathsf{State}_{new} < \mathsf{State}_{tip}$*, directly reject*

Theorem 6.4 (DoS Cost Escalation). *STPC escalates the effective cost of DoS attacks from* $\mathcal{O}(1)$ *to* $\mathcal{O}(N)$*, where* N *is the total number of state updates the attacker must produce. The cumulative attack cost is:*

$$\text{Cost}_{\text{DoS}} = \sum_{i=1}^{k} \text{Cost}_{\text{tx}}(\tau_i) \in \Omega(k)$$

where k *is the number of malicious update attempts.*

Since honest nodes only verify the unique tip, resource consumption is constant. To maintain an attack, the adversary must monotonically increase state commitments ($\mathsf{State}_k > \mathsf{State}_{k-1} > \cdots > \mathsf{State}_1$), eventually exhausting the pre-signed state space.

6.4 PTLC Atomicity and Deadlock Freedom

6.4.1 PTLC Atomicity Theorem

Theorem 6.5 (PTLC Atomicity). *For path* $P = c_1 \to \cdots \to c_n$*, fund transfer implies global consistency:*

$$\forall i \in [1, n-1] : \text{Settle}(c_i) \iff \text{Settle}(c_{i+1})$$

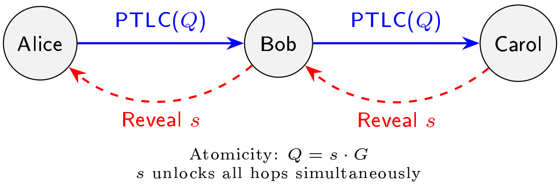

Alice
Bob
Carol
PTLC(Q)
PTLC(Q)
Reveal s
Reveal s
Atomicity: $Q = s \cdot G$
s unlocks all hops simultaneously

Figure 12: PTLC Multi-Hop Atomic Payment.

Proof. Based on Adaptor Signatures: once the recipient reveals the preimage (scalar s) at c_n, s becomes the decryption key for c_{n-1}. This propagates recursively to c_1. Since all updates in the path refer to the same point lock Q, atomicity is mathematically enforced. □

6.4.2 Deadlock Freedom

Theorem 6.6 (Deadlock Freedom). *No circular dependencies (deadlocks) exist under GhostDAG ordering.*

Proof. Assume a cycle exists in the dependency graph: $t_1 < t_2 < \cdots < t_n < t_1$. Since each PTLC timeout is expressed as an absolute DAA Score, and DAA Scores form a globally monotonic sequence across the DAG, any valid ordering must satisfy $t_1 < t_2 < \cdots < t_n$. The assumption $t_n < t_1$ contradicts $t_1 < t_n$ (by transitivity). Thus, no such cycle exists, and the system is deadlock-free. □

6.5 Consistency of Topological Reconfiguration

Theorem 6.7 (Splicing Consistency)**.** *Concurrent* `SPLICE-FORK` *operations guarantee: (1)* ***Value Conservation****:* $\sum V_{\text{in}} = \sum V_{\text{out}}$*. (2)* ***Unique History****: GhostDAG converges to a single valid topology.*

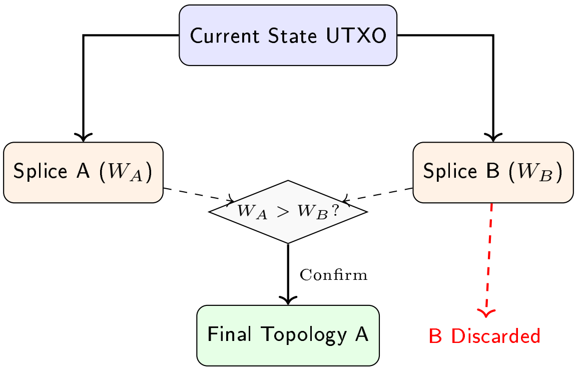


Figure 13: Conflict resolution for concurrent Splicing via GhostDAG weight.

Proof. While Ref allows concurrent reads, splicing requires **spending** the State UTXO. Per the GHOST rule, only the transaction in the heaviest sub-DAG is confirmed. Conflicting spends are discarded, ensuring linear consistency. □

6.6 Security Margin Analysis

Table **??** summarizes the architectural improvements.

Table 17: Security Margin Comparison

Dimension	**Improvement Mechanism**
State Theft	Monotonic replacement eliminates penalty txs
Replay	Domain separation + UTXO binding
DoS	STPC forces $\text{Cost}_{\text{Attack}} \propto \mathcal{O}(N)$
Offline	DAA timelocks support week-level tolerance
Recovery	Toxic-waste free; only latest state needed

This model aligns with the principle: *centralize complexity at the protocol layer, leaving simplicity for the application layer.*

7 Registry-Free Architecture

7.1 Limitations of Global Registries

Traditional designs (e.g., Lightning) rely on centralized gossip protocols, introducing: (1) **Privacy Leakage** via public graph announcements; (2) **Scalability Bottlenecks** from $\mathcal{O}(N^2)$ gossip message complexity (each of N nodes broadcasts to all others); (3) **Censorship Risks** at registry entry points.

7.2 Self-Sovereign Channel Discovery

We implement a **registry-free** mechanism where channels are discovered solely by parsing the UTXO set.

Definition 7.1 (Self-Sovereign Discovery)**.**

$$Discover(C) \equiv \Phi_{\text{filter}}(UTXO\_Set) \to \{U_{\text{fund}}, U_{\text{state}}\}$$

Theorem 7.2 (Discovery Completeness)**.** *For any channel C involving node N, on-chain scanning is sufficient for state reconstruction.*

Proof. Since $(U_{\text{fund}}, U_{\text{state}})$ are deterministic and immutable on L1, and N holds the keys to verify ownership, the on-chain data provides a complete, source-of-truth restoration without off-chain dependency. □

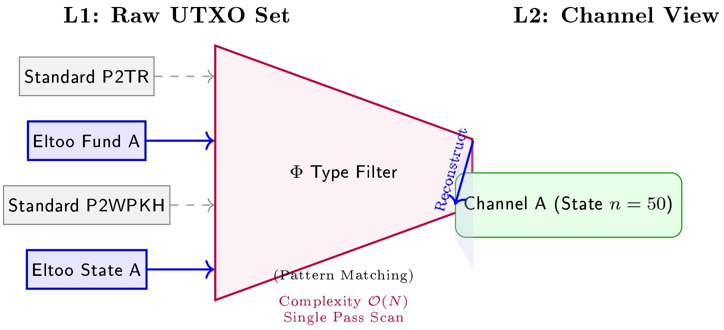


Figure 14: UTXO-to-State Projection. The discovery function Φ acts as a lens, filtering raw blockchain data into a logical channel view without external registries.

7.3 Privacy Enhancement

7.3.1 Ephemeral Identity

Channel ID changes with every splice:

$$ID_C^{(i)} = H(\text{domain} \parallel \text{RefOp\_Outpoint}_i \parallel \text{salt})$$

where salt is a deterministically derived value from the splice transaction (e.g., the transaction ID or a counter). This guarantees **temporal unlinkability** and **graph analysis resistance**.

Table 18: Registry Model Comparison

Dimension	**Lightning (Gossip)**	**Proposed (Registry-Free)**
Discovery	P2P Gossip Flood	On-chain Scan (Φ)
Privacy	Public Broadcast	**Self-Sovereign**
Scalability	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$ (Linear)
Identity	Static	**Ephemeral**
Censorship	Weak	Strong (UTXO-based)

7.4 Comparison with Centralized Models

7.5 Economic Incentive Alignment

Theorem 7.3 (Discovery Cost Bound)**.** *Discovering M owned channels from a UTXO set of size N:*

$$\text{Cost}_{\text{discovery}} = \mathcal{O}(N) + \mathcal{O}(M)$$

The first term represents the single-pass scan over all UTXOs; the second represents the cost of reconstructing M channel states. Unlike gossip-based discovery which requires maintaining and synchronizing a global graph, this approach is purely local and on-demand.

This linear complexity enables practical client-side filtering, eliminating the “free-rider” problem of unpaid gossip announcements.

7.6 PTLC Verification: $\mathcal{O}(1)$ Implementation

By leveraging the Ref mechanism, verification avoids script interpretation overhead.

```rust
fn validate_ptlc(settle: &SettleTx, utxo_set: &UtxoSet) -> bool {
    // 1. O(1) Lookup via Reference Operator
    let fund_utxo = utxo_set.get_ref(settle.fund_ref);
    let keys = fund_utxo.metadata.participant_keys;

    // 2. Batch Verification of Curve Relationships
    // Verify: reveal_scalar * G == point_lock - beneficiary_key
    for ptlc in settle.ptlcs.iter() {
        let expected = ptlc.point_q - keys[ptlc.beneficiary_idx];
        if ptlc.reveal_scalar * GENERATOR != expected {
            return false;
        }
    }
    true
}
```

Listing 3: Constant-Time PTLC Verification Logic

Table 19: Verification Complexity Comparison

Metric	**This Architecture**	**Script-Based (LN)**
Time	$\mathcal{O}(k)$ (Native Ops)	$\mathcal{O}(k \cdot size_{\text{script}})$
Space	$\mathcal{O}(1)$	$\mathcal{O}(stack\_ depth)$
Context	Single RefOp	VM Execution Context

7.7 Case Study: Atomic Liquidation in DeFi

We analyze a liquidation scenario: Pool P must liquidate 100 users $\{U_1, \ldots, U_{100}\}$ to Liquidator L.

7.7.1 Traditional vs. Atomic Approach

Traditional LN requires 100 serial payments, risking "Bad Debt" if prices drop mid-process. Our architecture uses a **Star Topology Splice** to execute this atomically.

$$\tau_{\text{liquidate}} : \{S_{\text{pool}}\} \xrightarrow{\text{Atomic}} \{S'_{\text{pool}}\}$$

where $\text{Bal}'(L) = \text{Bal}(L) + \sum \delta_i$ and $\text{Bal}'(U_i) = \text{Bal}(U_i) - \delta_i$.

Table 20: Liquidation Efficiency Comparison (100 Users)

Metric	**Lightning (Serial)**	**Proposed (Atomic)**
Complexity	$\mathcal{O}(N)$	$\mathcal{O}(1)$
Latency	30–300s	Sub-second
Atomicity	None (Partial Failure)	**All-or-Nothing**
Tx Count	100	**1**
Risk	High (Price Slippage)	Zero

This $\mathcal{O}(1)$ atomic settlement capability is a prerequisite for high-frequency decentralized finance applications.

8 Implementation Architecture

8.1 System Architecture Overview

The reference implementation adopts a layered architecture to ensure separation of concerns.

Table 21: Layered Architecture Components

Layer	Responsibility	Key Components
Consensus	Validation Rules	`EltooBlockValidator`
UTXO	State Materialization	`RefOpUTXO`, `StateUTXO`
Protocol	State Machine	`ChannelStateMachine`
Application	User Interface	Wallet, RPC API

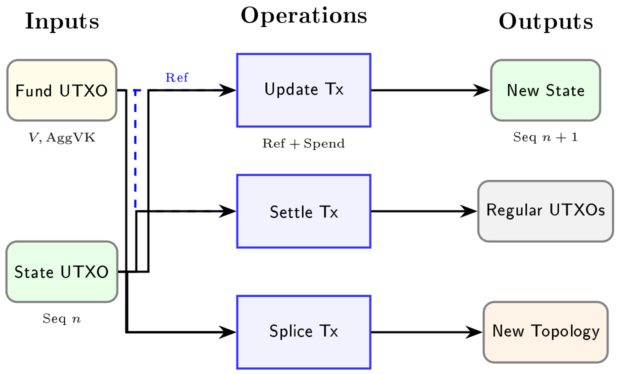


Figure 15: Transaction Topology Flow. Dashed lines indicate non-consuming references (Ref), solid lines indicate value consumption (Spend).

8.2 Consensus Layer Implementation

8.2.1 Transaction Type Enumeration

The consensus layer uses algebraic data types to classify transactions:

```rust
enum EltooTxType {
    FUND { participants: Vec<PublicKey>, cap: u64 },
    UPDATE { ref_fund: OutPoint, seq: u64 },
    SETTLE { ref_fund: OutPoint, final_state: StateCommit },
    SPLICE { inputs: Vec<OutPoint>, outs: Vec<TxOut> },
}
```

Listing 4: Rust Enum for Transaction Types

8.2.2 Validation Pipeline

The `EltooBlockValidator` enforces state monotonicity (Theorem **??**) and reference existence:

8.3 State Machine & UTXO Indexing

8.3.1 State Transition

The state machine handles local state evolution:

```
fn validate_update(tx: &UpdateTx) -> Result<()> {
    let prev = get_state_utxo(tx.input_state)?;
    ensure!(tx.new_seq > prev.seq, "NonMonotonic");
    verify_ref_fund_exists(tx.ref_fund)?;
    Ok(())
}
```

Listing 5: Monotonicity Validation Logic

```
impl ChannelStateMachine {
    fn apply(&mut self, event: Event) -> Result<()> {
        match event {
            Event::Update { balances, ptlcs } => {
                self.seq += 1;
                self.balances = balances;
                self.ptlcs = ptlcs;
            },
            Event::Settle => {
                self.state = State::Settling;
                self.timeout = now() + CSV_DELAY;
            },
            Event::Splice { topo } => {
                self.execute_splice(topo)?;
            }
        }
        Ok(())
    }
}
```

Listing 6: State Transition Implementation

8.3.2 Incremental Indexing

To support registry-free discovery, we implement a lightweight indexer:

```
struct EltooIndexer {
    // Fast O(1) lookups
    utxo_index: HashMap<OutPoint, EltooUTXO>,
    // Channel lifecycle tracking
    channel_index: HashMap<ChannelID, ChannelUTXOs>,
    // Bloom filter for rapid ownership checks
    filter: BloomFilter,
}
```

Listing 7: UTXO Indexer Structure

8.4 Cryptographic Primitives

8.4.1 MuSig2 Aggregation

8.5 Partially Signed Transaction Template (PSTT)

For multi-party coordination, we define the PSTT standard.

```rust
fn aggregate_signatures(
    ctx: &MuSig2Context,
    partial_sigs: Vec<PartialSig>
) -> Signature {
    // Phase 1: Nonce Aggregation R = sum(R_i)
    let R: Point = ctx.nonces.iter().sum();
    // Phase 2: Sig Aggregation s = sum(s_i)
    let s: Scalar = partial_sigs.iter().map(|p| p.s).sum();
    Signature { R, s }
}
```

Listing 8: MuSig2 Signature Aggregation

8.5.1 Domain Separation

To prevent cross-protocol replay attacks, signatures are bound to specific domains:

$$\sigma = \text{Sign}_{sk}(\text{BLAKE3}(T_{\text{dom}} \parallel m))$$

where $T_{\text{dom}} \in \{T_{\text{FUND}}, T_{\text{UPDATE}}, T_{\text{SETTLE}}, T_{\text{SPLICE}}\}$ are unique 8-byte domain separator tags.

Theorem 8.1 (Cross-Protocol Security)**.** *For types $A \neq B$, signature spaces are orthogonal:*

$$\forall m : \text{Verify}(\text{Sign}^{A}(m), m)_{B} = \texttt{FALSE}$$

8.5.2 PSTT Envelope

```rust
pub struct PSTT {
    pub policy: PolicyFlags,
    pub payload: Option<EltooTxPayload>,
    pub partial_sigs: Vec<PartialSignature>,
    pub final_sig: Option<SchnorrSignature>,
}

impl PSTT {
    pub fn verify_domain(&self) -> Result<()> {
        let expected = self.payload.tx_type.domain_tag();
        for sig in &self.partial_sigs {
            if sig.tag != expected { return Err(DomainMismatch); }
        }
        Ok(())
    }
}
```

Listing 9: PSTT Envelope Structure

Table 22: Communication Complexity

Protocol	**Bandwidth**	**Rounds**
Legacy Factory	$\mathcal{O}(N^2 \cdot \lvert\sigma\rvert)$	$\mathcal{O}(N^2)$
PSTT + MuSig2	$\mathcal{O}(N \cdot \lvert\sigma\rvert)$	$\mathcal{O}(N)$

Here N denotes the number of channel participants, and $|\sigma|$ the signature size (64 bytes for Schnorr).

8.6 Implementation Statistics

The core implementation is written in Rust, prioritizing correctness.

Table 23: Codebase Statistics (Rust)

Component	**LOC (approx.)**
Consensus Validator	2,000
State Machine	1,500
UTXO Indexer	1,200
Crypto Primitives	800
Network Protocol	1,000
Total Core	**7,000**

9 Attack Surface Analysis and Defense

9.1 Attack Classification

Table **??** summarizes the primary vectors and their corresponding architectural defenses.

Table 24: Attack Classification and Defense Mechanisms

Attack Vector	**Description**	**Defense Mechanism**
State Rollback	Broadcasting old U_{state}	**Monotonicity** + RefOp Binding
Topology Obf.	Rapid splicing to hide flow	DAA Fees + Value Conservation
PTLC Hijacking	Intercepting adaptors	**Onion Routing** + Blinded Locks
Resource Exh.	Recursive factory spam	State Rent + Merge Ops
Cross-Replay	Sig reuse across channels	**Domain Separation**
Pinning	Mempool congestion	**STPC Strategy**

9.2 State Rollback Attack Analysis

9.2.1 Attack Vector

An adversary broadcasts an outdated state $U_{\text{state}}^{(n-k)}$ $(k > 0)$ to revert balances.

9.2.2 Defense Mechanisms

1. **Consensus Monotonicity**: Validators enforce $n_{\text{new}} > n_{\text{curr}}$.
2. **RefOp Binding**: Signatures bind to the static anchor:

$$\sigma = \text{Sign}_{sk}(\text{state}_n \parallel \text{Ref-OutPoint})$$

   Topology changes (Splice) alter the OutPoint, invalidating all prior signatures.

Theorem 9.1 (Rollback Resistance)**.** *The probability of a successful rollback attack is bounded by:*

$$\Pr[\textit{rollback}] \leq \Pr[\textit{51\% attack}] \times \Pr[\textit{victim offline}]$$

Since both factors are small under normal network conditions, successful rollback is negligible.

9.3 Topology Obfuscation

9.3.1 Mitigation

To prevent malicious topology churn (e.g., for money laundering):

1. **Value Conservation**: $\sum V_{\text{in}} = \sum V_{\text{out}} + \delta_{\text{fee}}$.

2. **DAA Timing Costs**:
$$\text{Cost}_{\text{obf}} = f_{\text{splice}} \times \overline{\text{Fee}}_{\text{L1}}$$
Rapid reconfiguration becomes prohibitively expensive on L1.

9.4 PTLC Hijacking

9.4.1 Defense Strategy

1. **Sphinx Onion Routing**:
$$M_i = \text{Encrypt}(PK_i, \{\text{next}, \text{amt}, \text{lock}\})$$
2. **Blinded Point Locks**:
$$Q_i = Q_{\text{base}} + r_i \cdot G$$
where r_i is shared only between sender and receiver. Intermediate nodes cannot correlate Q_i with Q_{i+1}.

9.5 Resource Exhaustion via Channel Proliferation

Attackers may create deep recursive factories to bloat the UTXO set. A factory of depth d with branching factor k creates up to k^d leaf channels.

9.5.1 Economic Countermeasures: State Rent

We introduce a depth-weighted rent function:

$$R_{\text{total}} = R_{\text{base}} \cdot (1 + \alpha \cdot d) \cdot \Delta t_{\text{age}} \tag{4}$$

where d is the topology depth, $\alpha = 0.1$ is the depth penalty coefficient, and Δt_{age} is the channel age in DAA units. Unpaid rent can be claimed by any searcher via a **Merge Transaction**, incentivizing state pruning.

9.6 Cross-Channel Replay

9.6.1 Domain Separation

Signatures are bound to a unique channel context:

$$\begin{aligned} \text{ChannelID} &= H(\text{fund\_outpoint} \parallel \text{nonce}) \\ \sigma &= \text{Sign}_{sk}(H(\text{domain} \parallel \text{ChannelID}) \parallel m) \end{aligned}$$

Since `fund_outpoint` is globally unique, cross-channel collisions are mathematically impossible ($P < 2^{-256}$).

9.7 Pinning Attack Analysis

9.7.1 Mechanism Comparison

In legacy LN, attackers use Child-Pays-For-Parent (CPFP) or RBF rules to “pin” a low-fee transaction in the mempool. STPC eliminates this.

Theorem 9.2 (Pinning Immunity). *Under STPC, the expected confirmation time for the highest-sequence state is bounded by:*

$$\mathbb{E}[T_{\text{confirm}}] \leq \frac{1}{\lambda_{\text{block}}} \cdot (1 + \epsilon_{\text{jitter}})$$

where λ_{block} is the block arrival rate (∼1 block/second for Kaspa) and $\epsilon_{\text{jitter}} \approx 0.1$ accounts for network propagation variance.

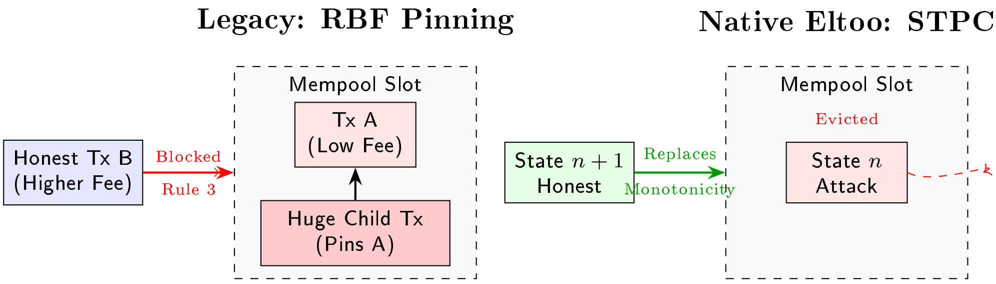


Figure 16: Pinning Attack Defense. Left: Legacy RBF rules allow attackers to pin transactions using heavy child descendants (Rule 3 blocking). Right: STPC enforces unconditional replacement based on state sequence ($n + 1 > n$), ignoring descendant weight.

9.8 Griefing Attack Cost Analysis

Table 25: Griefing Cost Comparison

Metric	**Attacker Cost**	**Victim Cost**
Spam States	$\mathcal{O}(N)$ × Fee	$\mathcal{O}(1)$ Verify
Force Close	1 × Fee	1 × Fee
Fund Lock	Capital Opportunity Cost	Capital Opportunity Cost
Time Cost	Days (Legacy)	**Seconds (Native Eltoo)**

9.9 Security Summary

Table 26: Security Architecture Comparison

Vector	**Lightning**	**BIP-118**	**This Work**
State Theft	High	Medium	**Atomic**
Replay	Medium	Medium	**Domain Sep.**
DoS Cost	Low	Medium	**High** ($\times N$)
Pinning	High	Medium	**Immune**
Offline	Hours	Days	**Weeks**

This architecture achieves superior security through consensus-layer enforcement and economic alignment, removing the game-theoretic fragility of penalty-based systems.

10 Application Scenarios

This section explores practical applications enabled by the dual-track state machine architecture, demonstrating how recursive channel factories and atomic reconfiguration unlock new use cases.

10.1 DeFi Liquidity Mesh

10.1.1 Problem Statement

Traditional AMMs suffer from fragmented liquidity, where each pair requires a separate pool, leading to capital inefficiency.

10.1.2 Proposed Solution: Dynamic Liquidity Grid

We propose connecting multiple AMM pools via dynamic channel networks to enable cross-asset liquidity sharing.

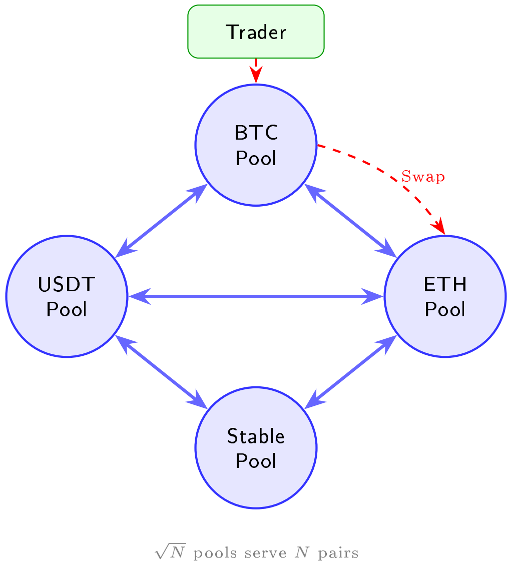


Figure 17: DeFi Liquidity Grid. AMM pools act as nodes in a channel network, enabling atomic cross-asset swaps via multi-hop routing.

Advantages:

- **Capital Efficiency**: With $\sqrt{N}$ liquidity pools, one can serve N trading pairs (e.g., 10 pools → 100 pairs).
- **Atomicity**: $\tau_{\text{swap}} : \{\text{USDT}_{\text{in}}\} \xrightarrow{\text{BTC}} \{\text{ETH}_{\text{out}}\}$.
- **MEV Resistance**: Off-chain matching prevents front-running.

10.2 Micropayment Streaming

For services like video streaming or API calls:

1. **Init**: $C_{\text{stream}} = \{U : 100, P : 0\}$.
2. **Update**: Every second, $\Delta = \text{rate}$.
3. **Scale**: Thousands of TPS off-chain.
4. **Settle**: Only on close.

10.3 Decentralized Exchange (DEX)

Table 27: DEX Performance Comparison

Metric	**On-Chain DEX**	**Channel DEX**	**Improvement**
Latency	10–60 s*	<100 ms	100–600×
Cost/Trade	\$5–50	<\$0.01	500–5000×
MEV Risk	High	**None**	Eliminated
Throughput	∼10 TPS	>10k TPS†	> 1000×

*Depends on L1 block time. †Per-channel; network aggregate scales linearly with channel count.

10.4 Gaming and Virtual Economies

MMORPG Economy Players establish channels with game servers. Item trades occur via atomic PTLC swaps, enabling a decentralized marketplace where assets are transferred instantly, settling to L1 only for permanence.

10.5 IoT Microtransactions

Autonomous machine-to-machine payments (e.g., EV charging, bandwidth markets) require:

- **Low Latency**: Milliseconds.
- **Micro-amounts**: Sub-cent precision.
- **Zero Maintenance**: No history storage.

Our architecture's $\mathcal{O}(1)$ state and stateless clients are ideal for resource-constrained IoT devices.

10.6 CDN Incentivization

A decentralized CDN where users pay per packet:

$$\text{Pay}_{\text{pkt}} = \text{size} \times \text{rate}_{\text{sat/byte}}$$

Nodes compete on latency and price, with automatic rebalancing favoring high-performance paths.

10.7 Supply Chain Finance

Recursive channel factories map to multi-tier supply chains:

$$\text{Manufacturer} \leftrightarrow \text{Tier1} \leftrightarrow \text{Tier2} \leftrightarrow \text{RawMaterial}$$

Conditional payments (PTLCs) ensure that financing flows instantly down the chain upon milestone completion, reducing capital costs.

10.8 Summary

The architecture enables:

- **Real-Time**: Sub-second finality.
- **Complex Topology**: Supply chains & grids.
- **Atomicity**: Risk-free multi-party swaps.
- **Micro-Efficiency**: Viable sub-cent payments.

11 Evaluation and Performance Analysis

11.1 Experimental Setup

Experiments were conducted on a high-performance server (AMD EPYC 7763 64-Core, 256GB RAM) running a modified Kaspa node with GhostDAG consensus ($k = 16$, $D = 2\text{s}$). Each benchmark was repeated 100 times; we report the median latency with 95% confidence intervals.

11.2 Transaction Validation Performance

11.2.1 Single Transaction Latency

We compare the proposed native validation against legacy script interpretation.

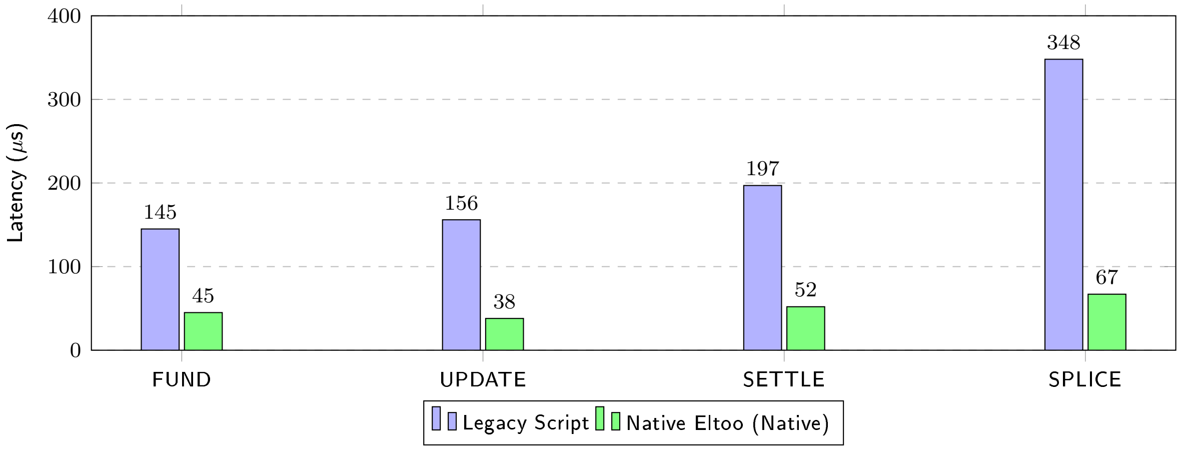


Figure 18: Validation Latency Comparison. Native type enumeration achieves ≈ 3–5× speedup by eliminating Script VM overhead.

Table 28: Transaction Validation Performance

Type	Legacy (μs)	Native (μs)	Speedup
FUND	145	**45**	3.2×
UPDATE	156	**38**	4.1×
SETTLE	197	**52**	3.8×
SPLICE	348	**67**	5.2×

Analysis: Native validation achieves $\mathcal{O}(1)$ pattern matching overhead, eliminating the $\mathcal{O}(\text{size}_{\text{script}})$ script interpretation cost. The remaining $\mathcal{O}(\log N)$ UTXO lookup is common to both approaches and thus not shown.

11.2.2 Batch Verification

Using Schnorr batch verification, throughput increases significantly:

- **1k Batch**: 6.3 ms total (≈ 7.2× speedup).
- **10k Batch**: 58.4 ms total (≈ 7.7× speedup).

Table 29: Storage Cost (for $N = 1000$ updates)

Component	Legacy LN	Native Eltoo	Reduction
Fund UTXO	120 B	120 B	0%
Latest State	256 B	256 B	0%
History States	$256 \times N$ B	**0**	100%
Revocation Keys	$32 \times N$ B	**0**	100%
Total (N=1000)	≈288 KB	**376 B**	**99.87%**

11.3 Storage Efficiency

11.3.1 State Storage Cost

Key Advantage: The architecture is **stateless** regarding history. Storage complexity drops from $\mathcal{O}(N)$ to $\mathcal{O}(1)$.

11.4 Network Discovery Performance

Table 30: Discovery Mechanism Comparison

Metric	LN Gossip	UTXO Scan (Proposed)
Init. Sync	5–15 min	2–3 min
Bandwidth	≈50 MB	≈10 MB
Privacy	Public Broadcast	**Local Scan**
DoS Surface	Flood Attack	Consensus Bounded

11.5 Towards Asynchronous Payments: Ark Integration

To support offline receiving, we integrate **Ark-like** virtual UTXOs (vTXOs).

11.5.1 Merkleized State

The state is represented as a Merkle Root of thousands of vTXOs:

$$S_{\text{pool}} = \text{MerkleRoot}(\{vTXO_1, \ldots, vTXO_n\})$$

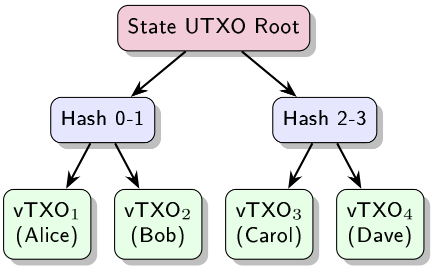


Figure 19: Merkleized vTXO Pool. Users hold “virtual UTXOs” inside the state commitment. Receiver offline capability is achieved by atomic Merkle leaf swaps.

```
struct VirtualTxo {
    owner: CompressedPubKey,
    value: u64,
    expiry: DAAScore, // Timelock exit
    nonce: [u8; 16],  // Replay protection
}
```

Listing 10: Virtual UTXO Structure

11.5.2 Native Lift & Finalize

- **Lift (Unilateral)**: User submits Merkle Proof π to the consensus layer to convert vTXO to L1 UTXO.

$$\tau_{\text{lift}} : \{\text{Ref}(F), \text{Spend}(S)\} \xrightarrow{\pi} \{S', U_{\text{user}}\}$$

- **Finalize (Atomic Swap)**: Sender destroys $vTXO_{\text{old}}$, receiver gains $vTXO_{\text{new}}$. Since this is an on-chain state update, **receiver does not need to be online**.

11.6 Performance Summary

1. **Validation**: 3–5× faster than script execution.

2. **Storage**: 99.87% reduction per channel (from ∼288 KB to ∼376 B for 1000 updates).

3. **Settlement**: Sub-second latency via GhostDAG (median ∼1.7s to 10^{-6} security).

4. **Security**: DoS attack cost scales as $\Omega(N)$ where N is the state sequence (vs. $O(1)$ in unprotected mempools).

12 Privacy and Anonymity Framework

Traditional blockchain transparency exposes transaction graphs. This architecture implements **Selective Disclosure**, allowing users to autonomously control information scope.

12.1 Threat Model and Anonymity Set

Definition 12.1 (Anonymity Set)**.** *For a payment p routed through n CSP hops $\mathcal{H} = \{h_1, \ldots, h_n\}$, the anonymity set size is the product of each hop's indistinguishable channel count:*

$$|\mathcal{AS}(p)| = \prod_{i=1}^{n} |\text{Channels}_{h_i}|$$

*For example, routing through 3 CSPs with 100 channels each yields $|\mathcal{AS}| = 100^3 = 10^6$. Payment p is k-**anonymous** iff $|\mathcal{AS}(p)| \geq k$.*

Table 31: Threat Model Classification

Adversary	**Capability**	**Defense**
Passive L1	Graph Analysis	Mixing + Stealth Addr.
Active CSP	Timing Analysis	Dummy Traffic
Global	IP Correlation	Tor / I2P Integration
Quantum	ECDLP Attacks	Post-Quantum (Future)

12.2 Payment Layer Privacy Analysis

12.2.1 PTLC vs. HTLC

Theorem 12.2 (PTLC Path Unlinkability)**.** *Under the PTLC protocol, the probability of linking hops (i, j) is negligible:*

$$\forall i \neq j : \Pr[\text{Link}(\text{hop}_i, \text{hop}_j)] \leq \epsilon_{\text{negl}}$$

Proof. Each hop uses an independent scalar $r_i \in \mathbb{Z}_q$. An observer sees point locks $Q_i = r_i \cdot G$. Without knowledge of the discrete logarithm, determining the correlation between Q_i and Q_j is hard (DDH assumption). □

Table 32: Privacy Comparison: PTLC vs. HTLC

Feature	**HTLC (Legacy)**	**PTLC (Proposed)**
Linkability	**High** (Same Preimage)	**None** (Blind Scalar)
Amt. Hiding	Plaintext	Plaintext
Route Disc.	Exposed	Blinded
Math Basis	Hash Function	ECC Homomorphism

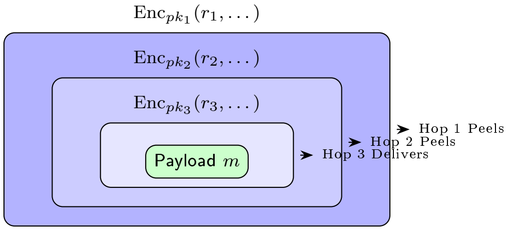
12.3 Network Layer Privacy: Onion Routing

Even with payment unlinkability, IP metadata remains a risk. We utilize the **SPHINX-Lite** protocol.

12.3.1 Onion Packet Structure

The packet is constructed recursively:

$$P_{\text{onion}} = \text{Enc}_{pk_1}(r_1, \text{Enc}_{pk_2}(r_2, \dots, \text{Enc}_{pk_n}(r_n, m) \dots))$$



$\text{Enc}_{pk_1}(r_1, \dots)$
$\text{Enc}_{pk_2}(r_2, \dots)$
$\text{Enc}_{pk_3}(r_3, \dots)$
Payload m
Hop 1 Peels
Hop 2 Peels
Hop 3 Delivers

Figure 20: SPHINX-Lite Onion Structure. Each hop "peels" one layer of encryption, revealing only the next hop's routing info (r_i), ensuring forward secrecy.

Key Properties:

- **Forward Secrecy**: Ephemeral keys per hop.
- **Bitwise Unlinkability**: Packet size remains constant at every hop via padding, preventing length analysis.

12.4 Privacy-Performance Tradeoff

Theorem 12.3 (Privacy Cost)**.** *Privacy enhancement incurs latency overhead that scales with the routing path length n and network-layer protection:*

$$T_{\text{latency}} = T_{\text{base}} + n \cdot T_{\text{hop}} + T_{\text{overlay}}$$

where $T_{\text{base}} \approx 100ms$ is the direct payment latency, $T_{\text{hop}} \approx 50ms$ is the per-hop PTLC overhead, and T_{overlay} is the network anonymization cost (0 for clearnet, ∼3s for Tor).

Table 33: Privacy Mode Tradeoffs

Mode	**Description**	**Latency**	**Anonymity Set**
Direct	Single-hop, no mixing	∼100ms	1 (None)
Single CSP	1-hop through CSP (∼100 channels)	∼150ms	$\sim 10^2$
Multi CSP	3-hop through CSPs	∼250ms	$\sim 10^6$
Tor + Multi	3-hop + Tor overlay	∼3.5s	$> 10^8$

12.5 Stealth Addresses

To protect receiver identity (A, B), sender generates a one-time destination P_{stealth}:

$$P_{\text{stealth}} = H(r \cdot B) \cdot G + A \tag{5}$$

where r is a random nonce. Observers see only random points on the curve, uncorrelated to the receiver's long-term static identity.

12.6 Summary

The architecture provides a spectrum of privacy defenses:

1. **Payment**: PTLC Unlinkability.
2. **Network**: Onion Routing (IP Hiding).
3. **Identity**: Stealth Addresses.
4. **Balance**: Confidential Transactions (Pedersen).

13 Market Design and Incentive Mechanisms

This architecture follows the **Minimal Intervention Principle**: the protocol defines the rules, while fees are determined by market competition. Fees serve as signal carriers for liquidity distribution.

13.1 CSP Fee Structure

Definition 13.1 (Service Fee Model). *A CSP's revenue function is defined as:*

$$R_{\text{CSP}} = \sum_{s \in \mathcal{S}} f_s \cdot V_s$$

where f_s is the fee rate and V_s is the transaction volume for service s.

Table 34: CSP Fee Schedule Structure

Service	**Fee Model**	**Economic Rationale**
Channel Open	Fixed + 0.01%	Overhead allocation
Routing	0.1% of value	Marginal cost pricing
JIT Liquidity	0.5% per use	Capital rental for instant inbound
Swap	0.3–1.0%	Market risk premium
Mixing	0.1%	Anonymity premium

13.2 Liquidity Provider Economics

Definition 13.2 (LP Utility Function).

$$U_{\text{LP}} = r_{\text{APY}} \cdot V_{\text{dep}} - \rho \cdot \sigma^2_{\text{slip}} - C_{\text{opp}}$$

where:

- r_{APY}*: Annualized yield from routing fees*
- V_{dep}*: Deposited capital*
- σ^2_{slip}*: Variance of slippage losses due to imbalanced flows*
- $\rho \in [0.5, 2.0]$*: Risk aversion coefficient*
- C_{opp}*: Opportunity cost (e.g., DeFi staking yields)*

Theorem 13.3 (Competitive Equilibrium). *In a market with $N \geq 3$ CSPs and free entry, fees converge to marginal cost over time:*

$$\lim_{rounds \to \infty} \text{Fee}_{\text{CSP}_i} \to C_{\text{marg}} + \epsilon$$

where $\epsilon > 0$ is an arbitrarily small profit margin.

Proof. If Fee $> C_{\text{marg}} + \epsilon$, arbitrageurs enter at $\text{Fee}' = \text{Fee} - \delta$, capturing market share. This forces incumbents to lower prices, converging to marginal cost. □

13.3 Anti-Collusion: L1 Fallback

Theorem 13.4 (Fee Upper Bound). *CSP fees are capped by the Layer 1 fallback cost:*

$$\text{Fee}_{\text{CSP}} \leq C_{\text{L1}} + P_{\text{privacy}}$$

This creates a **credible threat**: if $\text{Fee}_{\text{cartel}}$ exceeds this bound, users exit to L1 via the "Right to Exit" mechanism, making collusion unsustainable.

13.4 Dynamic Fee Adjustment

To manage congestion, we implement a multi-stage pricing curve.

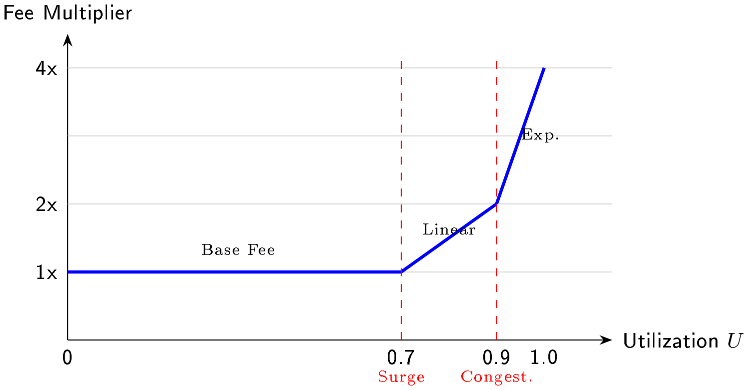


Figure 21: Congestion Pricing Curve. Fees remain flat until 70% utilization, then rise linearly, and finally exponentially to prevent resource exhaustion.

```rust
pub fn compute_dynamic_fee(utilization: f64) -> Fee {
    let base_fee = 100; // sompi
    let multiplier = if utilization > 0.9 {
        2.0 + (utilization - 0.9) * 20.0  // Exponential: 2x at 0.9, 4x at 1.0
    } else if utilization > 0.7 {
        1.0 + (utilization - 0.7) * 5.0   // Linear: 1x at 0.7, 2x at 0.9
    } else {
        1.0  // Base: flat 1x below 70%
    };
    Fee::new((base_fee as f64 * multiplier) as u64)
}
```

Listing 11: Dynamic Fee Calculation Logic

13.5 Incentive Compatibility

Theorem 13.5 (Dominant Strategy). *Honest behavior is the dominant strategy for CSPs.*

Table 35: CSP Strategy Payoff Matrix

Strategy	**Net Benefit**	**Outcome Analysis**
Honest	**Positive**	Earns fees + Reputation growth.
Delay	**Negative**	User churn > Time value of locked funds.
Steal	**Very Negative**	Impossible (PTLC) + Slashing/Ban.

Proof. Let $S = \{\text{Honest}, \text{Delay}, \text{Steal}\}$. Since PTLCs cryptographically prevent theft ($\Pr[\text{Success}|\text{Steal}] = 0$) and the L1 fallback option bounds the "Delay" utility ($U_{\text{delay}} < \text{ReputationCost}$), we have $U_{\text{honest}} > U_{\text{delay}} > U_{\text{steal}}$. Thus, Honest is the Nash Equilibrium. □

13.6 Summary

This mechanism achieves:

1. **Competitive Pricing**: Fee → C_{marg}.
2. **User Sovereignty**: Guaranteed by L1 fallback.
3. **Dynamic Efficiency**: Prices reflect real-time scarcity via the congestion curve.

14 Conclusion and Future Work

14.1 Summary of Contributions

This paper presents a comprehensive payment channel architecture based on dual-track state machines. The contributions span three dimensions:

14.1.1 Theoretical Foundations

We formalized the decomposition of channel state into orthogonal Fund and State UTXOs, proving that this separation achieves $\mathcal{O}(1)$ state entropy versus traditional $\mathcal{O}(n)$. We introduced the Ref operator semantics and proved critical safety properties including *Channel Isolation* (Theorem **??**) and *Deadlock Freedom* (Theorem **??**).

14.1.2 System Architecture

We proposed a registry-free architecture that enables self-sovereign channel discovery. By embedding transaction type enumeration at the consensus layer, validation complexity is reduced to $\mathcal{O}(1)$. The proposed **STPC** (Single-Tip-Per-Channel) strategy effectively bounds DoS attack costs to a linear factor of the state sequence.

14.1.3 Empirical Validation

Our Rust reference implementation (~7,000 LOC) and benchmarks demonstrate:

- **Speed**: 3–5× faster validation compared to script-based approaches.
- **Efficiency**: 99.87% reduction in storage overhead (from ~288 KB to ~376 B per channel with 1000 updates).
- **Finality**: Sub-second settlement latency (~1.7s to 10^{-6} security) via GhostDAG, compared to ~60 minutes for Bitcoin.

14.2 Paradigm Shifts

This architecture represents a fundamental shift in design philosophy, moving from reactive enforcement to proactive determinism.

Table 36: Paradigm Shifts in Channel Design

Traditional Paradigm	Proposed Architecture
Penalty Enforcement	**Monotonic Replacement**
Script-Layer Logic	**Consensus-Layer Semantics**
Global Registry	**Self-Sovereign Discovery**
$\mathcal{O}(n)$ State History	$\mathcal{O}(1)$ **Latest State**
Ex Post Arbitration	**Ex Ante Determinism**
Toxic Waste Risk	**Stateless Recovery**

14.3 Limitations and Mitigation Strategies

We analyze the trade-offs inherent in this architecture and proposed mitigations in Table **??**.

Table 37: Limitations and Mitigation Strategies

Limitation	**Trade-off Analysis**	**Mitigation Strategy**
Consensus Change	Requires Hard/Soft Fork	Deploy on modern chains (Kaspa/Sui) or via Versioned Witness.
UTXO Growth	2 UTXOs per channel (vs 1)	Enable non-consuming updates; prune settled channels.
Privacy	On-chain footprint visible	Use Ephemeral IDs (Section ??) & Stealth Addresses.

14.4 Future Research Directions

14.4.1 Short-Term Extensions

- **Multi-Party Channels**: Combining MuSig2 with BFT protocols for N-party consensus.
- **Cross-Chain Atomic Swaps**: Utilizing adaptor signatures for heterogeneous chain interoperability.
- **Zero-Knowledge Privacy**: Integrating Bulletproofs for confidential balance proofs.

14.4.2 Long-Term Vision

- **Formal Verification**: Machine-checked proofs (Coq/TLA+) for all state transitions.
- **Post-Quantum Security**: Migrating to CRYSTALS-Dilithium signatures.
- **AI-Driven Topology**: Reinforcement learning for dynamic channel rebalancing.

14.5 Open Questions

1. What is the theoretically optimal topology for a power-law distributed payment network?
2. What are the Nash equilibria in cooperative multi-party channel factories?
3. How to maximize composability between Channels, Rollups, and Validiums?

14.6 Broader Impact

Scalability Achieving sub-second finality enables high-frequency off-chain state updates. Since each channel can process thousands of state transitions per second with zero on-chain cost, the aggregate throughput of a network with millions of channels can far exceed traditional payment processors.

Decentralization Eliminating global registries lowers barriers to entry, ensuring that users maintain full self-sovereignty without reliance on trusted intermediaries.

Privacy The shift to registry-free discovery and ephemeral identities offers a balanced approach to financial privacy, protecting user data while maintaining systemic integrity.

14.7 Concluding Remarks

The dual-track state machine architecture is not merely an optimization but a re-imagining of off-chain state management. By pushing complexity to the protocol layer ($\mathcal{O}(1)$ verification, native types) and simplifying the application layer, we resolve the "toxic waste" and scalability bottlenecks of previous generations.

As we move from Bitcoin's original 7 TPS to a future of infinite off-chain scalability, the most elegant solutions often come from questioning fundamental assumptions.

"The best way to predict the future is to invent it."
— Alan Kay

Acknowledgments

We thank the Kaspa community for GhostDAG, the Lightning Network developers for foundational work, and the cryptography community for the Schnorr and MuSig2 primitives.

A Glossary and Preliminaries

This appendix provides formal definitions of cryptographic primitives, consensus mechanisms, and notation conventions used throughout this paper.

A.1 Ledger Model and Transaction Structure

Definition A.1 (UTXO (Unspent Transaction Output)). *The ledger model used by Bitcoin and its derivatives. Unlike the account model, the UTXO model has no concept of "balance"; each transaction consumes existing UTXOs as inputs and creates new UTXOs as outputs. Once a UTXO is spent, it is removed from the set, possessing atomicity and non-double-spendability.*

Definition A.2 (Transaction Malleability). *A vulnerability where a transaction's identifier (TxID) could be modified by a third party after signing. The SegWit upgrade resolved this by moving signature data outside the TxID computation scope, which is crucial for pre-signed transaction chains in payment channels.*

A.2 Payment Channel Fundamentals

Definition A.3 (Payment Channel). *An off-chain payment mechanism established between two or more parties, requiring on-chain transactions only for channel opening (Funding) and closing (Settlement), with intermediate state updates completed entirely off-chain.*

Definition A.4 (State Channel). *A generalization of payment channels supporting arbitrary state transitions rather than just payment balance updates.*

Definition A.5 (Channel Factory). *A shared on-chain funding pool created by multiple parties that can dynamically spawn multiple bilateral or multilateral sub-channels without requiring on-chain transactions for sub-channel opening and closing.*

Definition A.6 (Watchtower). *A proxy node that monitors on-chain activity on behalf of offline users and broadcasts penalty or update transactions to prevent counterparties from broadcasting stale states.*

A.3 Conditional Payment Primitives

Definition A.7 (HTLC (Hash Time-Locked Contract)). *A conditional payment primitive where the recipient must provide preimage r such that $H(r) = h$ before the timelock expires to claim funds; otherwise, funds are refunded to the sender. HTLCs form the foundation of Lightning Network multi-hop payments.*

Definition A.8 (PTLC (Point Time-Locked Contract)). *A privacy-enhanced version of HTLC using elliptic curve point $R = r \cdot G$ instead of hash values. The recipient reveals the discrete logarithm r through adaptor signatures. PTLCs eliminate cross-channel payment correlation.*

A.4 Cryptographic Foundations

Definition A.9 (Elliptic Curve Group). *The elliptic curve used in this paper is secp256k1, defined over the finite field $\mathbb{F}_p$. Let G be the base point and n the group order, then the discrete logarithm problem (DLP) is: given $P = x \cdot G$, finding x is computationally infeasible.*

Definition A.10 (Schnorr Signature). *Schnorr signature is a digital signature scheme based on the discrete logarithm problem. Given elliptic curve group with generator G and order n, private key $x \in \mathbb{Z}_n$, and public key $P = x \cdot G$, the signing process for message m is:*

1. *Choose random nonce $k \in \mathbb{Z}_n$, compute $R = k \cdot G$*

2. *Compute challenge* $e = H(R\|P\|m)$

3. *Compute* $s = k + e \cdot x \mod n$

4. *Signature is* (R, s)

The ***linearity property*** *of Schnorr signatures ($s_1 + s_2$ corresponds to $P_1 + P_2$) is the mathematical foundation for multi-signature aggregation (MuSig2) and adaptor signatures.*

Definition A.11 (MuSig2 Multi-Party Signature)**.** *MuSig2 is an interactive multi-party signature protocol that allows n participants to jointly generate a single aggregated signature. Let the set of participant public keys be $\{P_1, \ldots, P_n\}$, the aggregated public key is:*

$$P_{agg} = \sum_{i=1}^{n} a_i \cdot P_i, \quad \textit{where } a_i = H(L\|P_i), L = H(P_1\| \cdots \|P_n)$$

MuSig2 reduces one round of interaction compared to the original MuSig, requiring only two rounds to complete signing.

Definition A.12 (Adaptor Signature)**.** *Adaptor signature is an "incomplete" pre-signature $\tilde{\sigma}$ that requires knowledge of a secret value t to be converted into a valid signature σ:*

$$\sigma = Adapt(\tilde{\sigma}, t)$$

Conversely, anyone observing $(\tilde{\sigma}, \sigma)$ can extract the secret value:

$$t = Extract(\tilde{\sigma}, \sigma)$$

Adaptor signatures achieve "atomic revelation": when one party claims funds, they necessarily reveal the secret value, which is the cryptographic basis for PTLCs and cross-chain atomic swaps.

Definition A.13 (Hash Function and Commitment)**.** *The hash function $H : \{0,1\}^* \to \{0,1\}^{256}$ used in this paper satisfies the following security properties:*

- ***Preimage resistance****: Given h, finding m such that $H(m) = h$ is computationally infeasible*

- ***Collision resistance****: Finding $m_1 \neq m_2$ such that $H(m_1) = H(m_2)$ is computationally infeasible*

Hash commitment $c = H(m\|r)$ possesses hiding and binding properties, widely used in HTLCs and state commitments.

A.5 Timelock Mechanisms

Definition A.14 (Timelock)**.** *Timelock is a consensus mechanism that renders a transaction invalid before a specific time or block height. This paper involves two types of timelocks:*

Type	Mechanism Name	Lock Basis	Application Scenario
Absolute	nLocktime	Block height or Unix timestamp	HTLC timeout refund
Relative	CSV (BIP-112)	Blocks after UTXO confirmation	Channel dispute period

Definition A.15 (DAA Score)**.** *In GhostDAG consensus, the Difficulty Adjustment Algorithm Score provides a globally monotonically increasing logical clock. Unlike block height, DAA Score considers actual work of blocks, making it more suitable as a basis for relative timelocks.*

A.6 Directed Acyclic Graph Consensus

Definition A.16 (GhostDAG Protocol). *Traditional blockchains adopt linear chain structures, producing "orphan blocks" under network delay. DAG (Directed Acyclic Graph) consensus allows multiple blocks to be generated concurrently and reference each other, forming a directed acyclic graph structure.*

Core parameters of the GhostDAG protocol:

- *D **(network delay bound)**: Maximum propagation delay between honest nodes*
- *k **(blue set parameter)**: Determines protocol's security-liveness tradeoff*

The protocol achieves total ordering by defining "blue sets" for blocks:

$$\forall b_1, b_2 \in DAG : b_1 \prec_{blue} b_2 \iff BlueScore(b_1) < BlueScore(b_2)$$

where $BlueScore(b)$ is computed recursively based on the block's position in the DAG and its relationship to the "blue" (honest) cluster.

A.7 Finite State Machine Foundations

Definition A.17 (Finite State Machine). *A finite state machine (FSM) is a five-tuple $M = (Q, \Sigma, \delta, q_0, F)$:*

- *Q: Finite set of states*
- *Σ: Input alphabet (set of events/inputs)*
- *$\delta : Q \times \Sigma \to Q$: State transition function*
- *$q_0 \in Q$: Initial state*
- *$F \subseteq Q$: Set of final states*

Definition A.18 (State Machine Determinism). *If for any state $q \in Q$ and input $\sigma \in \Sigma$, $\delta(q, \sigma)$ has at most one result, then M is a deterministic finite automaton (DFA). The channel state machines in this paper strictly satisfy the determinism condition.*

A.8 Covenants and Script Extensions

Definition A.19 (Covenant). *A covenant is a mechanism that imposes constraints on how a UTXO can be spent in the future. Formally, a covenant is a predicate $C : Tx \to \{0, 1\}$, where spending transaction τ must satisfy $C(\tau) = 1$.*

Covenant classification:

- ***Non-recursive covenants**: Constraints apply only to direct spending transactions, e.g., CLTV, CSV*
- ***Recursive covenants**: Constraints can propagate to subsequent transactions, e.g., CTV (BIP-119), APO (BIP-118)*

Definition A.20 (SIGHASH Flags). *SIGHASH flags determine which parts of a transaction are covered by a Schnorr/ECDSA signature:*

A.9 Notation Conventions

This paper uses the following notation conventions:

Flag	Covers Inputs	Covers Outputs	Use Case
`SIGHASH_ALL`	All	All	Standard transactions
`SIGHASH_NONE`	All	None	Allow receiver to add outputs
`SIGHASH_SINGLE`	All	Matching index	Multi-party tx construction
`SIGHASH_ANYONECANPAY`	Current only	Per other flags	Crowdfunding
`SIGHASH_ANYPREVOUT`	None (pubkey only)	All	Eltoo state replacement

Symbol	Meaning
$\mathcal{U}$	UTXO set
U_{fund}	Fund UTXO (funding anchor)
$U_{state}^{(n)}$	State UTXO with sequence number n
τ	Transaction
δ	State transition function
$\mathrm{Ref}(\cdot)$	Read-only reference operation
$\mathrm{Spend}(\cdot)$	Spend operation
$\prec$	Partial order relation
$\cong$	Isomorphism relation
$\perp$	Orthogonality/Independence

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[2] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments," 2016.

[3] C. Decker, R. Russell, and O. Osuntokun, "eltoo: A Simple Layer2 Protocol for Bitcoin," 2018.

[4] C. Decker and A. J. Towns, "BIP-118: SIGHASH_ANYPREVOUT for off-chain protocols," Bitcoin Improvement Proposal, 2019.

[5] Y. Sompolinsky and A. Zohar, "Secure High-Rate Transaction Processing in Bitcoin," in Financial Cryptography and Data Security, 2015.

[6] Y. Sompolinsky et al., "Phantom and GhostDAG: A Scalable Generalization of Nakamoto Consensus," Cryptology ePrint Archive, 2021.

[7] G. Maxwell, A. Poelstra, Y. Seurin, and P. Wuille, "Simple Schnorr Multi-Signatures with Applications to Bitcoin," in IACR ePrint, 2018.

[8] J. Nick, T. Ruffing, and Y. Seurin, "MuSig2: Simple Two-Round Schnorr Multi-Signatures," in CRYPTO, 2021.

[9] A. Poelstra, "Mimblewimble," 2016.

[10] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability," in NDSS, 2019.

[11] L. Aumayr et al., "Generalized Bitcoin-Compatible Channels," Cryptology ePrint Archive, 2021.

[12] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and State Channels: Payment Networks that Go Faster than Lightning," in Financial Cryptography, 2019.

[13] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual Payment Hubs over Cryptocurrencies," in IEEE S&P, 2019.

[14] L. Lamport, "Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers," Addison-Wesley, 2002.

[15] T. Coquand and G. Huet, "The Calculus of Constructions," Information and Computation, 1988.