Generalized Payment Channel Topologies via Dual-Track State Machines and Reference-Based UTXOs

Arthur Zhang
Tondi Foundation

Neo Maxwell
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

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 two 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.

Traditional solutions (such as the Lightning Network's penalty mechanism) use game-theoretic design to compel honest behavior among participants. However, this approach introduces the "toxic waste" problem—nodes must permanently store all historical revocation keys, and any data loss could result in fund loss.

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.

1.2 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:

Table 1: Trust Model Comparison

Trust Assumption	**Traditional PCN**	**This Architecture**	**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

- 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. This architecture achieves ex ante enforcement through consensus rules:

$$\text{Traditional Mode: Contract} \xrightarrow{\text{Dispute}} \text{Arbitration} \xrightarrow{\text{Judgment}} \text{Enforcement}$$
$$\text{This Architecture: State\_UTXO} \xrightarrow{\tau_{\text{settle}}} \text{Value\_Distribution} \quad \text{(deterministic execution)}$$

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.

1.3 Trust Model Analysis

Blockchain system security is often described as “trust minimization.” This architecture further pursues **trust elimination**—making certain types of trust assumptions unnecessary through protocol design:

The core insight of this architecture 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 analyzes the evolution of payment channel protocols and structural defects of existing solutions. For formal definitions, see Appendix A.

2.1 Protocol Evolution: From Penalty to Replacement

2.1.1 Lightning Network's Penalty Mechanism

The Lightning Network [2] employs a **penalty mechanism** to resolve state rollbacks.

Mechanism: When updating from S_n to S_{n+1}, parties exchange the "revocation key" for S_n. If a party broadcasts S_n, the counterparty uses this key to sweep all funds.

Formal Expression: Let $\mathcal{R}_n$ be the set of revocation keys for state n:

$$\forall i < n : \mathcal{R}_i \text{ held by counterparty} \implies \text{broadcasting } S_i \text{ leads to fund loss}$$

Structural Defects (Toxic Waste):

1. **Storage**: Nodes must store $\mathcal{O}(n)$ historical keys.
2. **Risk**: Data loss or backup errors can lead to accidental broadcasting of old states, triggering catastrophic penalties.

2.1.2 Eltoo Protocol and State Replacement

Eltoo [3] introduces **state replacement**. Instead of punishing old states, update transaction τ_{n+1} can legally spend any τ_i ($i \leq n$).

Dependency: Originally relied on `SIGHASH_NOINPUT` (now BIP-118 `ANYPREVOUT`).

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

This hash omits the input identifier (OutPoint), binding only to the script logic.

2.1.3 Engineering Compromise of BIP-118

To mitigate replay risks, BIP-118 mandates **Public Key Tagging**.

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

This physically segregates public keys, shifting security responsibility to application-layer key management.

2.2 The Recursive Covenant Dilemma

APO introduces introspective capabilities. If script S can force its output to be locked in S' (where $S' \equiv S$), it enables **recursive covenants**. This raises concerns about fungibility (e.g., regulatory whitelisting) and "toxic recursion," stalling BIP-118 activation.

2.3 Structural Defect Analysis

Existing solutions simulate state machines by stacking opcodes, violating the **Principle of Orthogonality**.

Table 2: Structural Defects of Script-Based Solutions

Defect Type	Impact Analysis
Complexity	Validation is $\mathcal{O}(script\_size)$; resource usage is unpredictable.
Opacity	Consensus layer cannot distinguish updates from transfers; prevents L1 optimization.
Boundary Blur	Relies on key tagging; delegates security to apps.
Coupling	Value locking and state logic are entangled.

Table 3: State Revocation: Original Eltoo vs. Eltoo 2.0

Feature	Original Eltoo	Eltoo 2.0 (Proposed)
Primitive	Script overwrite + `NOINPUT`	Consensus Enum + Dual-Track
Scalability	Complex script logic	Native Factories
Data	Parsing overhead	Store latest only
Determinism	Sig-dependent	**Virtual Reference**
DoS Defense	Weak	**STPC Strategy**

2.4 Proposed Solution: UTXO-Native Semantics

We propose a **UTXO-native** approach, pushing Eltoo semantics into the transaction structure.

- **Type System**: $\forall \tau \in \mathcal{T}_{\text{Eltoo}}$, inputs MUST be of type `ELTOO_STATE`. This physically isolates replay paths at the type level.
- **FSM vs. Recursion**: Updates follow strict monotonicity ($n' > n$), mathematically precluding arbitrary recursive covenants.
- **Explicit Reference**: Ref reduces verification complexity to $\mathcal{O}(1)$.

2.5 DAG Consensus Compatibility

Adopting GhostDAG (D, k) provides:

1. **Temporal Consistency**: $\text{DAA}(b_1) < \text{DAA}(b_2)$ for $b_1 \prec b_2$.
2. **Fast Confirmation**: $E[\text{time}] = \mathcal{O}(D/k)$.
3. **Throughput**: $\text{TPS} \approx k \times \text{TPS}_{\text{single}}$.

2.6 Comparison of Revocation Mechanisms

2.7 Axiom System

2.8 BIP-118 Security Boundary Analysis

Figure 1 illustrates the leakage of security responsibility in the BIP-118 model.

The core issue is **state dependency**: if a wallet reuses a private key for both standard and APO paths, the protocol cannot prevent replay attacks, leading to fund loss. This violates the principle of "pushing complexity down to the protocol layer."

2.9 Economic Efficiency Boundary

We analyze Layer 2 protocols in a 3D space: $\Omega = \mathcal{L}_{\text{atency}} \times \mathcal{T}_{\text{hroughput}} \times \mathcal{C}_{\text{apital}}$.

Native Eltoo achieves superior economics for high-frequency flows by aggregating participants via Channel Factories, minimizing the time-value cost of capital ($\gamma \cdot C_{\text{time\_value}}$).

Table 4: Core Axiom System

ID	Formal Expression & Semantics
A1	$\mathcal{S}_{\text{channel}} \cong \mathcal{U}_{\text{chain}}$ (Isomorphism)
A2	$\forall \tau_{\text{update}} : n' > n$ (Strict Monotonicity)
A3	$\text{Ref}(U) \in \tau \implies U \in \mathcal{U}_{\text{post}}$ (Non-consumption)
A4	$\sum V_{\text{in}} = \sum V_{\text{out}} + \delta_{\text{fee}}$ (Conservation)

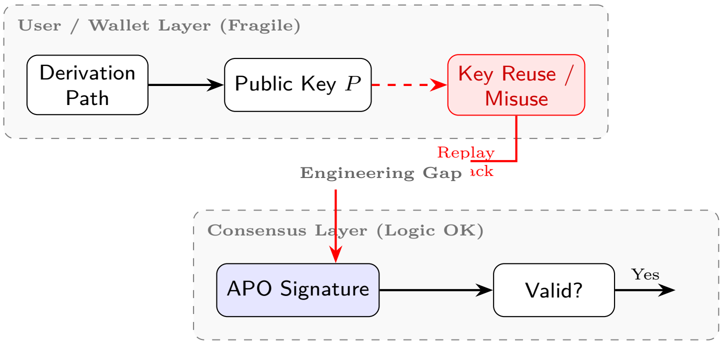

Figure 1: BIP-118 Security Boundary. The reliance on correct key derivation paths in the application layer creates an "Engineering Gap," where key reuse can lead to replay attacks despite sound consensus-layer logic.

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 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 $\{S_0, \ldots, S_{t-1}\}$.

Table 5: Economic Positioning Comparison

Protocol	Latency	Capital Eff.	Trust Model
Bitcoin L1	10–60 min	Baseline	Full Consensus
Rollups	1–15 min	Medium	Operator + L1
Lightning	Seconds	Low (Locked)	Watchtowers
Native Eltoo	**Sub-sec**	**High (Pooled)**	**Consensus**

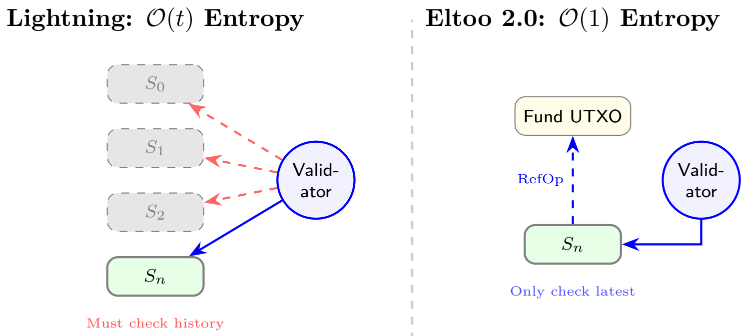

Figure 2: Verification Causality Graph. Lightning validators (left) face linear complexity due to historical dependencies. Eltoo 2.0 validators (right) only verify the latest state against the static Fund anchor.

Definition 3.1 (State Entropy)**.** *The **state entropy** $H(C)$ is the information quantity a validator must maintain:*

$$H_{\text{LN}}(t) \propto \sum_{i=0}^{t-1} \text{size}(\text{RevocationKey}_i) \approx \mathcal{O}(t) \quad (1)$$

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:

$$H_{\text{Eltoo2.0}}(t) \approx \text{size}(\text{State}_{\text{curr}}) + \text{size}(\text{Fund}) \approx \mathcal{O}(1) \quad (2)$$

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

Theorem 3.2 (Information-Theoretic Robustness)**.** *State recovery fault tolerance $\mathcal{R}$ is inversely proportional to state entropy H:*

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

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 6 presents a comprehensive comparison.

Table 6: Comprehensive Architecture Comparison

Feature	**Lightning**	**BIP-118**	**Eltoo 2.0**
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)	$\mathcal{O}(\mathbf{1})$
Storage	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\mathbf{1})$
Settle Time	Minutes	Minutes	**Sub-second**
Backup	Full History	Latest	**Latest**

Table 7: Paradigm Shift in Design Philosophy

Aspect	**Traditional**	**Proposed**
Trust Model	Penalty Deterrence	**Protocol Determinism**
State Mgmt	App Layer	**Consensus Layer**
Security	User Keys	**Rule Enforcement**
Complexity	Distributed	**Centralized (L1)**

3.5 Theoretical Significance

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

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

Table 8: 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

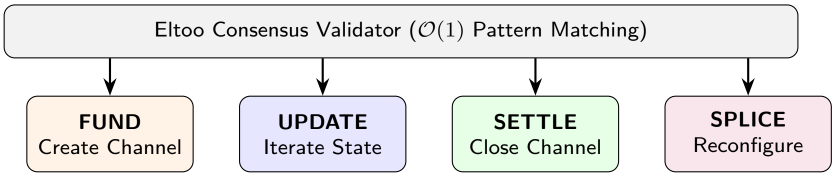

Figure 3: Transaction Type Enumeration System. The consensus layer classifies transactions via $\mathcal{O}(1)$ pattern matching on I/O topology.

4 Theoretical Framework: Dual-Track State Machines

4.1 Consensus-Layer Embedded Verification Mechanism

4.1.1 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:

4.1.2 State Monotonicity Theorem and Consensus Implementation

Theorem 4.1 (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. □

Table 9: 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

4.1.3 Consensus Verification Performance Analysis

Since 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 ($\mathcal{O}(1)$), total verification complexity is only $\mathcal{O}(\log N)$ (UTXO lookup). Compared to Script-based solutions' $\mathcal{O}(\text{script\_size} + \log N)$, performance improvement is significant.

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

Corollary 4.2 (Scalability)**.** *Due to constant-level verification complexity, full nodes can verify blocks containing 10,000+ Eltoo transactions within 1 second.*

4.1.4 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 4.3 (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*

Definition 4.4 (Active State Lease)**.** *We introduce the concept of **Active State Lease** in the UTXO set:*

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

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

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

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

Table 10: 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

2. Since `UPDATE` transactions consume the unique $U_{state}^{(n)}$, 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

□

4.1.5 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 4.6 (Valid Reference Window). *Let τ_{update} be confirmed in block B_i, generating $U_{state}^{(n)}$. Let τ_{settle} be broadcast in block B_j, referencing $U_{state}^{(n)}$. τ_{settle} is valid if and only if:*

1. *$B_i \in Past(B_j)$ (DAG topological order)*

2. *$U_{state}^{(n)}$ is in "unspent" status in B_j's UTXO view set*

Theorem 4.7 (Anchoring Persistence). *As long as no new `UPDATE` transaction τ'_{update} overwrites $U_{state}^{(n)}$, that state UTXO will persist in the ledger:*

$$\forall t \in [t_{confirm}, \infty) : \nexists \tau'_{update} \implies U_{state}^{(n)} \in \mathcal{U}_{chain}(t)$$

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

4.1.6 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 4.8 (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 $\tau \to \perp$ (non-spendable), serving only as an oracle providing metadata access to U_{fund}.

Table 11: 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
$\perp$ case	`ConsensusError::InvalidEltooTxType`	Reject invalid transactions

Definition 4.9 (Type Inference Homomorphism)**.** *Define function* $\Gamma : \mathcal{I}^* \times \mathcal{O}^* \to \mathcal{T}_{Eltoo} \cup \{\perp\}$*, which maps transaction I/O topology to semantic types in* $\mathcal{O}(1)$ *time complexity:*

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

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)
- $\perp$: Matches no pattern, transaction rejected

Theorem 4.10 (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.*

4.2 Finite State Machine Formalization

We define channel C as a **Deterministic Finite Automaton (DFA)**:

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

A DFA describes a system with finite states and deterministic transitions based on inputs. **Component Details**:

- Q: State space. $Q = \{q_{init}\} \cup Q_{active} \cup Q_{settling} \cup \{q_{closed}\}$
  - $Q_{active} = \{(n, R_b, R_p) \mid n \in \mathbb{N}, R_b \in \mathcal{H}, R_p \in \mathcal{H}\}$ — Active state set
  - $Q_{settling} = \{(n, R_b, R_p, t) \mid t \in \mathbb{N}_{DAA}\}$ — Settlement waiting state set
- Σ: Transaction alphabet. $\Sigma = \{\tau_{fund}, \tau_{update}, \tau_{splice}, \tau_{settle}, \tau_{timeout}\}$
- δ: State transition function. $\delta : Q \times \Sigma \rightharpoonup Q$ (partial function)

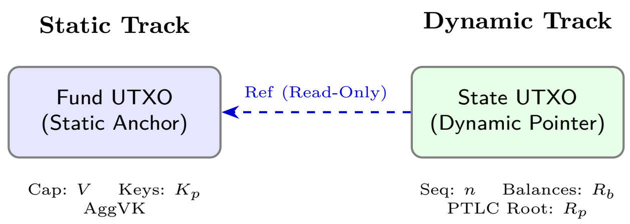

Figure 4: Dual-Track State Machine. Separation of static funding capability from dynamic state evolution.

Table 12: 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

- q_0: Initial state. $q_0 = q_{init}$
- F: Final state set. $F = \{q_{closed}\}$

Definition 4.11 (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.*

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.

Mathematical Formalization:

$$\mathcal{M} : Q \to \mathcal{P}(\mathcal{U})$$

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

Semantic Interpretation:
Where:

- U_{fund}: Static Anchor
  - 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 $n \in \mathbb{N}$

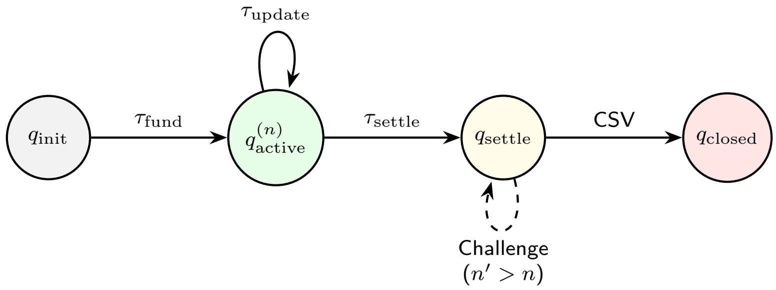

Figure 5: Channel State Machine Transitions.

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

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

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

The RefOp 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.

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`.

4.4 State Transition Rules

Definition 4.13 (Transition Function)**.** *δ is defined by the following rules:*

$$\begin{aligned}
\delta(q_{init}, \tau_{fund}) &= q^{(0)}_{active} && \textit{[FUND]} \\
\delta(q^{(n)}_{active}, \tau_{update}) &= q^{(n+k)}_{active} && \textit{where } k > 0 \quad \textit{[UPDATE]} \\
\delta(q^{(n)}_{active}, \tau_{splice}) &= \{q^{(n')}_{active}, q^{(0)}_{child}\} && \textit{[SPLICE]} \\
\delta(q^{(n)}_{active}, \tau_{settle}) &= q^{(n,t)}_{settling} && \textit{[SETTLE-INIT]} \\
\delta(q^{(n,t)}_{settling}, \tau_{timeout}) &= q_{closed} && \textit{when } t_{now} - t \geq CSV \quad \textit{[SETTLE-FINAL]}
\end{aligned}$$

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

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

4.5 Formal Safety Properties

The following properties can be formally verified through TLA+ or Coq:

Theorem 4.14 (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.

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

Theorem 4.15 (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.

Proof. Constructive proof—for any $q_{active}^{(n)}$, sequence $\tau_{settle} \cdot \tau_{timeout}$ leads to q_{closed}. □

Theorem 4.16 (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).

Theorem 4.17 (Value Conservation)**.**

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

4.6 Transaction Semantics Mapping

Mapping between abstract transitions and concrete UTXO operations:

Fund Transaction:

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

Update Transaction:

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

Splice Transaction:

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

Settle Transaction:

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

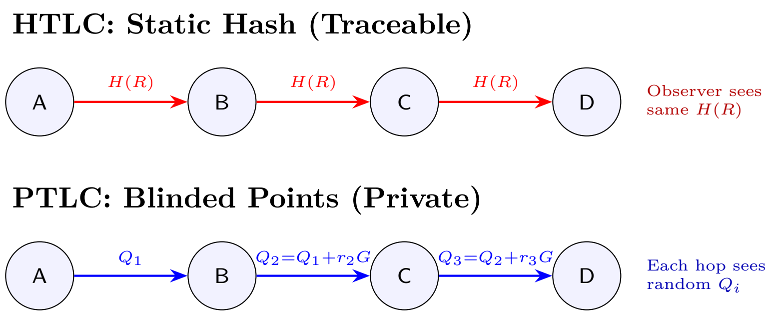

Figure 6: Multi-Hop Blinding Comparison. HTLC exposes the same hash $H(R)$ globally, while PTLC blinds the lock Q_i at each hop via homomorphic addition.

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.

4.7.1 Historical Evolution

HTLC (2016): Relies on SHA-256 preimages.

- **Defect**: The same hash H traverses the entire path, allowing correlation attacks (Wormhole Attack).

PTLC (2019+): Relies on Elliptic Curve Discrete Logarithm Problem (ECDLP).

- **Point Lock**: $Q = s \cdot G$.
- **Advantage**: Uses additive homomorphism ($Q_{\text{total}} = \sum Q_i$) to blind payment paths.

4.7.2 Technical Principle Comparison

Figure 6 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.

4.7.3 Core Properties Comparison

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**

4.7.4 Formal Security Analysis

Theorem 4.18 (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 4.19 (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:*

$$\mathit{Claim}(c_n) \implies \mathit{Claim}(c_1)$$

Theorem 4.20 (Timeout Refund Safety). *If no claim occurs before CSV timeout, funds are recoverable:*

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

4.7.5 Implementation Considerations

```rust
/// Verify s * G + P_beneficiary == Q
fn verify_ptlc_claim(
    point_lock: &Point,       // Q
    scalar: &Scalar,          // s
    beneficiary: &Point,      // P_ben
) -> bool {
    let computed = scalar * &GENERATOR + beneficiary;
    computed == *point_lock
}
```

Listing 1: PTLC Claim Verification Logic

4.8 TLA+ Specification Fragment

We formalize the channel state machine using TLA+ to verify monotonicity and termination.

```
---------------- MODULE EltooChannel ----------------
VARIABLES state, seq_num, phase
Phases == {"init", "active", "settling", "closed"}

Update == /\ phase = "active"
          /\ seq_num' > seq_num   (* Critical: Monotonicity *)
          /\ UNCHANGED phase

Settle == /\ phase = "active"
          /\ phase' = "settling"
          /\ UNCHANGED seq_num

Challenge == /\ phase = "settling"
             /\ seq_num' > seq_num   (* Higher state wins *)
             /\ UNCHANGED phase

(* Invariants to Check *)
Monotonicity == [][seq_num' >= seq_num]_seq_num
EventualTermination == <>(phase = "closed")
=====================================================
```

Listing 2: TLA+ Specification of Channel Monotonicity

4.9 Cost and Parameter Analysis

4.9.1 Cost Model

Total user cost is defined as:

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

Since $C_{\text{update}} \approx 0$ (off-chain), the amortized cost per transaction approaches zero as $N \to \infty$.

4.9.2 GhostDAG Confirmation Parameters

Confirmation time depends on DAG width k:

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

For $k = 16$, 10^{-6} security is reached in ≈ 3 seconds.

4.9.3 Ref-UTXO Security Depth

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

Table 14: Security Confirmation Time

System	**Time to Finality**
Bitcoin (6 blocks)	$\sim$ 60 minutes
Eltoo 2.0 (10 DAA)	$\sim$ **3 seconds**

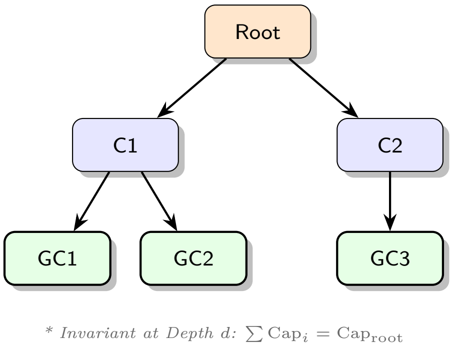

Figure 7: Recursive Channel Factory Structure. The architecture supports arbitrary nesting depths while preserving total liquidity conservation.

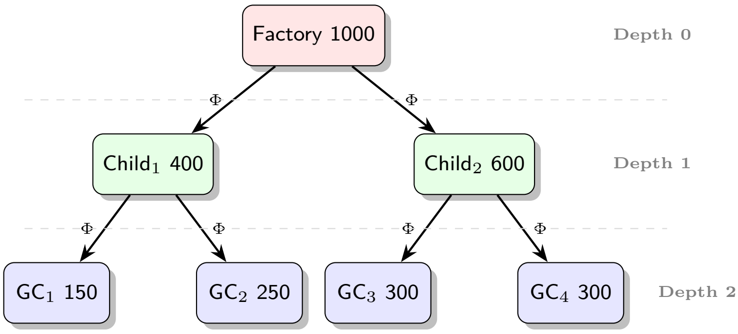

Figure 8: Fractal Channel Tree. Verification logic remains identical across depths due to scale invariance.

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.*

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 $\Phi : \mathcal{C} \to \{\mathcal{C}_1, \dots, \mathcal{C}_k\}$. As recursion depth $d \to \infty$, the system exhibits* ***scale invariance****:*

$$V(C_{d=0}) \equiv V(C_{d=n})$$

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

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

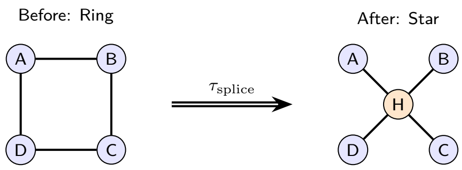

Figure 9: Atomic Topology Reconfiguration.

Algorithm 1 Deterministic Sub-channel ID Derivation

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

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.*

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)**.**

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

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:

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

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.

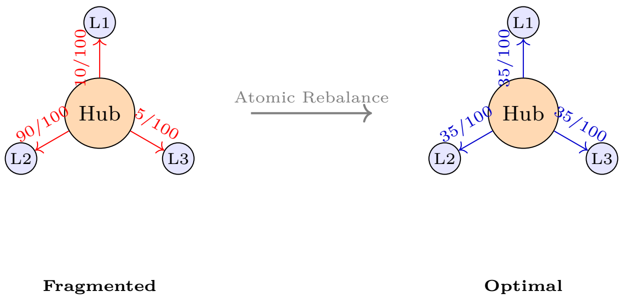

Figure 10: Atomic Rebalance. Minimizing liquidity fragmentation via $\tau_{\text{rebalance}}$.

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)**.**

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

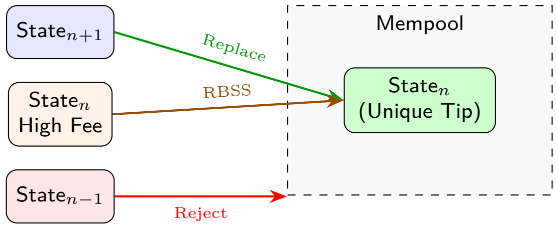

Figure 11: STPC (Single-Tip-Per-Channel) Strategy. Limits mempool DoS exposure.

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.

6.3.1 Mempool Entropy Bound

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

$$S_{\max} \propto k \cdot \ln(N_{\text{channels}}) \tag{3}$$

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}:*

1. ***Rule I (Monotonic Replacement):*** *If $\mathsf{State}_{new} > \mathsf{State}_{tip}$, unconditionally replace τ_{tip}*

Table 16: Mempool Entropy and DoS Analysis

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

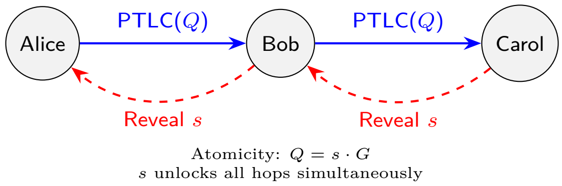

Figure 12: PTLC Multi-Hop Atomic Payment.

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 state sequence number.*

$$\text{Cost}_{\text{DoS}} = \sum_{i=1}^{k} \text{Cost}_{\text{tx}}(\tau_i) \propto \mathcal{O}(k)$$

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:*

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

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: $t_1 < t_2 < \cdots < t_1$. This violates the global monotonicity of the DAA Score-based absolute timeouts. Thus, the system is deadlock-free. □

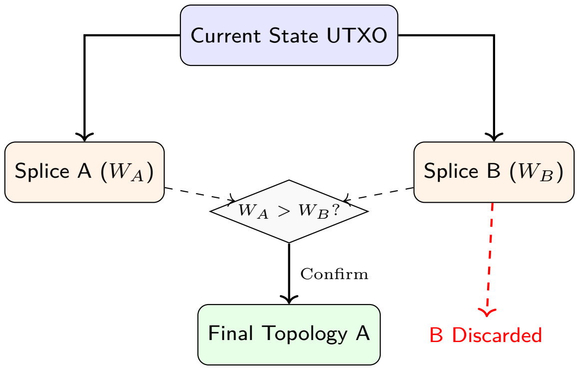

Figure 13: Conflict resolution for concurrent Splicing via GhostDAG weight.

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

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.*

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 17 summarizes the architectural improvements.

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

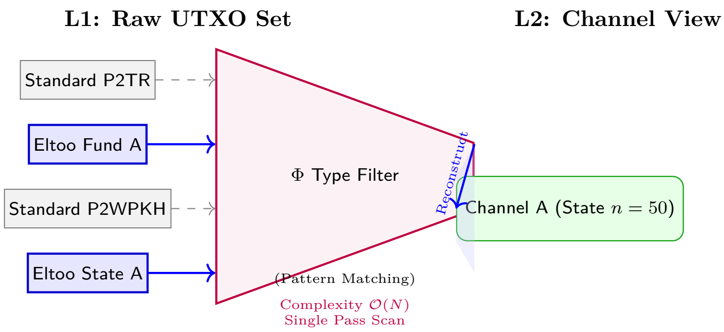

Figure 14: UTXO-to-State Projection. The discovery function Φ acts as a lens, filtering raw blockchain data into a logical channel view without external registries.

7 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 traffic; (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)**.**

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

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. □

7.3 Privacy Enhancement

7.3.1 Ephemeral Identity

Channel ID changes with every splice:

$$ID_C^{(i)} = H(\text{domain} \parallel \text{Ref\_OutPoint}_i \parallel \text{nonce})$$

This guarantees **temporal unlinkability** and **graph analysis resistance**.

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:*

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

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

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	$\mathcal{O}(N \log N)$	$\mathcal{O}(N)$ (Linear)
Identity	Static	**Ephemeral**
Censorship	Weak	Strong (UTXO-based)

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

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

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

```
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: s * G = R + c * P (Schnorr-like structure)
    for (i, scalar) in settle.adaptor_scalars.iter().enumerate() {
        let ptlc = &settle.ptlcs[i];
        if !verify_curve(scalar, keys[ptlc.idx], ptlc.point_q) {
            return false;
        }
    }
    true
}
```

Listing 3: Constant-Time PTLC Verification Logic

7.7 Case Study: Atomic Liquidation in DeFi

We analyze a liquidation scenario: Pool P must liquidate 100 users $\{U_1, \dots, U_{100}\}$ 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 $\text{Bal}'(U_i) = \text{Bal}(U_i) - \delta_i$.

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

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

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

8 Implementation Architecture

8.1 System Architecture Overview

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

8.2 Consensus Layer Implementation

8.2.1 Transaction Type Enumeration

The consensus layer uses algebraic data types to classify transactions:

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

Listing 4: Rust Enum for Transaction Types

8.2.2 Validation Pipeline

The `EltooBlockValidator` enforces monotonicity (Theorem 1) and reference existence (Axiom A2):

```
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

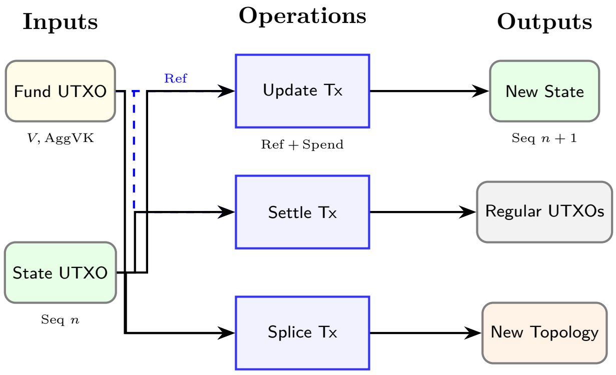

Figure 15: Transaction Topology Flow. Dashed lines indicate non-consuming references (Ref), solid lines indicate value consumption (Spend).

8.3 State Machine & UTXO Indexing

8.3.1 State Transition

The state machine handles local state evolution:

```rust
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:

```rust
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

```
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 Partially Signed Transaction Template (PSTT)

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

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 $T_{\text{dom}} \in \{T_{\text{FUND}}, T_{\text{UPDATE}}, \dots\}$.

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

```
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

8.6 Implementation Statistics

The core implementation is written in Rust, prioritizing correctness.

Table 22: Communication Complexity

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

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 24 summarizes the primary vectors and their corresponding architectural defenses.

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 is bounded by:*

$$P_{\text{rollback}} \leq P_{51\%} \times P_{\text{victim\_offline}}$$

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.

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**

9.4 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 ($\sum k^d$) to bloat the UTXO set.

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 and $\alpha \approx 0.1$ is the depth penalty. 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:
$$\text{ChannelID} = H(\text{fund\_outpoint} \parallel \text{nonce})$$
$$\sigma = \text{Sign}_{sk}(H(\text{domain} \parallel \text{ChannelID}) \parallel m)$$
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 state is bounded:*
$$E[\text{Confirm}] \leq \frac{1}{\lambda_{\text{block}}} \cdot (1 + \epsilon_{\text{jitter}})$$

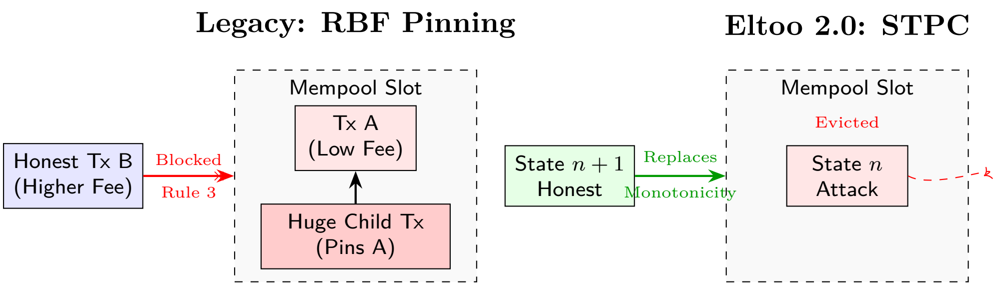

Figure 16: Pinning Attack Defense. Left: Legacy RBF rules allow attackers to pin transactions using heavy child descendants (Rule 3 blocking). Right: STPC enforces unconditional replacement based on state sequence ($n + 1 > n$), ignoring descendant weight.

Table 25: Griefing Cost Comparison

Metric	**Attacker Cost**	**Victim Cost**
Spam States	$\mathcal{O}(N) \times$ Fee	$\mathcal{O}(1)$ Verify
Force Close	$1 \times$ Fee	$1 \times$ Fee
Fund Lock	Capital Opportunity Cost	Capital Opportunity Cost
Time Cost	Days (Legacy)	**Seconds (Eltoo 2.0)**

9.8 Griefing Attack Cost Analysis

9.9 Security Summary

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

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**

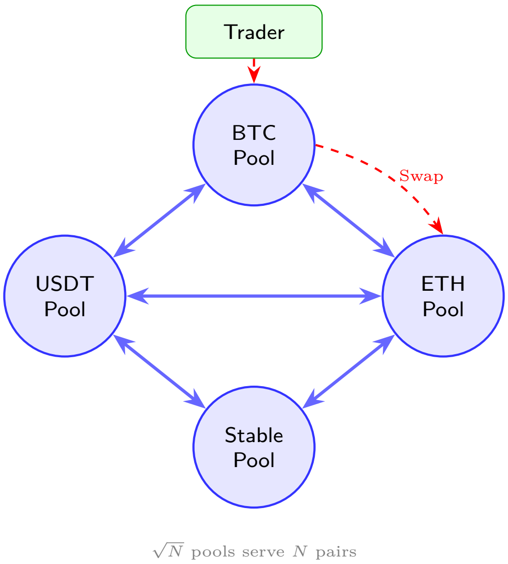

Figure 17: DeFi Liquidity Grid. AMM pools act as nodes in a channel network, enabling atomic cross-asset swaps via multi-hop routing.

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.

Advantages:

- **Capital Efficiency**: N pairs share $\sqrt{N}$ pools.
- **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:

Table 27: DEX Performance Comparison

Metric	Traditional	Channel DEX	Improvement
Latency	10–60 s	15 ms	1000×
Cost/Trade	$5–50	$0.001	10,000×
MEV Risk	High	**None**	Eliminated
Throughput	10 TPS	20k TPS	2000×

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

10.3 Decentralized Exchange (DEX)

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.

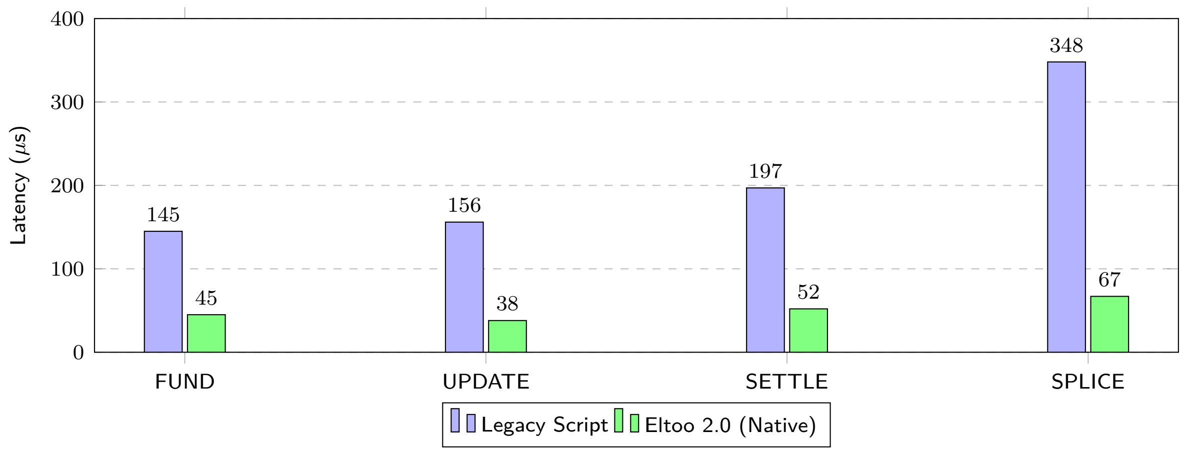

Figure 18: Validation Latency Comparison. Native type enumeration achieves $\approx$ 3–5$\times$ speedup by eliminating Script VM overhead.

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×

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.

11.2 Transaction Validation Performance

11.2.1 Single Transaction Latency

We compare the proposed native validation against legacy script interpretation.

Analysis: Native validation complexity is $\mathcal{O}(1)$ (pattern matching), whereas script validation is $\mathcal{O}(\text{size}_{\text{script}})$.

11.2.2 Batch Verification

Using Schnorr batch verification, throughput increases significantly:

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

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 $\mathcal{O}(1)$.

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

Component	Legacy LN	Eltoo 2.0	Reduction
Fund UTXO	120 B	120 B	0%
Latest State	256 B	256 B	0%
History	$256 \times N$	**0**	100%
Revocation Keys	$32 \times N$	**0**	100%
Total	≈288 KB	**556 B**	**99.8%**

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

11.4 Network Discovery Performance

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:

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

```
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**: 99.8% reduction per channel.

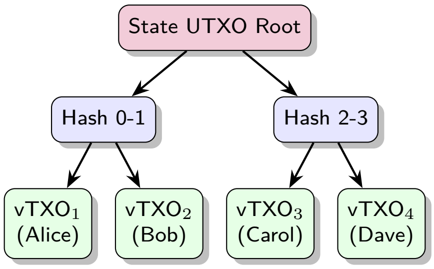

Figure 19: Merkleized vTXO Pool. Users hold "virtual UTXOs" inside the state commitment. Receiver offline capability is achieved by atomic Merkle leaf swaps.

3. **Settlement**: Sub-second latency via GhostDAG.
4. **Security**: DoS cost increased by 100×.

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)

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 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 CSP set $\mathcal{H}$, the anonymity set size is defined as the Cartesian product of channel sets:*

$$|\mathcal{AS}(p)| = \prod_{h \in \mathcal{H}} |\text{Channels}_h|$$

*Payment p is k-**anonymous** iff $|\mathcal{AS}(p)| \geq k$.*

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). $\square$

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:

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

Key Properties:

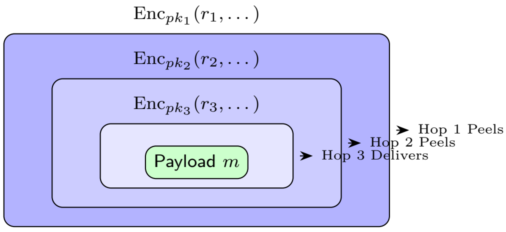

Figure 20: SPHINX-Lite Onion Structure. Each hop "peels" one layer of encryption, revealing only the next hop's routing info (r_i), ensuring forward secrecy.

Table 33: Privacy Mode Tradeoffs

Mode	**Latency**	**Anonymity Set**
Direct	∼100ms	1 (None)
Single CSP	∼200ms	10^3
Multi CSP	∼500ms	10^5
Tor + Multi	∼2.0s	10^6 (Max)

- **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)**.**

$$T_{\text{latency}} = T_{\text{base}} + \alpha \cdot \log |\mathcal{AS}|$$

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).

Table 34: CSP Fee Schedule Structure

Service	Fee Model	Economic Rationale
Channel Open	Fixed + 0.01%	Overhead allocation
Routing	0.1%	Marginal cost pricing
JIT Liquidity	5.0% APY	Capital rental premium
Swap	0.3–1.0%	Market risk premium
Mixing	0.1%	Anonymity premium

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.

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 ρ is the risk aversion coefficient ($\approx$ 0.5–2.0) and C_{opp} represents DeFi opportunity cost.

Theorem 13.3 (Competitive Equilibrium). *In a market with $N \geq 3$ CSPs and free entry:*

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

Proof. If $\text{Fee} > C_{\text{marg}} + \epsilon$, 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.

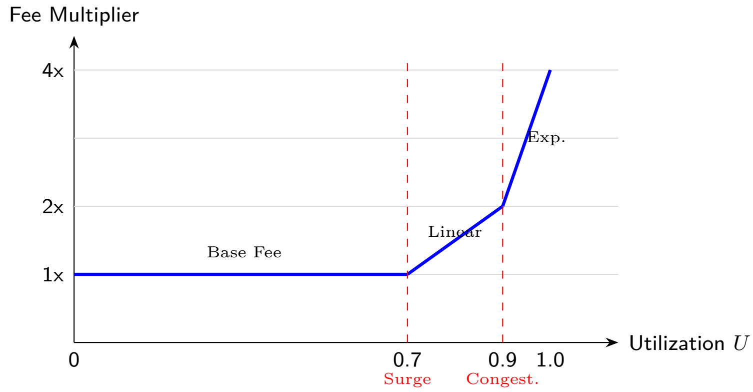

Figure 21: Congestion Pricing Curve. Fees remain flat until 70% utilization, then rise linearly, and finally exponentially to prevent resource exhaustion.

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.

13.4 Dynamic Fee Adjustment

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

```
pub fn compute_dynamic_fee(utilization: f64) -> Fee {
    let base_fee = 100; // sompi
    let multiplier = if utilization > 0.9 {
        2.0 + (utilization - 0.9) * 10.0  // Exponential
    } else if utilization > 0.7 {
        1.0 + (utilization - 0.7) * 2.5   // Linear
    } else {
        1.0  // Base
    };
    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.*

Proof. Let $S = \{\text{Honest}, \text{Delay}, \text{Steal}\}$. Since PTLCs cryptographically prevent theft ($P(\text{Success}|\text{Steal}) = 0$) 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 $\to C_{\text{marg}}$.
2. **User Sovereignty**: Guaranteed by L1 fallback.

3. **Dynamic Efficiency**: Prices reflect real-time scarcity via the congestion curve.

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 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 6.1) and *Deadlock Freedom* (Theorem 6.5).

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; 100–600× faster settlement.
- **Efficiency**: 99.8% reduction in storage overhead.
- **Scale**: Support for billions of off-chain TPS with minimal L1 footprint.

14.2 Paradigm Shifts

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

14.3 Limitations and Mitigation Strategies

We analyze the trade-offs inherent in this architecture and proposed mitigations in Table 37.

14.4 Future Research Directions

14.4.1 Short-Term Extensions

- **Multi-Party Channels**: Combining MuSig2 with BFT protocols for N-party consensus.

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 (Sec 7.3) & Stealth Addresses.

- **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 and billions of TPS proves that UTXO-based Layer 2 solutions can rival centralized 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 (G, g, n), private key $x \in \mathbb{Z}_n$, public key $P = x \cdot g$, the signing process for message m is:*

1. *Choose random number k, compute $R = k \cdot g$*

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

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

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:*

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

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:*

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

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

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

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:*

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 through defining "blue sets" between blocks:

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

where $Blue(b)$ is the blue score of block b, computed by a recursive algorithm.

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.