# 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 $\text{Ref} : \mathcal{U} \to \mathcal{U}^{readonly}$, 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 $O(1)$ 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 $O(1)$, 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?
2. **Trust Model Problem:** How to resolve disputes without third-party arbitration?

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.

**Preliminary Concepts**

**Ledger Model and Transaction Structure:**

- **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.
- **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.

**Payment Channel Fundamentals:**

- **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.
- **State Channel:** A generalization of payment channels supporting arbitrary state transitions rather than just payment balance updates.
- **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.
- **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.

**Conditional Payment Primitives:**

- **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.

- **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.

**Cryptographic Primitives:**

- **Multi-signature:** A mechanism requiring multiple private key holders to jointly sign to unlock funds. Traditional multi-sig (e.g., 2-of-3) produces multiple independent signatures; aggregated multi-sig (e.g., MuSig2) aggregates multiple signatures into a single signature, saving on-chain space and enhancing privacy.
- **Adaptor Signature:** An "incomplete" signature that requires knowledge of a secret value to be converted into a valid signature. In PTLCs, adaptor signatures achieve "atomic revelation": the recipient claiming funds necessarily reveals the secret value to the sender.
- **SIGHASH Flags:** Flags determining which parts of a transaction the signature covers. `SIGHASH_ALL` covers all inputs and outputs; `SIGHASH_ANYPREVOUT` (BIP-118 proposal) allows signatures not bound to specific inputs, which is the key dependency of the original Eltoo protocol.

## 1.2 Design Principles

The dual-track state machine architecture proposed in this paper is based on the following design principles:

**Principle 1: 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 2: 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 $O(\text{script\_size})$ to $O(1)$
- Eliminates uncertainty introduced by script interpreters

**Principle 3: 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:

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

$$\text{This Architecture: State\_UTXO} \xrightarrow{\tau_{\text{settle}}} \text{Value\_Distribution} \quad \text{(deterministic execution)}$$

## 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.

| 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 |

Table 1: Trust Model Comparison

# 2 Related Work and Technical Background

This chapter first introduces the technical evolution of payment channel protocols, then analyzes structural defects of existing solutions, providing a theoretical foundation for the architectural design in subsequent chapters.

## 2.1 Preliminaries

To facilitate understanding of subsequent content, this section systematically introduces core concepts and formal definitions involved in this paper.

### 2.1.1 Cryptographic Foundations

**Definition 2.1** (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 2.2** (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 $R = k \cdot g$*
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 2.3** (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 2.4** (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 $\sigma$:*

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

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

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

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

**Definition 2.5** (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.*

### 2.1.2 Timelock Mechanisms

**Definition 2.6** (Timelock). *Timelock is a consensus mechanism that renders a transaction invalid before a specific time or block height. This paper involves two types of timelocks:*

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

**Definition 2.7** (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.*

### 2.1.3 Directed Acyclic Graph Consensus

**Definition 2.8** (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.*

### 2.1.4 Finite State Machine Foundations

**Definition 2.9** (Finite State Machine). *A finite state machine (FSM) is a five-tuple* $M = (Q, \Sigma, \delta, q_0, F)$*:*

- $Q$*: Finite set of states*
- $\Sigma$*: 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 2.10** (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.*

### 2.1.5 Covenants and Script Extensions

**Definition 2.11** (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* $\tau$ *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)*

*The controversy around recursive covenants lies in their potential to harm Bitcoin's fungibility—see §1.2.*

**Definition 2.12** (SIGHASH Flags). *SIGHASH flags determine which parts of a transaction are covered by a Schnorr/ECDSA signature:*

| 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 |

*`SIGHASH_ANYPREVOUT` (BIP-118 proposal) is the key dependency of the original Eltoo protocol, allowing signatures to not bind to specific UTXOs, thereby enabling flexible state replacement.*

### 2.1.6 Notation Conventions

This paper uses the following notation conventions:

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

## 2.2 Protocol Evolution: From Penalty Mechanism to State Replacement

### 2.2.1 Lightning Network's Penalty Mechanism and Limitations

The Lightning Network proposed by Poon and Dryja (2016) adopts a **penalty mechanism** to resolve the state rollback problem. Its working principle is as follows:

**Mechanism Description**: When channel state updates from $S_n$ to $S_{n+1}$, both parties exchange the "revocation key" for $S_n$. If either party attempts to broadcast the old state $S_n$, the counterparty can use the revocation key to construct a penalty transaction, confiscating all funds of the cheating party.

**Formal Expression**: Let $\mathcal{R}_n$ be the set of revocation keys for state $n$, the security of the penalty mechanism relies on:

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

**Structural Defects**: This mechanism introduces the **Toxic Waste Problem**:

1. Nodes must permanently store all historical revocation keys $\{\mathcal{R}_0, \mathcal{R}_1, ..., \mathcal{R}_{n-1}\}$
2. Storage complexity is $O(n)$, where $n$ is the number of state updates
3. Any data loss or backup recovery error could cause nodes to accidentally broadcast old states, triggering the penalty mechanism

### 2.2.2 Eltoo Protocol and State Replacement Mechanism

Decker, Russell, and Osuntokun (2018) proposed the Eltoo protocol, using **state replacement** instead of the penalty mechanism.

**Core Idea**: Allow update transaction $\tau_{n+1}$ to directly spend any previous update transaction $\tau_i$ $(i \leq n)$, rather than having to spend predecessor transaction $\tau_n$. This means old states can be "skipped" without storing revocation keys.

**Technical Dependency**: The original solution depends on the `SIGHASH_NOINPUT` signature hash flag (later evolved into BIP-118 `SIGHASH_ANYPREVOUT`), whose semantics are: the signature does not bind to the specific input UTXO identifier (OutPoint), only to the output script and amount.

**Definition (ANYPREVOUT Signature)**: Given transaction $\tau$ and input index $i$, the traditional signature hash is computed as:

$$h_{traditional} = H(\tau.\text{inputs}[i].\text{outpoint} \| \tau.\text{outputs} \| ...)$$

The ANYPREVOUT signature hash omits the input identifier:

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

**Replay Attack Risk**: This design introduces the security concern of **replay attacks**. Consider the following attack scenario:

Suppose user $U$ uses private key $sk$ to control two UTXOs $A$ and $B$, both with identical locking scripts. If $U$ generates an ANYPREVOUT signature $\sigma$ for $A$:

$$\sigma = \text{Sign}_{sk}(H(\text{Output}_A \| \text{Script}))$$

Since $\sigma$ does not include $A$'s unique identifier, an attacker can replay $\sigma$ on $B$, constructing a valid spending transaction, resulting in unintended fund transfers.

### 2.2.3 Engineering Compromise of BIP-118

To mitigate replay risk, BIP-118 introduces **Public Key Tagging**, mandating the use of specifically tagged public keys for this signature type. This essentially shifts protocol-layer security responsibility upward to application-layer key management, violating the orthogonality principle of system design, and does not eliminate risks from key reuse.

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

This means if a user uses the same private key $sk$ but derives $P_{std}$ through the standard path, that public key is **physically** prohibited from using APO signatures. Only when the user explicitly understands the intent and derives $P_{apo}$ does the protocol allow "input decoupling" behavior.

## 2.3 The Recursive Covenant Dilemma

Beyond security vulnerabilities, the APO solution has triggered deeper economic-philosophical debates in the Bitcoin community regarding **covenant boundaries**.

**Risk Argument**: APO essentially introduces introspective capabilities. Researchers such as Shinobi have pointed out that if scripts are allowed unlimited constraints on "spending transaction outputs," APO may trigger security and fungibility issues related to **recursive covenants**.

Formally, if script $S$ can force its spending transaction's output to be locked in the same script $S$:

$$S \xrightarrow{\text{spend}} S' \quad \text{where } S' \equiv S$$

This leads to damage to Bitcoin's fungibility. For example, regulatory authorities could use this feature to create "whitelisted coins," forcing funds to flow only within specific compliant address sets, thereby implementing censorship at Layer 1 through technical means.

This concern about "toxic recursion" has prevented BIP-118 from being activated since 2018, in the absence of a global security model consensus (such as CTV, BIP-119).

## 2.4 Structural Defect Analysis of Existing Architectures

Existing scaling solutions based on script extensions attempt to simulate state machines by stacking opcodes. From a software engineering perspective, this approach violates the **Principle of Orthogonality**—different components of the system should change independently without affecting each other.

**Design Proposition**: Separate **value transfer** and **state transition** into two orthogonal concerns:

- **Value Transfer**: Handled by the UTXO model, preventing double-spending, ensuring fund security

- **State Transition**: Handled by consensus-layer embedded finite state machines, managing logical evolution

The essential problem with the BIP-118 solution is attempting to implement state transition logic at the value transfer layer (Script), causing public keys to be forced to carry permission semantics. This paper's dual-track architecture achieves physical isolation of two concerns by introducing **native transaction types** at the consensus layer.

| Defect Type | Description | Impact |
|---|---|---|
| Verification Complexity | State validation logic executed via bytecode, $O(\text{script\_size})$ | Nodes cannot predict resource consumption |
| Semantic Opacity | Consensus layer cannot distinguish channel update from regular transfer | Layer 2 logic cannot obtain Layer 1 optimization |
| Security Boundary Blur | Relies on public key tagging to prevent replay attacks | Security responsibility delegated to application layer |
| Concern Coupling | Value locking and state evolution logic entangled | Violates single responsibility principle |

Table 2: Structural Defects of Script-Based Solutions

## 2.5 This Paper's Solution: UTXO-Native Semantics

Addressing the above problems, this paper proposes a **UTXO-native** solution. By pushing Eltoo semantics down to the transaction structure itself, this approach resolves scalability issues while avoiding BIP-118's security risks.

**Core Design**: Introduce native transaction type set $\mathcal{T}_{Eltoo} = \{\tau_{fund}, \tau_{update}, \tau_{settle}, \tau_{splice}\}$ and reference-type operator Ref.

**Property 1: Type System Prevents Replay Attacks**

Unlike ANYPREVOUT's reliance on signature scope delimitation, this solution prevents replay through **strict type binding**:

$$\forall \tau \in \mathcal{T}_{Eltoo} : \text{Input}(\tau) \text{ MUST be of type } \texttt{ELTOO\_STATE}$$

Ordinary UTXOs are physically excluded from channel state inputs at the type level, isolating replay paths.

**Property 2: Finite State Machine Avoids Recursion Risks**

This solution implements channel updates through consensus-layer defined **finite state transition rules**, rather than supporting arbitrary recursive scripts:

$$\text{Transition} : S_n \xrightarrow{\tau_{update}} S_{n+1} \quad \text{where } n' > n \text{ (strict monotonicity)}$$

This design mathematically precludes the possibility of constructing arbitrary recursive covenants.

**Property 3: Explicit Reference Mechanism**

Define read-only reference operator $\text{Ref} : \mathcal{U} \to \mathcal{U}^{readonly}$, enabling state update transactions to access fund anchor metadata without consuming that UTXO. This reduces verification complexity from $O(\text{script\_size})$ to $O(1)$.

## 2.6 DAG Consensus and Protocol Compatibility

This paper's architecture adopts the GhostDAG consensus protocol, whose characteristics align well with the state replacement mechanism. Let GhostDAG parameters be $(D, k)$, where $D$ is the network delay constraint and $k$ is the blue set parameter.

**Property 1 (DAA Score Temporal Consistency)**: GhostDAG's Difficulty Adjustment Algorithm Score provides a globally consistent logical clock:

$$\forall b_1, b_2 \in DAG : b_1 \prec_{topo} b_2 \Rightarrow \text{DAA}(b_1) < \text{DAA}(b_2)$$

This property guarantees determinism and predictability of relative timelocks (CSV), unaffected by timestamp manipulation.

**Property 2 (Fast Confirmation)**: GhostDAG's parallel block generation mechanism makes confirmation time satisfy:

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

Fast confirmation reduces the economic risk window of channel disputes, allowing shorter CSV periods.

**Property 3 (High Throughput)**: GhostDAG's throughput compared to single chains satisfies:

$$\text{TPS}_{GhostDAG} = O(k \cdot \text{TPS}_{single\_chain})$$

This enables complex topological operations (such as multi-party Splice) to execute efficiently on-chain.

**Corollary 2.13** (Protocol Completeness)**.** *This paper's architecture constitutes a closed protocol system, with all semantics fully implemented at the consensus layer, requiring no dependency on external soft forks or script extensions.*

## 2.7 Formal Comparison of State Revocation Mechanisms

This section provides a formal comparative analysis of state revocation mechanisms between the original Eltoo protocol and this paper's consensus-layer native implementation (hereinafter "Eltoo 2.0").

### 2.7.1 Original Eltoo's State Overwrite Mechanism

Original Eltoo adopts **state overwrite** rather than penalty mode, using `SIGHASH_NOINPUT` (or `ANYPREVOUT`) and `nLocktime` state numbering to allow update transactions to bind to previous update outputs with higher state numbers, invalidating old settlement transactions.

However, this scheme has inherent limitations:

- Depends on a non-existent soft fork (`SIGHASH_NOINPUT`)
- Uses script locktime and `nLocktime` workarounds to encode state transitions
- Intermediate states are not fully consistent across participants, hindering multi-party channel extension

### 2.7.2 Key Improvements in This Architecture

**Improvement 1: Consensus-Layer Native State Revocation**

This architecture designs state revocation as a first-class citizen at the consensus layer, implementing "only the latest state can settle" constraint verification directly in consensus rules through the **Fund UTXO (static anchor) + State UTXO (dynamic state) dual-track model**, without relying on script-layer tricks or undeployed soft forks.

**Improvement 2: Symmetric Participant State Revocation with Native Multi-Party Extension**

While original Eltoo theoretically achieves symmetry, its script-level implementation still causes intermediate state inconsistency across participants. This architecture ensures formal consistency of each sub-channel's revocation logic through the **sub-channel independence axiom** and **normalized State UTXO representation**.

**Improvement 3: State Management Without Preimage Storage Burden**

Original Eltoo simplified data management compared to penalty mechanisms—no longer needing to save hash preimages of revocation keys for each invalid state. This architecture further strengthens this property: consensus rules fundamentally guarantee that all outdated states and their corresponding HTLC/PTLC entries cannot be confirmed on-chain, as only the **current State UTXO** is eligible for consumption by Settle transactions.

**Improvement 4: State Determinism via Virtual Reference Mechanism**

This architecture introduces **Fund UTXO Virtual Reference**, making channel identifiers completely deterministically derived from `funding_outpoint`, without relying on script execution results or multi-round interactive protocols.

**Improvement 5: Atomic Composition Guarantee of State Revocation and PTLC**

This architecture tightly couples state revocation with **native PTLC**, making conditional payment validity equally subject to the constraint that "only PTLC entries in the latest State UTXO can be triggered."

**Improvement 6: Consensus-Level DoS Protection and State Size Bounds**

In original Eltoo design, outdated state chains may accumulate in the mempool, and mempool behavior is difficult to formally predict, constituting a potential denial-of-service attack vector. This architecture fundamentally ensures that state revocation operations do not cause unbounded growth in on-chain resource consumption through **STPC (Single-Tip-Per-Channel) policy** and **state size hard limits** (`MAX_STATE_INCREMENT`, `MAX_PTLC_ENTRIES`, etc.).

| Feature Dimension | Original Eltoo | This Architecture (Eltoo 2.0) |
|---|---|---|
| Revocation Primitive | Script-level overwrite + `SIGHASH_NOINPUT` | Consensus-level tx enum + dual-track UTXO |
| Multi-Party Scalability | Theoretically feasible, script complexity increases | Native sub-channel support + recursive factory ops |
| State Data Management | Simplified but script parsing overhead remains | Store latest state only, outdated states unconfirmable |
| State Determinism | Depends on script execution & signature parsing | Fund UTXO virtual reference provides unforgeable anchor |
| Conditional Payment | Combined with HTLC scripts | Native PTLC with atomic protection |
| DoS Protection | Discussed but lacks hard constraints | STPC policy + state size hard limits |

Table 3: State Revocation Mechanism Comparison

## 2.8 Axiom System

To provide formal foundations for subsequent theorem proofs, this section explicitly lists the core axioms of this paper's architecture. These axioms form the metatheoretical basis of the system; all safety and liveness theorems can be rigorously deduced from these axioms.

## 2.9 BIP-118 Security Boundary Analysis

The following diagram illustrates how security responsibility escapes from the protocol layer to the user behavior layer in the BIP-118 solution:

**Core Issue**: Even when consensus layer verification succeeds, implementation errors can still lead to fund loss, violating the engineering principle of "pushing complexity down to the protocol layer."

| Axiom | Formal Expression | | Semantic Interpretation |
|---|---|---|---|
| **A1** | $\mathcal{S}_{channel} \cong \mathcal{U}_{chain}$ | | Channel state is isomorphic to on-chain UTXO set; no separate registry |
| **A2** | $\forall \tau_{update} : n' > n$ | | State sequence numbers must be strictly increasing (Theorem 1) |
| **A3** | $\text{Ref}(U) \in \tau \Rightarrow U \in \mathcal{U}_{post}$ | | Read-only reference operations do not consume UTXOs |
| **A4** | $\sum V_{in} = \sum V_{out} + \delta_{fee}$ | | Input total equals output total plus fees (Theorem 4) |
| **A5** | $\text{Security}(C_{child})$<br>$\text{Activity}(C_{parent})$ | $\perp$ | Sub-channel security is orthogonal to parent liveness (Theorem 6) |
| **A6** | $\Sigma_{Type_A} \cap \Sigma_{Type_B} = \emptyset$ | | Signature spaces of different tx types are strictly orthogonal |

Table 4: Core Axiom System

### 2.9.1 Formal Verification Logic: Security Boundary Fragility

Despite the mathematical guarantee that $\mathcal{K}_{std} \cap \mathcal{K}_{apo} = \emptyset$, system-level security depends on **external behavioral consistency**. We establish the following formal attack model:

Assume wallet $W$ has seed $S$, with derivation function $\text{Derive}(S, \text{path})$. The safety property requires:

$$\forall P = \text{Derive}(S, \text{path}), \quad (\text{IntendedUsage}(P) \neq \texttt{APO}) \implies (P \notin \mathcal{K}_{apo})$$

However, the BIP-118 scheme introduces **state dependency** risks:

1. **Key Reuse Disaster**: If legacy or malicious software ignores derivation path semantics and uses $sk$ to generate both $P_{std}$ (for cold storage) and $P_{apo}$ (for Lightning), although the public key bytes differ, the private key is identical

2. **Non-determinism in Backup Recovery**: When recovering a wallet, users must know exactly which UTXOs belong to $\mathcal{K}_{apo}$. If a user imports their mnemonic into a wallet that doesn't support BIP-118, funds may be invisible or incorrectly spent

### 2.9.2 Historical Discussion and Academic Critique

BIP-118's "public key tagging" design triggered intense debate, as it violates Bitcoin script's **statelessness** and **address abstraction** principles.

**1. Anthony Towns' Compromise (2019)**: When restructuring `NOINPUT` to `ANYPREVOUT`, Towns explicitly acknowledged this as a tradeoff. He noted that simple `NOINPUT` is a "footgun," requiring changes to public key structure to force users to "prove intent."

**2. ZmnSCPxj's Engineering Warning**: ZmnSCPxj repeatedly emphasized that any mechanism allowing replay, if dependent on user behavior (such as not reusing addresses), will ultimately lead to fund loss. BIP-118 adds a lock, but this lock requires all downstream infrastructure (exchanges, hardware wallets, block explorers) to correctly identify and handle these "special public keys."

**3. Architectural Critique**: From an architectural perspective, BIP-118 makes **public keys no longer pure**. In classical cryptography, public keys only represent identity; in BIP-118, public keys are forced to carry **permission semantics**. This **tight coupling** leads to system brittleness:

- **Loss of Unknowability**: Observers cannot judge public key purpose solely by mathematical properties; they must parse metadata (tags)

- **Forward Compatibility Breakage**: Future script upgrades requiring new signature modes may need to define new key tags, leading to infinite fragmentation of public key formats

BIP-118's "public key tagging" fix is a **local optimum** within the existing Script architecture, not an architecturally correct solution. It trades protocol-layer security for increased application-layer entropy (complexity), directly violating the system engineering principle of "push complexity down to the protocol layer, leave simplicity for the application layer."

## 2.10 Economic Efficiency Boundary of Payment Systems

In distributed ledger scaling solution design, engineering tradeoffs are often simplified as a binary opposition of "security-scalability." This section proposes a more refined **three-dimensional economic efficiency space** $\Omega$ for quantitative analysis of different Layer 2 protocols:

$$\Omega = \mathcal{L}_{atency} \times \mathcal{T}_{hroughput} \times \mathcal{C}_{apital}$$

| 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) | Watchtower |
| Native Eltoo | **Sub-second** | **High (pooled)** | **Consensus-native** |

Native Eltoo combines low latency with high capital efficiency through channel factories and consensus-layer semantics.

Figure 1: Layer 2 Protocol Economic Positioning: Comparison of key metrics across scaling solutions.

The marginal cost function $MC(v)$ of a payment system is defined as:

$$MC(v) = \alpha \cdot C_{on-chain} + \beta \cdot C_{routing} + \gamma \cdot C_{time\_value}$$

For high-frequency, large-value financial flows (such as exchange clearing, market maker hedging), $\gamma$ (time value of capital) becomes the dominant factor. This architecture achieves superior economics in this market segment through **Channel Factories** aggregating $N$ participants' funds into a single UTXO.

# 3 Research Contributions

Traditional payment channel networks (such as the Lightning Network) are typically constrained by point-to-point linear topology structures. Constructing more complex channel structures (such as multi-party channel factories, recursive channel nesting) faces two major challenges: **state synchronization complexity** and **toxic waste from penalty mechanisms**, significantly raising the operational threshold and security risks for ordinary users.

This paper proposes a dual-track state machine architecture through consensus-layer native transaction types and reference-based UTXO mechanisms, implementing a **Dual-Track State Machine** model. This paper will formally prove that this architecture not only resolves structural limitations of traditional channel networks but also constructs a state machine framework supporting arbitrarily complex financial topologies.

## 3.1 Main Contributions

The main contributions of this paper include:

1. **Formalized State Machine Model**: Defining payment channels as the five-tuple $(Q, \Sigma, \delta, q_0, F)$, supporting formal verification tools such as TLA+ and Coq
2. **Registry-Free Architecture**: Through RefOp-Fund semantic design, completely eliminating dependency on independent state registries
3. **Recursive Channel Isolation Theorem**: Formally proving orthogonality between subchannel security and parent channel liveness
4. **Topological Invariant Verification**: Defining and proving value conservation and state monotonicity invariants in complex channel networks
5. **Constant-Time PTLC Verification**: Achieving $O(1)$ conditional payment verification by directly deriving participant public keys from Fund UTXO
6. **Complete Protocol Specification**: Providing directly implementable consensus-layer protocol specifications

## 3.2 Information-Theoretic Analysis of State Determinism

Traditional payment channels (such as Poon-Dryja penalty mechanisms) rely on penalty deterrence to maintain security. From an information-theoretic perspective, verifying the validity of current state $S_t$ requires not only the information entropy of $S_t$ itself but also the revocation key information of all historical abandoned states $\{S_0, \ldots, S_{t-1}\}$.

**Definition 3.1** (State Entropy)**.** *We define the* ***state entropy*** $H(C)$ *of a channel as the amount of information that verification nodes must maintain:*

$$H_{LN}(t) \propto \sum_{i=0}^{t-1} size(RevocationKey_i) \approx O(t)$$

This entropy that grows linearly with the number of transactions $t$ leads to:

- **Watchtower storage cost inflation**: Must store all historical revocation keys
- **Catastrophic complexity of state recovery**: Losing any historical fragment may result in total fund loss ("toxic waste")

This architecture introduces a **low-entropy state machine model**. Utilizing UTXO atomicity and consensus-layer strict monotonicity rules, outdated states are "replaced" (rather than physically deleted) at the protocol level. Its state entropy collapses to constant level:

$$H_{Eltoo2.0}(t) \approx \text{size}(\text{State}_{current}) + \text{size}(\text{FundAnchor}) \approx O(1)$$

| Protocol Model | State Entropy | Encoding Paradigm | Security Info Source |
|---|---|---|---|
| Lightning (Penalty) | $O(t)$ linear | Error Detection | Full history comparison |
| **This Architecture** | $O(1)$ constant | Forward Error Correction | Latest state only |

Table 5: Information-Theoretic Comparison

This design essentially upgrades the state verification mechanism from **error detection coding** (requiring complete historical comparison) to **forward error correction** (requiring only latest state information). This is not merely an engineering optimization but a structural improvement in system robustness at the information-theoretic level.

### 3.2.1 Verification Causality Graph Comparison

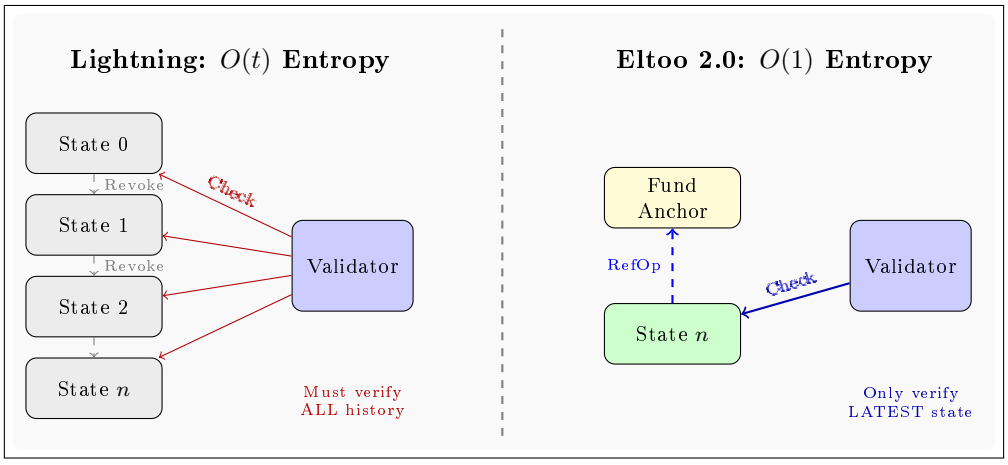


Figure 2: Verification Causality Graph: Lightning vs Eltoo 2.0

**Information-Theoretic Interpretation**:

| Protocol Model | State Entropy | Encoding Paradigm | Security Info Source |
|---|---|---|---|
| Lightning (Penalty) | $O(t)$ linear | Error Detection | Full history comparison |
| **This Architecture** | $O(1)$ constant | Forward Error Correction | Latest state only |

Table 6: Information-Theoretic Comparison (Updated)

**Theorem 3.2** (Information-Theoretic Robustness)**.** *For any payment channel protocol* $\Pi$*, its fault tolerance for state recovery* $\mathcal{R}$ *and state entropy* $H$ *satisfy an inverse relationship:*

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

**Corollary 3.3.** *Low-entropy protocols possess higher fault tolerance and state recoverability. Under identical storage resource constraints, constant-entropy protocols have significant deployment advantages compared to linear-entropy protocols.*

### 3.3 Architectural Advantages

This paper's dual-track state machine architecture provides the following key advantages:

1. **Orthogonal Separation**:
   - Fund UTXO (static anchor) - stable lifecycle, low-frequency updates
   - State UTXO (dynamic pointer) - high-frequency evolution, independent state
2. **Type Safety**:
   - Transaction type determined by I/O topology structure
   - Compile-time guarantee through algebraic data types
   - Eliminates script interpretation uncertainty
3. **Constant Complexity**:
   - Verification complexity: $O(1)$ vs Script-based $O(\text{script\_size})$
   - Storage complexity: $O(1)$ latest state vs $O(n)$ full history
   - PTLC verification: Direct derivation, no cross-structure queries
4. **Topological Freedom**:
   - Atomic Splicing supports arbitrary topology reconfiguration
   - Recursive channel factories enable fractal structure
   - Sub-channel isolation guarantees security independence

### 3.4 Comparison with Existing Solutions

| **Dimension** | **Lightning (Penalty)** | **BIP-118 Eltoo** | **This Architecture** |
|---|---|---|---|
| Consensus Dependency | No soft fork | Requires Bitcoin soft fork | Native support |
| State Representation | Script + HTLC | Script encoding | Native UTXO types |
| Value/State Separation | Coupled | Coupled | Orthogonal (dual-track) |
| Cross-State Reference | None | Implicit via signature hash | RefOp-UTXO primitive |
| Type Safety | Runtime | Runtime | Compile-time |
| Verification Complexity | $O(\text{script})$ | $O(\text{script})$ | $O(1)$ |
| State Storage/Update | $O(n)$ history | $O(1)$ latest | $O(1)$ latest |
| Multi-Party Rounds | $O(m^2)$ | $O(m^2)$ | $O(m)$ (PSTT) |
| Settlement Time | Minutes | Minutes | Sub-second |
| Backup Complexity | Full history | Latest state | Latest state |

Table 7: Comprehensive Architecture Comparison ($n$ = updates, $m$ = participants)

| Aspect | Traditional Approach | This Architecture |
|---|---|---|
| Trust Model | Penalty-based deterrence | Protocol-enforced determinism |
| State Management | Application-layer storage | Consensus-layer native |
| Verification | Script interpretation | Type system matching |
| Security Boundary | User key management | Consensus rule enforcement |
| Complexity Location | Distributed to applications | Centralized at protocol |

Table 8: Paradigm Shift in Design Philosophy

### 3.5 Theoretical Significance

This architecture's core contribution lies in elevating state channel design from script-level engineering techniques to consensus-level formal protocols, achieving a paradigm shift from "ex post penalty game theory" to "ex ante deterministic execution":

By pushing complexity down to the protocol layer, this architecture achieves simplicity at the application layer, aligning with the system engineering principle of "centralize complexity at the protocol layer, leave simplicity for the application layer".

# 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 $O(1)$ time complexity pattern matching verification. Transaction types are uniquely determined by their input/output (I/O) topology structure:

| 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` | $\{\text{Spend}(U_{fund}), \text{Spend}(U_{state}^{(n)})\}$ | $\{U'_{fund}, U'_{state}, \ldots\}$ | Topology transform |

Table 9: Transaction Type Enumeration System

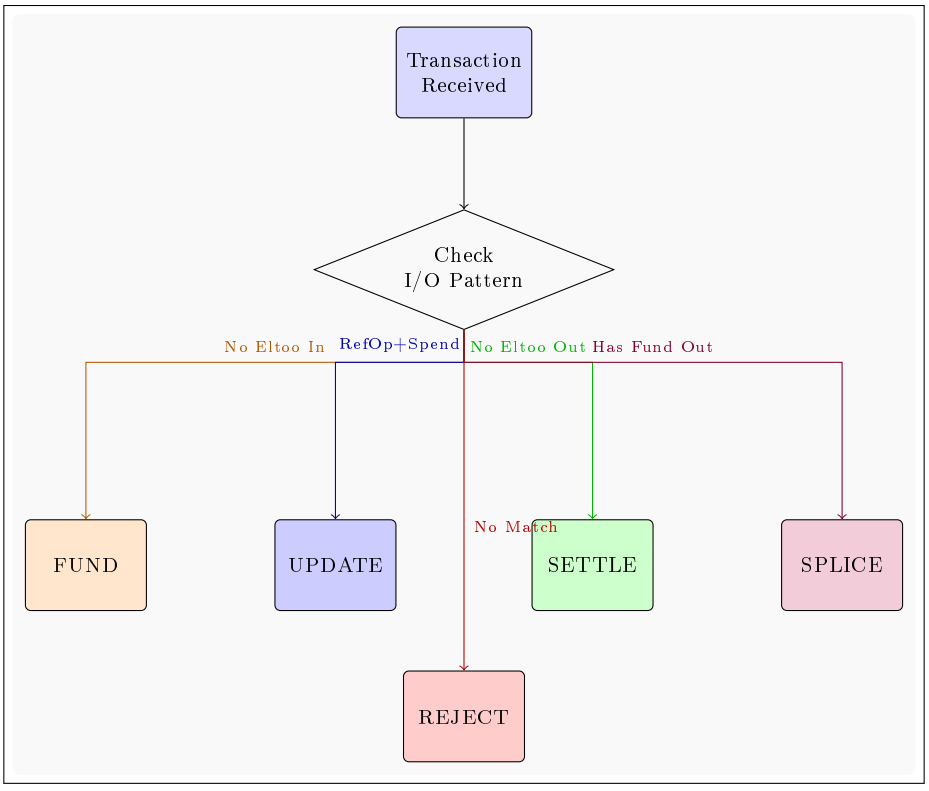


Figure 3: Transaction Type Classification Flow ($O(1)$ Pattern Matching)

### 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.*

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

**Formula Interpretation**: This formula states that for any update transaction $\tau_{update}$, if it transforms state UTXO from version $n$ to version $n'$, then $n'$ must be strictly greater than $n$.

The arrow $\xrightarrow{\tau}$ represents "transformed via transaction $\tau$". In simple terms: **state numbers can only go up, never down**, which fundamentally prevents state rollback attacks.

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

1. **Parsing Phase**: Extract $U_{state}^{(n)}$ from $\tau_{update}$ 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 $\tau_{update}$ is on-chain, old state $U_{state}^{(n)}$ is consumed and cannot be used as input again
4. **Physical Defense**: Physically prevents state rollback attacks at the protocol layer

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

#### 4.1.3 Consensus Verification Performance Analysis

Since transaction types are identified via pattern matching ($O(1)$), monotonicity is checked via integer comparison ($O(1)$), and signatures are verified via aggregation ($O(1)$), total verification complexity is only $O(\log N)$ (UTXO lookup). Compared to Script-based solutions' $O(\text{script\_size} + \log N)$, performance improvement is significant.

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

| 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 |

Table 10: Consensus Verification Latency

**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 $\tau_a, \tau_b$ referencing the same $U_{fund}$:*

1. ***Exclusive Write**: If both $\tau_a, \tau_b$ 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 $\tau_a, \tau_b$ 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:*

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

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

**Formula Interpretation**: This formula defines a mapping function Lease from Fund UTXO set $\mathcal{U}_{fund}$ to transaction IDs. Specifically:

- **Input**: A Fund UTXO (channel's static anchor)
- **Output**: The transaction ID of the last valid update for that channel

This mechanism is analogous to a "lock lease"—each channel's Fund UTXO records which update transaction most recently modified it, 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:*

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

*where $\lambda$ is a convergence constant positively correlated with parameter $k$.*

**Formula Interpretation**:

- $P(\cdot)$ denotes probability
- $d$ represents block depth (the greater the depth, the more confirmations)
- $e^{-\lambda d}$ is an exponentially decaying function

**Intuitive meaning**: As block depth increases, the probability of state forks decreases exponentially. For example, if $\lambda = 0.5$, then at depth $d = 10$, the fork probability is less than $e^{-5} \approx 0.7\%$. This guarantees that channel states eventually converge to a unique consistent state.

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

□

| 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 |

Table 11: Concurrent Safety Analysis

### 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 $\tau_{update}$ be confirmed in block $B_i$, generating $U_{state}^{(n)}$. Let $\tau_{settle}$ be broadcast in block $B_j$, referencing $U_{state}^{(n)}$. $\tau_{settle}$ 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 $\tau'_{update}$ overwrites $U_{state}^{(n)}$, that state UTXO will persist in the ledger:*

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

**Formula Interpretation**:

- $\forall t \in [t_{confirm}, \infty)$: For all times $t$ from the confirmation moment to infinity
- $\nexists \tau'_{update}$: If there exists no new update transaction
- $U_{state}^{(n)} \in \mathcal{U}_{chain}(t)$: Then state UTXO $n$ remains in the on-chain UTXO set

**Plain language**: Once a state is confirmed on-chain, unless someone submits a newer state update, this state will **permanently** remain in the ledger. This allows users to safely settle at any later time based on this confirmed state.

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

**Practical Significance**: This property is crucial for GhostDAG, as it allows users to safely settle based on confirmed stable states even during network partitions or high latency, without worrying about temporary state invisibility caused by block reorganization.

### 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 $\tau \to \bot$ (non-spendable), serving only as an oracle providing metadata access to $U_{fund}$.*

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

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

**Formula Interpretation**:

- $\Gamma$ is a **classification function** that takes transaction inputs $In$ and outputs $Out$ as parameters
- $\mathcal{I}^*$ represents a sequence of input types, $\mathcal{O}^*$ represents a sequence of output types
- Symbol $\cong$ means "structurally isomorphic" (i.e., contains exactly these type components)
- Symbol $\wedge$ means "and", $\cap$ means set intersection, $\emptyset$ means empty set
- $\bot$ (bottom) represents "undefined/invalid"

**Reading guide**: Each branch of this piecewise function defines a transaction type. For example, the second line states: "If inputs are {FundRef, StateSpend} AND output is {ChannelState}, then this is an UPDATE transaction." This enables the system to classify transactions in $O(1)$ time just by examining I/O structure.

**Pattern Explanation**:

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

**Theorem 4.10** (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* $\Gamma$ *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.*

| 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 |
| $\Gamma$ function | `EltooTxType::classify()` | $O(1)$ pattern matching |
| $\bot$ case | `ConsensusError::InvalidEltooTxType` | Reject invalid transactions |

Table 12: Type System Implementation Mapping

## 4.2 Finite State Machine Formalization

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

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

**DFA Five-Tuple Intuitive Explanation**: A DFA is a mathematical model describing a system that can be in one of a finite number of states and can transition between states based on inputs. It's analogous to the relationship between floors and elevator buttons: the current floor is the state, button presses are inputs, and the destination floor is the new state.

**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

  *Note: $n$ is sequence number, $R_b$ is balance Merkle root, $R_p$ is PTLC Merkle root, $t$ is timestamp. The symbol $\mid$ reads as "such that", defining element conditions.*

- $\Sigma$: Transaction alphabet. $\Sigma = \{\tau_{fund}, \tau_{update}, \tau_{splice}, \tau_{settle}, \tau_{timeout}\}$

  *Note: This is the set of all possible "inputs" (transactions) that can trigger state transitions.*

- $\delta$: State transition function. $\delta : Q \times \Sigma \rightharpoonup Q$ (partial function)

  *Note: $\delta$ takes current state and transaction, returns new state. The symbol $\rightharpoonup$ indicates a partial function (not all (state, transaction) pairs have defined transitions).*

- $q_0$: Initial state. $q_0 = q_{init}$, representing virtual state before channel creation

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

  *Note: Reaching a final state means the channel lifecycle is complete.*

**Definition 4.11** (State Space Structure)**.** *State space $Q$ constitutes a **partially ordered set (Poset)** $(Q, \preceq)$, where:*

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

*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.

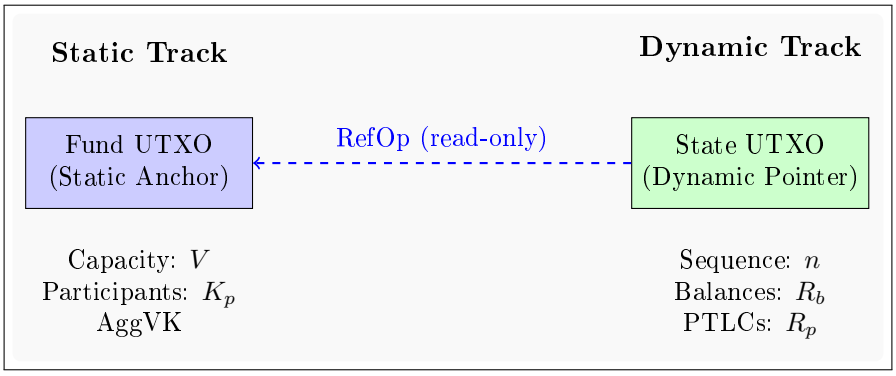


Figure 4: Dual-Track State Machine Architecture

**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:

| 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 |

Table 13: Dual-Track Model Components

- $U_{fund}$: 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 $n \in \mathbb{N}$
  - 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* $\text{Ref} : \mathcal{U} \to \mathcal{U}^{readonly}$*:*

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

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

**Formula Interpretation**:

- Ref is a **read-only reference operator** that only reads UTXO information without consuming it
- Symbol $\triangleq$ means "defined as"
- $\langle \cdot, \cdot \rangle$ represents an ordered pair (tuple)
- The condition $\forall \tau : \text{Ref}(U) \in \text{inputs}(\tau) \Rightarrow U \in \text{UTXO\_Set}_{\text{post}(\tau)}$ states: "For any transaction $\tau$, if it references $U$ via RefOp, then $U$ still exists in the UTXO set after $\tau$ executes"

**Analogy**: RefOp is like "looking at your bank account balance"—you can read the balance but don't withdraw money. Traditional Spend operations are like "withdrawing money"—after withdrawal, the balance is consumed.

### 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). *$\delta$ is defined by the following rules:*

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

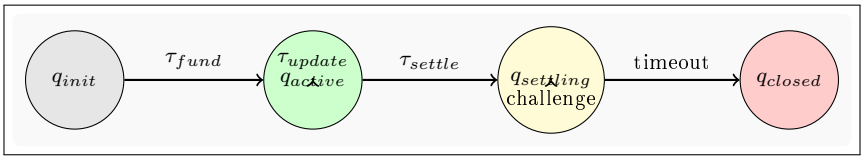


Figure 5: Channel State Machine Transitions

**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 $\delta^*$ is the transitive closure of $\delta$, 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 $q^{(n)}_{active}$, 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**:

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

**Update Transaction**:

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

**Splice Transaction**:

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

**Settle Transaction**:

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

### 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 function-based simple locking to algebraic structure-based homomorphic locking.

#### 4.7.1 Historical Evolution

**HTLC Origin and Limitations (2016)**

Hash Time-Locked Contract (HTLC) was first formalized by Poon and Dryja in the 2016 Lightning Network whitepaper.

- **Mechanism**: Uses SHA-256 hash function's one-wayness. Receiver generates secret $R$ (preimage), broadcasts its hash $H = \text{SHA256}(R)$ along the path. All intermediate nodes construct script: `OP_SHA256 <H> OP_EQUAL`.
- **Historical Significance**: HTLC was a pragmatic choice in an era when Bitcoin Script capabilities were limited (only ECDSA support, no complex algebraic operations). It could be implemented in Bitcoin Script without soft forks.
- **Defect Exposure**: As network scale grew, researchers discovered HTLC has severe **privacy correlation defects**. Since the same hash value $H$ traverses the entire payment path, attackers controlling multiple nodes can easily correlate sender and receiver (Wormhole Attack / Correlation Attack).

**Scriptless Scripts and Schnorr Enlightenment (2017-2019)**

Andrew Poelstra proposed the concept of "Scriptless Scripts" in 2017, exploring how to leverage Schnorr signature's algebraic properties to implement contract logic without script exposure.

**PTLC Formalization (2019-Present)**

PTLC matured as a concept with Taproot activation (2021). Its core idea is replacing hash locks with point locks:

- Hash lock: $y = H(x)$, proving knowledge of preimage $x$
- Point lock: $Q = s \cdot G$, proving knowledge of scalar $s$ (discrete logarithm)

### 4.7.2 Technical Principle Comparison

**HTLC: Hash-Based Rigid Locking**

HTLC's security assumption is based on hash function preimage resistance.

- **Lock condition**: $y = H(x)$
- **Unlock method**: Provide $x$
- **Mathematical limitation**: $y$ is an invariant constant throughout the entire path. This not only leaks privacy but also does not support arithmetic operations—cannot "add" two hash values to obtain a third meaningful hash value.

**PTLC: Scalar-Based Algebraic Locking**

PTLC's security assumption is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP).

- **Lock condition**: $Q = s \cdot G$, where $G$ is the base point, $Q$ is a public key point
- **Unlock method**: Provide scalar $s$ such that the equation holds
- **Algebraic advantage**: Utilizing elliptic curve **additive homomorphism**:

$$Q_{total} = Q_1 + Q_2 \iff s_{total} = s_1 + s_2$$

This property allows "blinding" of the lock point at each hop, thereby breaking correlation in payment paths.

**Additive Homomorphism Detailed Explanation**:

- **Mathematical meaning**: If $Q_1 = s_1 \cdot G$ and $Q_2 = s_2 \cdot G$, then $Q_1 + Q_2 = (s_1 + s_2) \cdot G$
- **Symbol** $\iff$: Means "if and only if", i.e., the two conditions are equivalent
- **Practical application**: Each intermediate node can add a random blinding factor $r_i$ to construct new lock point $Q_i' = Q + r_i \cdot G$. Externally, each hop sees a different $Q_i'$, but ultimately all $r_i$ can be combined through algebraic properties to unlock the original $Q$.
- **Analogy**: Like adding different "disguises" to a secret at each leg of a relay, but at the destination all disguises can be removed to reveal the original secret.

### 4.7.3 Core Properties Comparison

### 4.7.4 Formal Security Analysis

**Theorem 4.18** (PTLC Redemption Uniqueness)**.** *Under the hardness assumption of the Elliptic Curve Discrete Logarithm Problem (ECDLP), PTLC's scalar $s$ is the unique redemption credential:*

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

**Formula Interpretation**:

- $\forall Q \in \mathcal{E}$: For any point $Q$ on elliptic curve $\mathcal{E}$

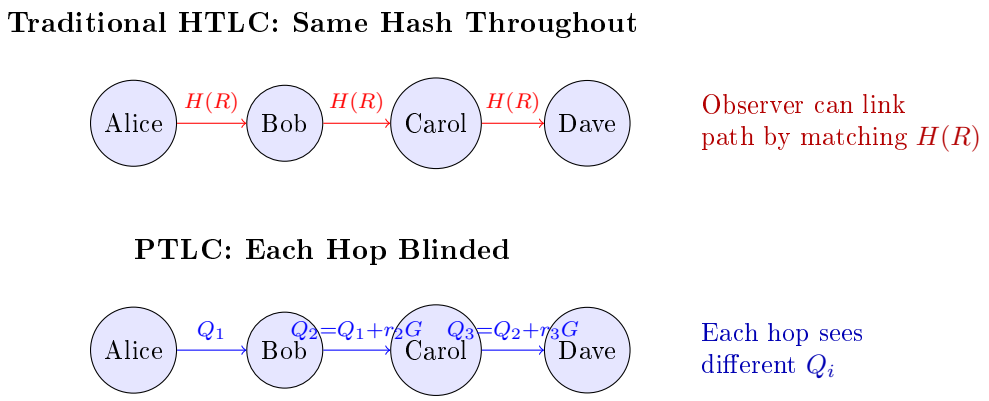


Figure 6: Multi-Hop Blinding: HTLC vs PTLC

| Dimension | HTLC | PTLC | Difference Analysis |
|---|---|---|---|
| Privacy | Weak (path correlatable) | Strong (path decorrelated) | PTLC supports multi-hop blinding |
| Verification Cost | $O$(ScriptSize) | $O(1)$ | HTLC needs script interpreter |
| Batch Verification | Not supported | Supported | Schnorr signature batch verification |
| Functional Extension | Limited | Programmable | Supports Barrier Escrows, etc. |
| On-chain Resources | High (32-byte preimage) | Low | Collaborative settlement off-chain |
| Mathematical Property | No homomorphism | Additive homomorphic | Allows $k$-of-$n$ threshold PTLC |

Table 14: HTLC vs PTLC Comparison

- $\exists!$: "There exists exactly one" (existence and uniqueness)
- $s \in \mathbb{Z}_n$: $s$ is an integer in the finite field $\mathbb{Z}_n$ (where $n$ is the curve order)
- $Q = s \cdot G$: Point $Q$ equals base point $G$ multiplied by scalar $s$

**Security meaning**: For each lock point $Q$, there is one and only one scalar $s$ that can unlock it. This uniqueness is guaranteed by the computational hardness of ECDLP—even knowing $Q$ and $G$, it's computationally infeasible to find $s$.

**Theorem 4.19** (Multi-Hop Atomicity). *For path $P = c_1 \to c_2 \to ... \to c_n$, when all hops use the same base Point Lock $Q$:*

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

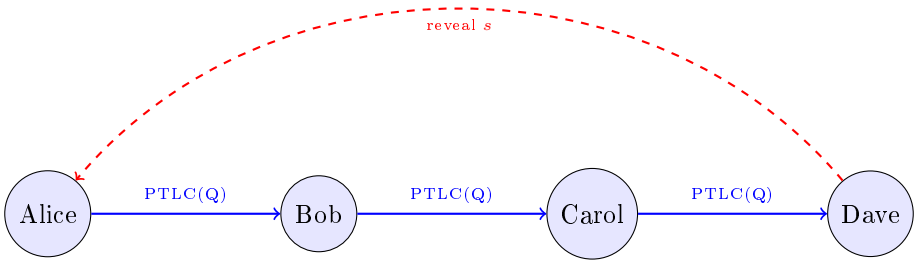


Figure 7: PTLC Multi-Hop Atomic Payment ($Q = s \cdot G$)

*Proof.* 1. Recipient claims funds at $c_n$ by revealing $s$

2. Once $s$ is public, each intermediate node can use $s$ to unlock its adaptor signature

3. Due to decreasing timelocks ($\Delta t_i > \Delta t_{i+1}$), each node has sufficient time to claim its share
Therefore, PTLC paths satisfy atomicity. □

**Theorem 4.20** (Timeout Refund Safety)**.** *If the recipient does not claim before CSV timeout, the sender can safely recover funds:*

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

*This mechanism is protected by DAA Score providing manipulation-resistant time measurement.*

### 4.7.5 Implementation Considerations

Transforming PTLC from theory to engineering implementation requires solving the following key problems:

**Adaptor Signature Verification**:

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

| Operation | HTLC (Script) | PTLC (Algebraic) |
|---|---|---|
| Locking | `OP_SHA256 <H> OP_EQUAL` | Store $Q$ (32 bytes) |
| Unlocking | Provide 32-byte preimage | Adaptor signature conversion (off-chain) |
| Verification | SHA256 + script execution | 1 point multiplication + 1 point addition |
| Batch optimization | None | $O(n/\log n)$ Strauss algorithm |

Table 15: Programming Complexity Comparison: HTLC vs PTLC

### 4.7.6 Summary

The evolution from HTLC to PTLC represents a paradigm shift in conditional payment primitives from "knowledge-based proofs" to "algebra-based proofs". This transformation is not an innovation of any specific protocol, but rather a natural evolution following the maturation of cryptographic infrastructure (Schnorr signatures, Taproot). PTLC's advantages—privacy, efficiency, programmability—have been widely recognized and are being explored for implementation in multiple projects.

## 4.8 TLA+ Specification Fragment

The channel state machine can be formally specified using TLA+ for model checking:

```
--------------------------- MODULE EltooChannel ---------------------------
VARIABLES state, seq_num, phase

Phases == {"init", "active", "settling", "closed"}

Init == /\ state = "init"
        /\ seq_num = 0
        /\ phase = "init"

Fund == /\ phase = "init"
        /\ phase' = "active"
        /\ seq_num' = 0
        /\ UNCHANGED state

Update == /\ phase = "active"
          /\ seq_num' > seq_num  \* Monotonicity enforced
          /\ UNCHANGED phase

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

Challenge == /\ phase = "settling"
             /\ seq_num' > seq_num  \* Higher state challenge
             /\ UNCHANGED phase

Timeout == /\ phase = "settling"
           /\ phase' = "closed"
           /\ UNCHANGED seq_num

Next == Fund \/ Update \/ Settle \/ Challenge \/ Timeout

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

This specification can be verified using the TLC model checker for properties `Monotonicity` and `EventualTermination`.

## 4.9 Cost and Parameter Analysis under GhostDAG

To clarify the impact of L1 parameters on L2 security and cost, this section provides a transparent cost model.

### 4.9.1 Cost Composition Model

Total user cost $C_{total}$ in this architecture consists of three components:

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

**Key Advantage**: This architecture's off-chain updates require no routing fees, contrasting with traditional Lightning Network's HTLC routing fee model.

| Cost Item | Meaning | Reference Value |
|---|---|---|
| $C_{open}$ | On-chain fee to open channel | 1 `FUND` tx (∼250 Bytes) |
| $C_{update}$ | Cost per state update | **0 Gas** (pure off-chain) |
| $C_{settle}$ | On-chain fee to settle channel | 1 `SETTLE` tx (∼300 Bytes) |
| $N$ | Number of off-chain state updates | Unlimited (L1 cost only on dispute) |

Table 16: Cost Composition Model

#### 4.9.2 Impact of GhostDAG Parameter $k$

GhostDAG's width parameter $k$ directly affects confirmation speed and security.

**Confirmation Time Formula**:

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

where:

- $D$: Network delay constraint (seconds)
- $k$: GhostDAG width parameter (maximum concurrent blocks)
- $\epsilon$: Security level (e.g., $10^{-6}$ means one-in-a-million reorg probability)

**Practical Values**: For $k = 16$, confirmation time to reach $10^{-6}$ security level is approximately **3 seconds**.

#### 4.9.3 Ref-UTXO Security Depth

Ref-UTXO security depends on $U_{fund}$ not being deeply reorganized. We recommend:

$$\text{Min\_Ref\_Depth} = 10 \text{ DAA Score}$$

| System | Secure Confirmation Time | User Experience |
|---|---|---|
| Bitcoin (6 blocks) | ∼60 minutes | Long wait |
| This Architecture (10 DAA) | ∼3-5 seconds | Near-instant |

Table 17: Security Confirmation Time Comparison

# 5 Topological Primitives for Complex Structures

## 5.1 Recursive Channel Factories

Channel factories are one of the core primitives of this paper's architecture, allowing "splitting" of multiple sub-channels from a parent channel, with each sub-channel being an independent state machine.

**Definition 5.1** (Channel Factory). *A channel $C_{parent}$ can generate a set of sub-channels $\{C_{child_i}\}$ through $\tau_{splice}$ transaction, and once created, the sub-channels' lifecycles are completely decoupled from the parent channel.*

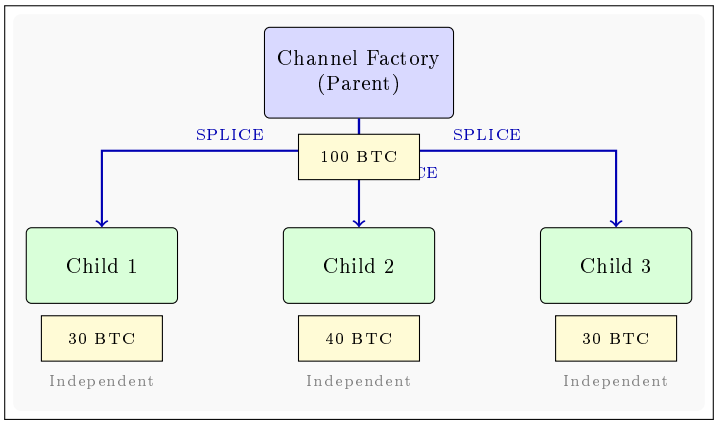


Figure 8: Recursive Channel Factory Structure

### 5.1.1 Key Topological Invariants

- **Isolation**: $C_{child}$ settlement does not depend on $C_{parent}$ state
- **Independence**: $C_{child}$ can perform arbitrarily complex operations without affecting $C_{parent}$
- **Nestability**: $C_{child}$ can serve as parent channel for $C_{grandchild}$

### 5.1.2 Fractal Topology and Self-Similarity

This paper's architecture allows recursive channel factories to enable a channel to spawn sub-channels, which in turn spawn grandchild channels. This structure topologically manifests as a **self-similar $k$-ary tree**, embodying the property of fractal geometry where simple rules produce complex structures through iteration.

**Definition 5.2** (Split Operator). *Define mapping operator $\Phi : \mathcal{C} \to \{\mathcal{C}_1, ..., \mathcal{C}_k\}$ as the channel split operation. When recursion depth $d \to \infty$, the system exhibits characteristics of **fractal geometry: scale invariance**.*

*Regardless of layer 0 (L1 main chain) or layer $n$ (deep sub-channels), verification logic $V$ for state updates $\tau_{update}$ and settlement $\tau_{settle}$ remains completely consistent:*

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

**Theorem 5.3** (Liquidity Conservation). *For any depth $d$, the sum of capacities of all active nodes at that level equals root node capacity (ignoring gas losses):*

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

*Proof.* By induction:

1. **Base case**: For $d = 0$, trivially holds
2. **Inductive hypothesis**: Assume holds for $d = k$
3. **Inductive step**: For $d = k + 1$, by strong value conservation, each $\Phi$ operation preserves total value

Therefore the theorem holds. □

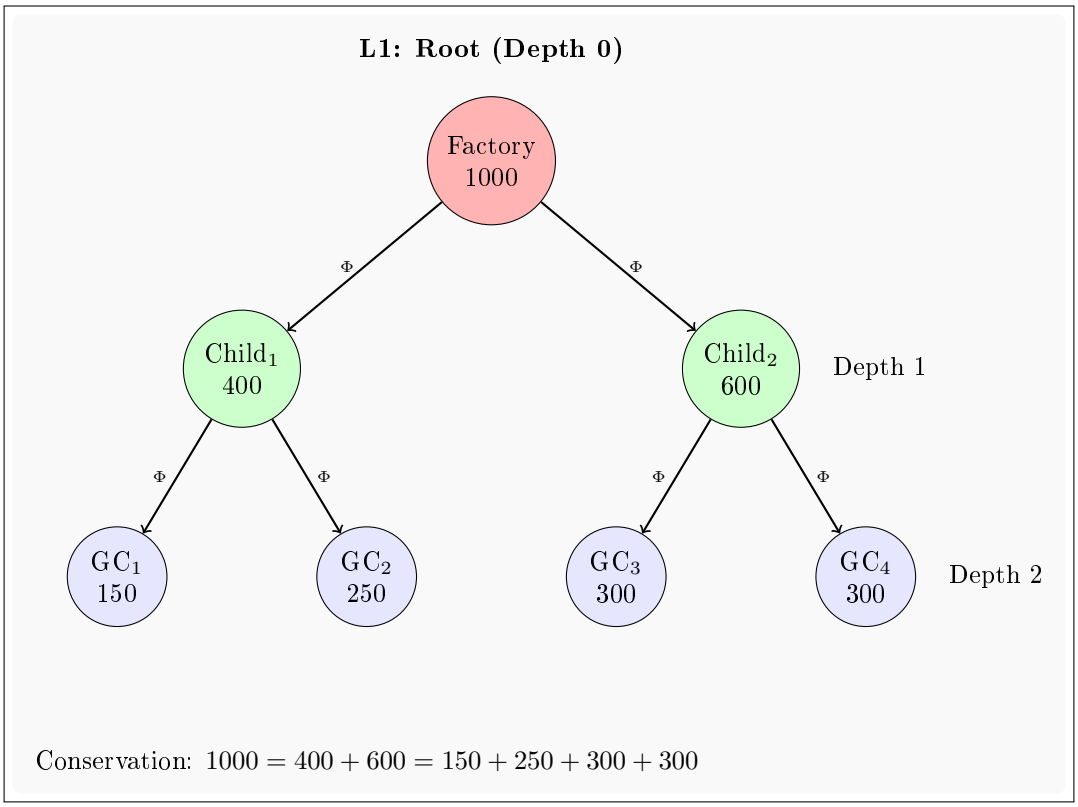


Figure 9: Fractal Channel Tree: Self-Similar Structure

## 5.2 Dynamic Mesh Reconfiguration

**Definition 5.4** (Topological Isomorphism)**.** *Two channel networks $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are topologically isomorphic if there exists a bijection $\phi : V_1 \to V_2$ such that $\forall (u, v) \in E_1, (\phi(u), \phi(v)) \in E_2$.*

**Theorem 5.5** (Atomic Reconfiguration)**.** *Any topologically isomorphic channel networks can be atomically transformed through a single $\tau_{splice}$ transaction.*

*Proof.* Guaranteed by UTXO model atomicity. A $\tau_{splice}$ transaction either succeeds completely or fails completely; intermediate states are not visible. □

### 5.2.1 Topological Homotopy and Atomic Transformation

We view channel reconfiguration as a **homotopic transformation** on graph $G$. The $\tau_{rebalance}$ operator actually defines a continuous transformation path $f : G_{star} \to G_{mesh}$. During this transformation, the network's total energy (i.e., total locked value TVL) remains constant ($\frac{dE}{dt} = 0$), only the distribution of energy across edges is reconfigured.

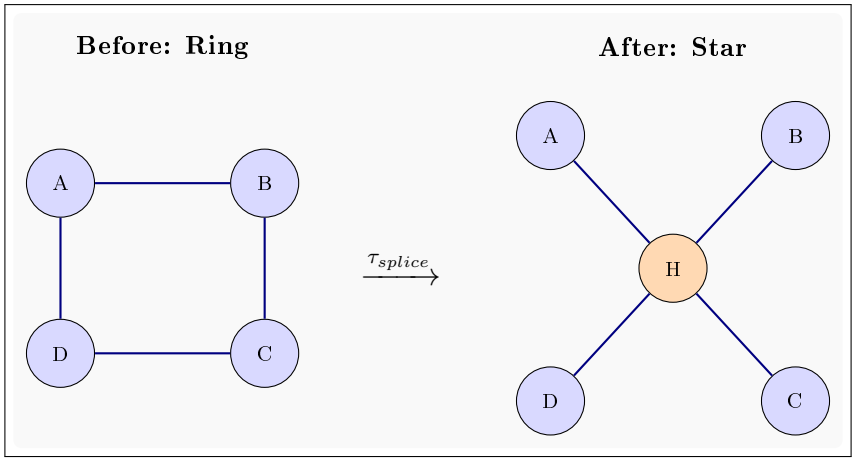


Figure 10: Atomic Topology Reconfiguration: Ring to Star

**Definition 5.6** (Topological Homotopy). *Let $G = (V, E, w)$ be a weighted channel graph, where $w : E \to \mathbb{R}^+$ is the liquidity weight function. Topological homotopy transformation $\mathcal{H}$ is defined as:*

$$\mathcal{H} : G_1 \simeq G_2 \iff \exists \tau \in \Sigma_{splice} : \delta(G_1, \tau) = G_2 \wedge \sum_{e \in E_1} w(e) = \sum_{e \in E_2} w(e)$$

*This property guarantees no liquidity vacuum occurs during financial network restructuring.*

**Energy Conservation Constraint**:

$$E_{before} = 10 + 90 + 5 = 105 \text{ BTC} = 40 + 40 + 25 = E_{after}$$

This mathematical property is analogous to the law of conservation of energy in physics: resources can be redistributed within the system, but total quantity cannot be created or destroyed from nothing.

## 5.3 Atomic Rebalancing Operator and Value Conservation

We define the reconfiguration operation $\tau_{rebalance}$ of channel factories as an atomic combination of basic topological transformations.

**Definition 5.7** (Rebalance Operator). *$\tau_{rebalance}$ is a mapping $\Omega : \mathcal{S}_{parent} \times \{\mathcal{S}_{child}\}_M \to \mathcal{S}'_{parent} \times \{\mathcal{S}'_{child}\}_N$, where $M$ is the number of input sub-channels and $N$ is the number of output sub-channels.*

**Invariant 4.1 (Strong Value Conservation)**:

$$V(U_{fund}^{parent}) + \sum_{i=1}^{M} V(U_{fund}^{child\_i}) = V(U_{fund}^{parent'}) + \sum_{j=1}^{N} V(U_{fund}^{child\_j}) + \delta_{fee}$$

**Theorem 5.8** (Rebalancing Atomicity). *Since $\tau_{rebalance}$ is a single on-chain transaction, GhostDAG consensus guarantees atomicity of the state transition.*

*Proof.* 1. UTXO model atomicity: transactions either fully apply or don't apply at all

2. GhostDAG's total order ensures deterministic effect in any consensus snapshot

3. Value conservation is mandatorily checked during validation; violators are rejected

□

**Algorithm 4.1 (Deterministic Sub-channel ID Derivation)**:
To ensure auditability and prevent ID collisions after reconfiguration, new sub-channel IDs must be derived deterministically via hashing, independent of user input:

$$ID_{child\_j} = \text{BLAKE3}(Domain_{sub} \parallel ID_{parent} \parallel OutPoint_{fork} \parallel j \parallel Root_{participants})$$

| Parameter | Meaning | Purpose |
|---|---|---|
| $Domain_{sub}$ | Domain identifier | Prevent cross-protocol collisions |
| $ID_{parent}$ | Parent channel ID | Inheritance relationship |
| $OutPoint_{fork}$ | Fork transaction outpoint | On-chain anchor |
| $j$ | Sub-channel index in output | Ordering uniqueness |
| $Root_{participants}$ | Participant pubkey Merkle root | Identity binding |

Table 18: Sub-channel ID Derivation Parameters

**Security Guarantee**: Since BLAKE3 is collision-resistant and all inputs are on-chain verifiable, each sub-channel has a globally unique addressing identifier even in deep recursion ($Depth \leq 3$).

## 5.4 Atomic Splicing Protocol

Simply describing transaction structure is insufficient; Splicing is fundamentally a **Multi-Party Computation (MPC) protocol**. This section defines the non-blocking reconfiguration protocol, addressing key challenges:

- What if one party goes offline during Splicing signature process?
- What if one party attempts to broadcast old state during Splicing?
- How to achieve "zero-downtime" channel reconfiguration?

**Definition 5.9** (Non-blocking Splicing Protocol)**.** *Traditional channel reconfiguration (e.g., Lightning Loop) requires "stop-the-world" maintenance where channels are unavailable during reconfiguration. This architecture implements* ***Zero-Downtime Splicing****.*

**Protocol Phases**:

| Phase | Operation | Key Feature |
|---|---|---|
| Phase 1: Proposal | Alice constructs $\tau_{splice}$, broadcasts to participants | Timeout: $T_{ack} = 3$ |
| Phase 2: Async Signing | Participants generate partial signatures, **channel stays active** | "Changing wheels ing train" |
| Phase 3: Broadcast Convergence | Broadcast $\tau_{splice}$, DAG ordering resolves conflicts | Rollback mechani sures safety |
| Phase 4: Commitment Transfer | New $U^{(0)'}_{state}$ inherits old state's Merkle roots | Atomic PTLC mig |

Table 19: Non-blocking Splicing Protocol Phases

**Theorem 5.10** (Non-blocking Guarantee)**.** *During protocol $\Pi_{splice}$ execution, channel payment capability remains unaffected.*

*Proof.* 1. **Phase 2 Parallelism**: $\tau_{splice}$ signing process does not consume any UTXO, so old channel state $U^{(n)}_{state}$ can still be operated by normal $\tau_{update}$

2. **State Fork Handling**: If $\tau_{splice}$ and some $\tau_{update}$ broadcast nearly simultaneously:
   - Case A: $\tau_{update}$ on-chain first $\rightarrow$ $\tau_{splice}$'s referenced $U_{state}^{(n)}$ invalidated $\rightarrow$ protocol rollback
   - Case B: $\tau_{splice}$ on-chain first $\rightarrow$ new topology takes effect, $\tau_{update}$ rejected as double-spend
3. **Atomicity Guarantee**: Either case, channel always in valid state, no "half-reconfiguration" intermediate state exists

Therefore, channel liquidity continuity is guaranteed. □

**Game-Theoretic Analysis: Holdup Attack Cost**

Malicious participants may attempt "holdup" attacks during Splicing—deliberately delaying signatures to extort other participants.

| Strategy | Attacker Gain | Honest Party Loss |
|---|---|---|
| Honest signing | $r \cdot V_{share} \cdot T$ | 0 |
| Holdup attack | $\epsilon$ (extortion) | $r \cdot V_{total} \cdot T_{delay}$ |
| Honest party rollback | 0 | $O(gas_{failed})$ |

Table 20: Holdup Attack Game-Theoretic Analysis

**Defense Mechanism**: Due to this architecture's non-blocking design, honest participants can immediately rollback to normal operation after $T_{sign}$ timeout. Maximum gain from holdup attack is limited to liquidity loss during $T_{sign}$ period, while attacker equally bears this loss.

**Corollary**: Under rational economic actor assumption, holdup attacks do not constitute Nash Equilibrium; honest cooperation is the dominant strategy.

## 5.5 Liquidity Dynamics in Star Topologies

### 5.5.1 Liquidity Utilization Definition

Define channel factory $F$ as a star graph $G = (V, E)$, where central node Hub connects $N$ leaf nodes.

**Liquidity Utilization**:

$$U(t) = \frac{\sum_{i=1}^{N} |\text{Flow}_i(t)|}{\sum_{i=1}^{N} \text{Cap}_i}$$

where $U(t)$ is utilization at time $t$, $\text{Flow}_i(t)$ is instantaneous flow in channel $i$, and $\text{Cap}_i$ is capacity of channel $i$.

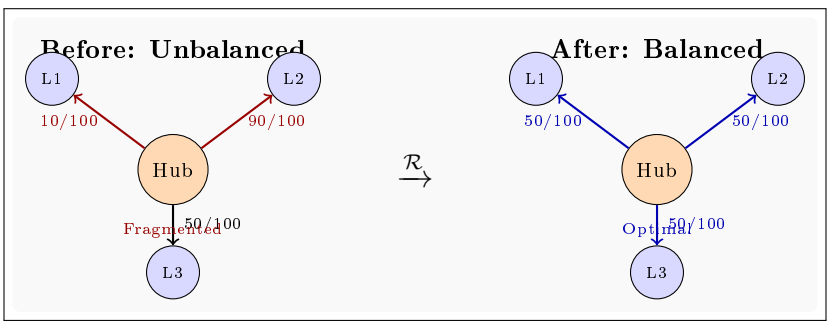


Figure 11: Atomic Rebalance: Zero-Friction Liquidity Transfer

**Theorem 5.11** (Balanced Flow Optimal Allocation)**.** *For any flow distribution $\vec{f}$, there exists a reconfiguration strategy $\mathcal{R}$ minimizing liquidity fragmentation:*

$$\min_{\mathcal{R}} \sum_{i=1}^{N} (Cap'_i - f_i)^2 \quad s.t. \ \sum Cap'_i = \sum Cap_i$$

**Theorem 5.12** (GhostDAG Flow Lower Bound)**.** *For $\mathcal{R}$ to be economically feasible, L1 throughput must satisfy:*

$$TPS_{L1} \geq \frac{F_{rebalance}}{BlockSize} \times \alpha$$

*where $F_{rebalance}$ is reconfiguration frequency and $\alpha$ is safety margin.*

**Corollary**: Channel factories adopting this architecture can maintain near **100% capital efficiency**, difficult to achieve in traditional payment channel networks.

# 6 Safety Analysis

## 6.1 Isolation Theorem

**Theorem 6.1** (Channel Isolation)**.** *Sub-channel $C_{child}$ security is not affected by parent channel $C_{parent}$ liveness or malicious behavior.*

*Proof.* 1. **Physical Layer Isolation**: $U_{fund}^{child}$ exists as independent UTXO on L1

2. **Logical Layer Isolation**:
   - $C_{child}$'s $\tau_{update}$ only references $\text{Ref}(U_{fund}^{child})$
   - $C_{child}$'s $\tau_{settle}$ only depends on $U_{state}^{child}$
3. **Settlement Independence**:
   - Parent channel settlement does not affect sub-channel UTXO
   - Even if parent channel is maliciously settled, as long as $U_{fund}^{child}$ creation transaction is confirmed, sub-channel security is unaffected
4. **Temporal Isolation**: Each channel has independent CSV timer based on DAA Score rather than block height

Therefore, sub-channels achieve complete isolation from parent channels at both physical and logical layers. □

## 6.2 State Monotonicity and Anti-Replay

Based on Theorem 1 (Section 3.1.2) monotonicity guarantee, we further prove cross-topology anti-replay security.

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

*Proof (Brief).* 1. **RefOp-OutPoint Binding**: $\sigma = \text{Sign}_{sk}(\text{state}_n \| \text{RefOp\_OutPoint})$

2. **Topology Changes TxID**: $\tau_{splice}$ creates new $U'_{fund}$, thereby changing RefOp_OutPoint
3. **Key Derivation Isolation**: $AggVK_{child} = H(AggVK_{parent} \| \text{index}) \neq AggVK_{parent}$

Therefore: $\forall \sigma_{old} : \nexists$ valid replay in $C_{new}$ □

**Corollary**: Each Splicing naturally forms a cryptographic barrier, preventing state replay both within channels and across channels.

## 6.3 Anti-DoS Equilibrium under STPC Strategy

Traditional payment channel networks rely on "state count limits" to prevent mempool flooding, but this introduces pinning attack risks. This paper's architecture implements **Single-Tip-Per-Channel (STPC)** strategy, changing the attacker's game payoff matrix by enforcing state uniqueness.

### 6.3.1 Mempool Entropy Bound

The STPC strategy is essentially an **entropy-reducing filter**. In open networks, attackers attempt to increase system thermodynamic entropy (disorder) by broadcasting numerous invalid states. STPC enforces the uniqueness principle, physically constraining maximum entropy $S_{max}$:

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

This means that regardless of how much computational power attackers invest in broadcasting transactions, they cannot break through this information-theoretic entropy bound, theoretically eradicating mempool resource exhaustion DoS attacks.

| Model | Mempool Entropy | Attacker Capability | DoS Upper Bound |
|---|---|---|---|
| Traditional LN | $O(\infty)$ unbounded | Can expand infinitely | None |
| This Architecture STPC | $O(\ln N_{channels})$ | Strictly limited | $\leq N_{channels}$ |

Table 21: Mempool Entropy Comparison

**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 $\tau_{new}$:*

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

**Lemma 6.4** (Mempool Convergence)**.** *Under STPC strategy, mempool size $|\mathcal{M}|$ has linear relationship with active channels $N_{channels}$:*

$$|\mathcal{M}| \leq N_{channels}$$

*Proof.* For any channel $C_i$, STPC rules guarantee at most one $\tau \in \mathcal{M}$ such that $ID(\tau) = C_i$. Therefore space complexity is $O(N)$ and does not grow with attacker's broadcast frequency $f_{attack}$. □

**Theorem 6.5** (DoS Cost Escalation)**.** *STPC strategy escalates effective cost of DoS attacks from $O(1)$ to $O(N)$, where $N$ is state sequence number.*

$$Cost_{effective}^{DoS} = \sum_{i=1}^{k} Cost_{tx}(\tau_i) \propto O(k)$$

Due to uniqueness of $U_{state}$, honest node verification resource consumption is strictly limited to constant level.

**Game-Theoretic Model**:

Consider an attacker attempting to consume validation node resources by broadcasting invalid updates.

- $C_{bw}$: Node verification and broadcast bandwidth cost (constant)
- $Cost_{attack}$: Attacker's transaction construction computational cost and network broadcast cost

| Model | Attacker Cost | Node Load | State Exhaustion Condition |
|---|---|---|---|
| Traditional Model | $O(1)$ per tx | $k \cdot C_{bw}$ | None |
| STPC Model | $O(1)$ per tx | $C_{bw}$ (constant) | $State \geq MAX\_STATE$ |

Table 22: DoS Attack Game-Theoretic Model Comparison

In the STPC model, for an attacker to maintain $k$ valid replacements:

$$State(\tau_k) > State(\tau_{k-1}) > \cdots > State(\tau_1)$$

This means the attacker must monotonically increase their state commitments. Once the attacker exhausts their pre-signed state space (limited by $MAX\_STATE\_INCREMENT$), the attack terminates immediately.

## 6.4 PTLC Atomicity and Deadlock Freedom

### 6.4.1 PTLC Atomicity Theorem

**Theorem 6.6** (PTLC Atomicity)**.** *For any cross-channel payment path $P = c_1 \to c_2 \to \cdots \to c_n$, fund transfer either succeeds completely in all $c_i$ or rolls back completely.*

***Formal Expression**:*

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

*Proof.* Based on cryptographic properties of Adaptor Signatures:

1. **Preimage Propagation**: Once recipient reveals preimage (Scalar $s$) at $c_n$, this preimage becomes the decryption key for $c_{n-1}$
2. **Recursive Unlocking**: Each intermediate node can use received $s$ to unlock previous channel's PTLC, propagating back to $c_1$
3. **Consensus Layer Enforcement**: This architecture's `UPDATE` transactions mandate that all PTLCs include same Point Lock, with consensus layer verification guaranteeing mathematical non-repudiation

Therefore, PTLC paths satisfy atomicity. □

### 6.4.2 Deadlock Freedom Theorem

**Theorem 6.7** (Deadlock Freedom)**.** *Under GhostDAG's partial order structure, no fund freezing exists due to circular dependencies.*

*Proof.* Eltoo 2.0 introduces absolute timeout mechanism based on DAA Score.

1. **Monotonic Timestamp**: All state transitions $\delta(q, \tau)$ are constrained by monotonically increasing DAA timestamp
2. **Proof by Contradiction**: Assume deadlock cycle exists, implying $t_1 < t_2 < \cdots < t_1$
3. **Contradiction**: This violates global monotonicity of DAA Score

Therefore: there exists no deadlock cycle in Eltoo 2.0. □

**Security Guarantee**: Even under network partition or malicious node non-response, DAA timeout mechanism guarantees funds can eventually be recovered by honest parties.

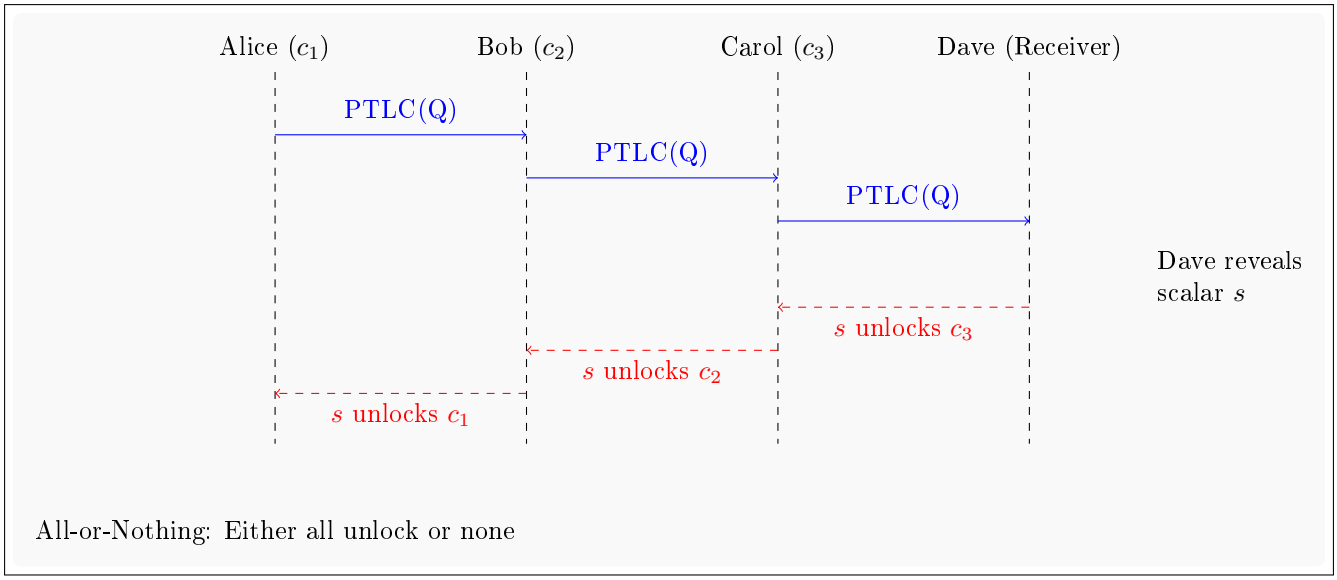


Figure 12: PTLC Multi-Hop Atomic Payment Sequence

### 6.5 Consistency of Topological Reconfiguration

**Theorem 6.8** (Splicing Consistency)**.** *When executing* `SPLICE-FORK` *operations in concurrent environments, the system guarantees:*

1. ***Value Conservation***: $\sum V_{in} = \sum V_{out}$
2. ***Unique History***: *If fork occurs, GhostDAG ultimately selects only one valid topology transition path*

*Proof.* Depends on exclusivity of `Spend(State_UTXO)`.

1. **RefOp-UTXO Concurrent Read**: While RefOp-UTXO allows concurrent reads, Splicing requires spending current State UTXO
2. **GHOST Rule**: According to GhostDAG's GHOST rule, only Splicing transactions in the heaviest-weighted subgraph are confirmed as valid
3. **Conflict Resolution**: Remaining conflicting transactions are discarded, guaranteeing linear consistency of topology evolution

Therefore, Splicing guarantees value conservation and history uniqueness. □

### 6.6 Security Margin Analysis

Based on above comparisons, we conclude the following security margin improvements:

1. **State Theft Defense**: UTXO atomicity + monotonic replacement mechanism eliminates penalty transaction complexity
2. **Replay Attack Defense**: Domain separation (Axiom A6) + type binding provides dual barriers
3. **DoS Resistance**: STPC strategy escalates attack cost from $O(1)$ to $O(N)$

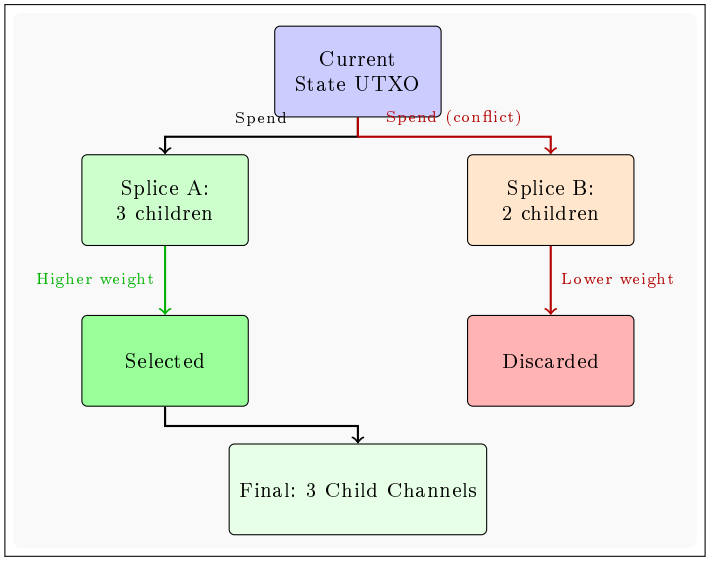


Figure 13: Concurrent Splice Conflict Resolution via GhostDAG

4. **Offline Tolerance**: DAA Score timelocks support week-level offline, reducing watchtower dependency

5. **Recovery Simplicity**: No "toxic waste", only need to backup latest state for complete recovery

**Comprehensive Security Analysis**: This paper's security model pushes complexity down to the consensus layer rather than distributing it to application developers, aligning with the system engineering principle of "centralize complexity at the protocol layer, leave simplicity for the application layer".

# 7 Registry-Free Architecture

## 7.1 Limitations of Global Registry Models

Traditional payment channel network designs (e.g., Lightning Network) rely on global channel registries to maintain network topology. This centralized design introduces the following issues:

1. **Privacy Leakage**: All channels must be publicly announced, exposing funding amounts and participant identities
2. **Scalability Bottleneck**: Global registries become performance bottlenecks as network scale grows
3. **DoS Attack Surface**: Attackers can flood the network with massive fake channel announcements
4. **Censorship Risk**: Centralized registries can become single points of failure subject to censorship

## 7.2 Self-Sovereign Channel Discovery Mechanism

This paper's architecture implements a **registry-free** channel discovery mechanism, where channels exist entirely through on-chain UTXO state without requiring off-chain announcement protocols.

**Definition 7.1** (Self-Sovereign Discovery)**.** *Channel discovery occurs through direct parsing of on-chain UTXO sets, not through gossip protocols:*

$$DiscoverChannel(C) \equiv ParseUTXO(BlockchainState) \rightarrow \{U_{fund}, U_{state}\}$$

### 7.2.1 Discovery Mechanism

1. **Type-Based Filtering**: Nodes scan the UTXO set, identifying Eltoo-type UTXOs through transaction type enumeration
2. **Ownership Verification**: Verify whether local keys have spending authority for discovered channels
3. **State Reconstruction**: Reconstruct complete channel state from UTXO metadata

**Theorem 7.2** (Discovery Completeness)**.** *For any channel $C$ where node $N$ is a participant, $N$ can discover and reconstruct $C$ through on-chain UTXO scanning.*

*Proof.* 1. Channel creation generates deterministic UTXO pair $(U_{fund}, U_{state})$
2. These UTXOs contain all necessary information (participants, balances, sequence numbers)
3. UTXO set is globally consistent and immutable
4. Nodes can verify ownership through signature verification

Therefore, discovery mechanism is complete. □

**Core Property**: Mapping $\Phi$ possesses **idempotency** and **atomicity**. Unlike traditional account models requiring replay of entire transaction history to rebuild state (State Rehydration), this architecture's state reconstruction requires only a single linear scan of current UTXO set snapshot.

$$\text{StateRecovery}(t) = \Phi(\text{UTXO\_Set}(t)) \quad \text{in } O(|\mathcal{U}|)$$

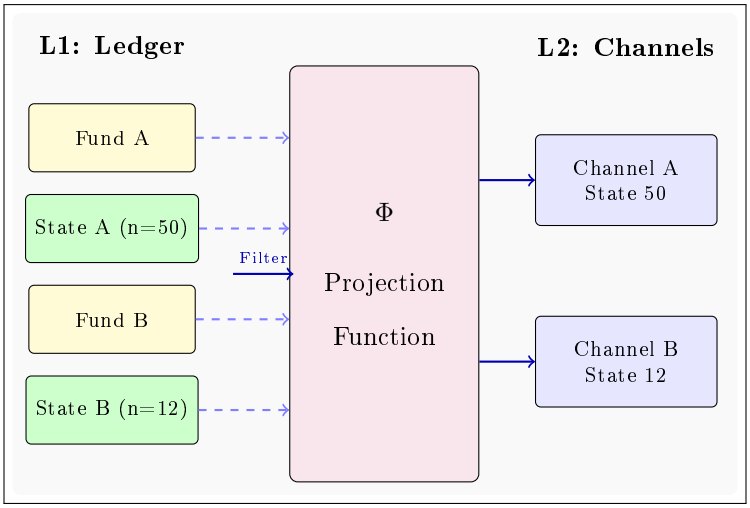


Figure 14: UTXO-to-State Projection: The Chain is the Registry

## 7.3 Privacy Enhancement

### 7.3.1 Unlinkability of Channel Identity

Traditional Lightning Network channels are identified through fixed `channel_id` derived from funding transaction outpoint. This paper's architecture implements **ephemeral identity** mechanism.

**Definition 7.3** (Ephemeral Channel Identity)**.** *Channel identity $ID_C$ changes with each Splice operation:*

$$ID_C^{(i)} = H(domain \| RefOp\_OutPoint_i \| nonce)$$

**Privacy Properties**:

- **Temporal Unlinkability**: $ID_C^{(i)} \not\approx ID_C^{(j)}$ for $i \neq j$
- **Path Decorrelation**: Payment paths cannot be correlated through channel IDs
- **Graph Analysis Resistance**: External observers cannot reconstruct complete network topology

### 7.3.2 Balance Privacy

**Theorem 7.4** (Balance Confidentiality)**.** *Without participant cooperation, external observers cannot determine channel balance distribution.*

*Proof.* Balance information is stored in $U_{state}$ commitment $R_b$:

$$R_b = \text{MerkleRoot}(\{\text{balance}_i\}_{i=1}^{n})$$

1. $R_b$ is a cryptographic hash providing computational hiding
2. Only Merkle proof holders (channel participants) can verify specific balance
3. On-chain only shows $U_{fund}$ total capacity, not internal distribution

Therefore, balance privacy is cryptographically guaranteed. □

| Dimension | Lightning Network | This Architecture |
|---|---|---|
| Discovery Mechanism | Gossip protocol | On-chain UTXO scanning |
| Privacy Level | Public announcement | Self-sovereign |
| Censorship Resistance | Weak | Strong |
| Scalability | $O(N \cdot \log N)$ | $O(N)$ |
| Attack Surface | Gossip flooding | Consensus-bounded |
| Channel Lifecycle | Static identity | Ephemeral identity |
| Balance Privacy | Weak | Strong (commitment-based) |

Table 23: Registry Model Comparison

## 7.4 Comparison with Centralized Registry Models

## 7.5 Economic Incentive Alignment

### 7.5.1 No Announcement Fee Problem

Traditional networks face the dilemma: announcing channels consumes bandwidth but provides no direct economic incentive. This paper's architecture **eliminates this dilemma**.

- **No Announcement Overhead**: Channels exist through on-chain UTXOs without off-chain announcements
- **Natural Discovery**: Nodes discover their own channels through local UTXO indexing
- **Routing Privacy**: Routing nodes do not need to know global topology, only local available channels

### 7.5.2 Discovery Cost Analysis

**Theorem 7.5** (Discovery Cost Bound)**.** *The computational complexity of discovering $M$ owned channels from UTXO set of size $N$ is:*

$$Cost_{discovery} = O(N) + O(M \cdot \log M)$$

**Formula Interpretation**:

- $O(N)$: One-time UTXO set scan
- $O(M \cdot \log M)$: Verification and indexing of owned channels

*Proof.* 1. **Scanning Phase**: Traverse UTXO set once, filtering Eltoo-type UTXOs — $O(N)$

2. **Verification Phase**: For each candidate UTXO, verify signature ownership — $O(M)$ signature verifications
3. **Indexing Phase**: Build index for owned channels — $O(M \cdot \log M)$

Total complexity: $O(N) + O(M \cdot \log M)$ □

**Practical Optimization**: Using Bloom filters and incremental indexing, actual cost can be reduced to near $O(\Delta N)$ (only scan new UTXOs).

## 7.6 Decentralized Routing

### 7.6.1 Source-Based Routing

This paper's architecture adopts **source-based routing** rather than global topology-based routing.

**Definition 7.6** (Source Routing)**.** *Payment sender $S$ specifies complete path $P = [c_1, c_2, ..., c_n]$, with intermediate nodes only forwarding according to path without needing global knowledge.*

**Advantages**:

1. **Privacy**: Intermediate nodes do not know full path
2. **Flexibility**: Sender can optimize paths based on local information
3. **No Global State**: Nodes do not need to maintain complete network topology

### 7.6.2 Onion Routing Integration

Combined with onion routing (e.g., Sphinx protocol), each hop only sees:

$$\text{Visible}_{hop_i} = \{\text{prev\_hop}, \text{next\_hop}, \text{amount}, \text{timelock}\}$$

Cannot see:

- Payment sender identity
- Final recipient identity
- Total path length
- Position in path

**Theorem 7.7** (Path Privacy)**.** *Under onion routing, intermediate nodes cannot infer sender, recipient, or complete path information.*

*Proof.* Sphinx protocol provides **computational indistinguishability**:

$$\text{View}_{adversary}^{hop_i} \approx_c \text{Random}$$

Meaning adversary's view is computationally indistinguishable from random data. □

## 7.7 Registry-Free Architecture Advantages Summary

1. **Enhanced Privacy**: Eliminates mandatory channel announcements, protecting user financial privacy
2. **Censorship Resistance**: Channels exist through on-chain UTXOs, resistant to off-chain censorship
3. **Reduced Attack Surface**: Eliminates gossip protocol flooding attack vectors
4. **Scalability**: Discovery cost grows linearly with UTXO set, not quadratically with channel count
5. **Self-Sovereignty**: Users fully control their channel lifecycle without relying on third-party registries

**Design Philosophy**: This paper's architecture embodies the blockchain principle of "on-chain as source of truth", treating all off-chain components as optional optimization layers rather than necessary infrastructure. This design significantly enhances system **anti-fragility** and **censorship resistance**.

## 7.8 PTLC Verification: $O(1)$ Implementation

### 7.8.1 Verification Algorithm

The PTLC verification leverages RefOp-Fund to achieve constant-time complexity:

```
fn validate_ptlc(settle: &SettleTx, utxo_set: &UtxoSet) -> bool {
    // 1. Obtain Fund UTXO via reference
    let fund_utxo = utxo_set.get_ref(settle.fund_ref);

    // 2. Extract participant keys from Fund UTXO
    let participant_keys = fund_utxo.metadata.participant_keys;

    // 3. Verify curve relationship for each PTLC
    for (i, scalar) in settle.adaptor_scalars.iter().enumerate() {
        let ptlc = &settle.ptlcs[i];
        let point_p = participant_keys[ptlc.beneficiary_idx];
        if !verify_curve_relationship(scalar, point_p, ptlc.point_q) {
            return false;
        }
    }

    true
}
```

### 7.8.2 Complexity Analysis

| Metric | This Architecture | Script-Based |
|---|---|---|
| Time Complexity | $O(k)$ where $k$ = PTLC count | $O(k \times \|script\|)$ |
| Space Complexity | $O(1)$ (no extra storage) | $O(\|script\|)$ |
| State Lookup | Single RefOp read | Script VM stack |

Table 24: PTLC Verification Complexity Comparison

**Key Advantage**: Through RefOp-Fund mechanism, PTLC verification completes entirely within the UTXO paradigm without cross-data-structure queries.

## 7.9 State Anchoring and Settlement Verification

### 7.9.1 Settlement Verification Rules

```
fn validate_settle(settle: &SettleTx, utxo_set: &UtxoSet) -> bool {
    // 1. Verify reference existence
    let fund_utxo = utxo_set.get_ref(settle.fund_ref);
    let state_utxo = utxo_set.get(settle.state_utxo);

    // 2. Verify state matching
    let balances_root = compute_balances_root(settle.balances);
    let ptlcs_root = compute_ptlcs_root(settle.ptlcs);

    if balances_root != state_utxo.balances_root ||
       ptlcs_root != state_utxo.ptlcs_root {
        return false;
```

```
    }

    // 3. Verify timelock
    let current_daa = get_current_daa_score();
    if current_daa < state_utxo.creation_daa + fund_utxo.min_csv {
        return false;
    }

    // 4. Verify signature
    let agg_vk = fund_utxo.agg_vk;
    let msg = build_signature_message(settle, state_utxo.state_number);
    verify_schnorr_signature(settle.signature, agg_vk, msg)
}
```

#### 7.9.2 Security Properties

- **State Consistency**: Settlement must match latest state commitment
- **Temporal Security**: DAA Score provides manipulation-resistant precise timing
- **Signature Integrity**: Domain separation prevents cross-transaction-type signature reuse

### 7.10 Case Study: Atomic Liquidation in DeFi Lending Pools

To demonstrate the architecture's application in complex financial scenarios, we analyze an Aave-like lending protocol's liquidation scenario.

#### 7.10.1 Scenario Setup

Lending pool $P$ needs to simultaneously liquidate 100 under-collateralized users $\{U_1, ..., U_{100}\}$ during market volatility, transferring collateral to liquidator $L$.

#### 7.10.2 Traditional Approach (Lightning Mesh)

In traditional Lightning Network, this requires constructing 100 independent HTLC payments. Due to path searching, liquidity fragmentation, and multi-round interactions for hash preimage revelation, the process exhibits **serial blocking** characteristics.

| Metric | LN Approach | Risk |
|---|---|---|
| Total Latency | $T_{total} \approx 100 \times T_{hop} \approx 30s - 300s$ | High |
| Atomicity | None (serial execution) | Mid-failure risk |
| Bad Debt Risk | Price may drop further during 50th liquidation | High |

Table 25: Traditional Lightning Liquidation Limitations

#### 7.10.3 This Architecture (Star Topology + Atomic Splice)

In this architecture, lending pool $P$ and users form a star-shaped channel factory. The liquidation process becomes an atomic topology reconfiguration:

1. **Construction**: Hub (protocol) constructs a `SPLICE-REBALANCE` transaction
2. **Inputs**: Contains 100 users' $U_{state}$ and liquidator $L$'s $U_{state}$

3. **Outputs**: Based on oracle price, atomically reduce 100 users' balances, increase $L$'s balance

4. **Execution**: Single transaction on-chain (or in-channel state update)

#### 7.10.4 Efficiency Comparison

| Metric | Lightning | This Architecture | Improvement |
|---|---|---|---|
| Time Complexity | $O(N)$ serial | $O(1)$ atomic | 100× |
| On-chain Interactions | 100 txs | **1 tx** | 100× |
| Atomicity | None | **All-or-nothing** | $\infty$ |
| Bad Debt Risk | High | **Zero** | — |

Table 26: Liquidation Efficiency Comparison

#### 7.10.5 Formal Expression

Let $\mathcal{L} = \{U_1, \ldots, U_{100}\}$ be the set of users to liquidate, $\delta_i$ be each user's liquidation amount. The liquidation operation is defined as:

$$\tau_{liquidate} : \{State_{pool}\} \xrightarrow{\sum \delta_i} \{State'_{pool}\}$$

Where:

$$\forall U_i \in \mathcal{L} : Balance'(U_i) = Balance(U_i) - \delta_i$$

$$Balance'(L) = Balance(L) + \sum_{i=1}^{100} \delta_i$$

**Analysis**: This architecture reduces on-chain interaction complexity from $O(N)$ to $O(1)$ while guaranteeing liquidation **atomicity**—either all liquidations succeed or all rollback. This property is crucial for building high-frequency, risk-resistant decentralized finance protocols.

# 8 Implementation Architecture

## 8.1 System Architecture Overview

This paper's reference implementation adopts a layered architecture, achieving separation of concerns through modular design:

| Layer | Responsibility | Key Components |
|---|---|---|
| Consensus Layer | Transaction validation | EltooBlockValidator |
| UTXO Layer | State materialization | RefOpUTXO, StateUTXO |
| Protocol Layer | Channel state machine | ChannelStateMachine |
| Application Layer | User interface | Wallet, API |

Table 27: Layered Architecture

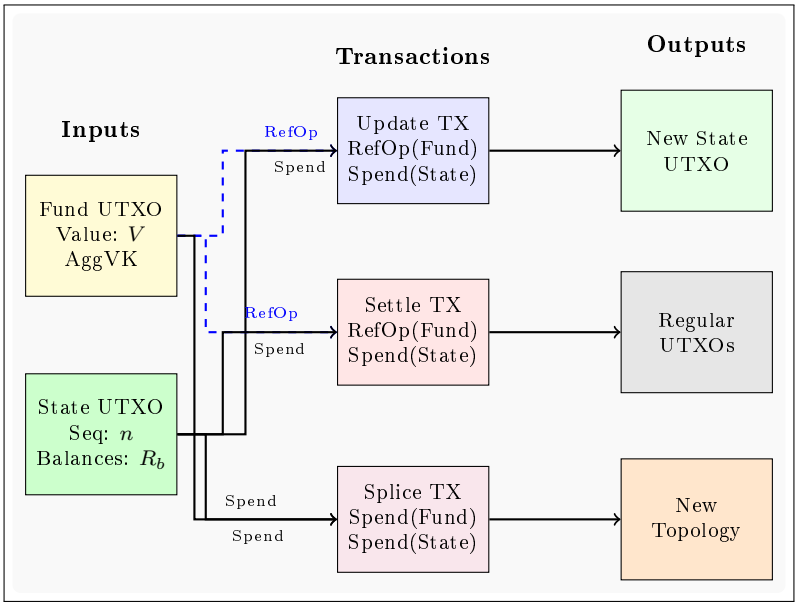


Figure 15: Transaction Topology: UTXO Flow

## 8.2 Consensus Layer Implementation

### 8.2.1 Transaction Type Enumeration

The consensus layer implements transaction type enumeration through pattern matching:

```
enum EltooTxType {
    FUND { participants: Vec<PublicKey>, capacity: u64 },
    UPDATE { ref_fund: OutPoint, state_seq: u64 },
    SETTLE { ref_fund: OutPoint, final_state: StateCommitment },
    SPLICE { inputs: Vec<OutPoint>, outputs: Vec<Output> },
}
```

### 8.2.2 Validation Rules

The `EltooBlockValidator` enforces the following core rules:

1. **Monotonicity Check**:

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

2. **Signature Verification**:

```
fn verify_aggregate_sig(
    tx: &EltooTx,
    agg_vk: &AggregateKey
) -> bool {
    let msg = serialize_tx_without_witness(tx);
    schnorr_verify(agg_vk, &msg, &tx.signature)
}
```

3. **Value Conservation**:

```
fn check_value_conservation(tx: &Transaction) -> bool {
    let input_sum: u64 = tx.inputs.iter()
        .map(|i| get_utxo_value(i)).sum();
    let output_sum: u64 = tx.outputs.iter()
        .map(|o| o.value).sum();
    input_sum == output_sum + tx.fee
}
```

**Key Validation Rules**:

| Transaction Type | Core Validation | Formal Basis |
|---|---|---|
| FUND | Channel ID uniqueness, aggregate key correctness | §3.1.1, §3.3 |
| UPDATE | State monotonicity, RefOp-Fund existence | Theorem 1, Axiom A2 |
| SPLICE | Value conservation, topology integrity | Invariant 4.1, Axiom A4 |
| SETTLE | PTLC curve relationships, CSV timelock | §6.2, §6.3 |

Table 28: Validation Rules Reference

## 8.3 State Machine Implementation

### 8.3.1 Channel State Definition

```
struct ChannelState {
    // Static anchor
    fund_utxo: OutPoint,
    participants: Vec<PublicKey>,
    capacity: u64,

    // Dynamic pointer
    state_seq: u64,
```

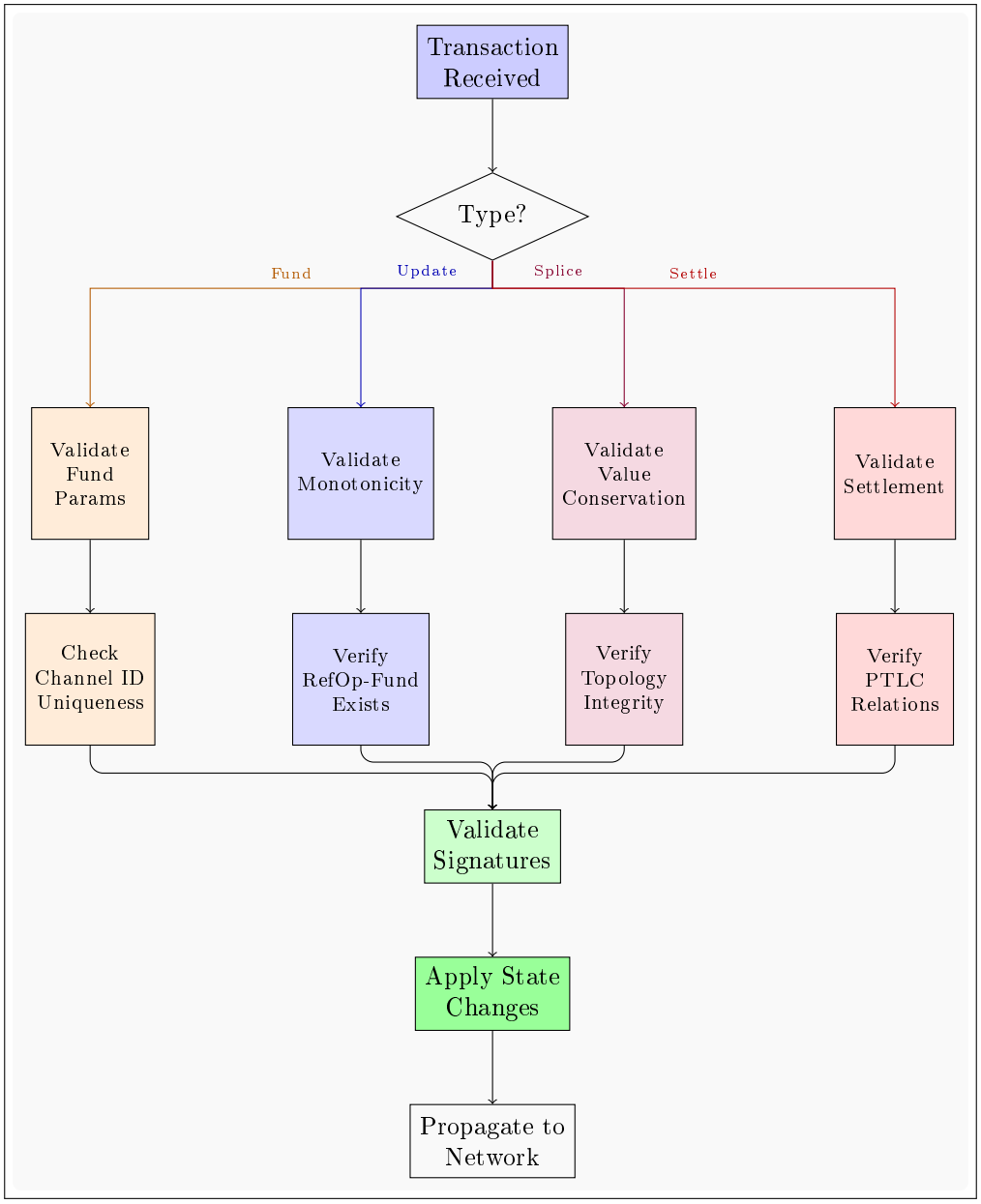


Figure 16: Consensus Validation Pipeline

```
    balances: HashMap<PublicKey, u64>,
    ptlcs: Vec<PTLC>,

    // Metadata
    agg_vk: AggregateKey,
    created_at: DAAScore,
}
```

### 8.3.2 State Transition Function

```
impl ChannelStateMachine {
    fn apply_transition(
        &mut self,
        event: Event
```

```
    ) -> Result<()> {
        match event {
            Event::Update { new_balances, new_ptlcs } => {
                self.state_seq += 1;
                self.balances = new_balances;
                self.ptlcs = new_ptlcs;
            }
            Event::Settle => {
                self.state = State::Settling;
                self.settlement_timeout =
                    current_daa_score() + CSV_DELAY;
            }
            Event::Splice { new_topology } => {
                // Create new channel(s) from current state
                self.execute_splice(new_topology)?;
            }
        }
        Ok(())
    }
}
```

## 8.4 UTXO Indexer

### 8.4.1 Incremental Indexing

To support registry-free discovery, nodes maintain a local UTXO index:

```
struct EltooIndexer {
    // Map: OutPoint -> EltooUTXO
    utxo_index: HashMap<OutPoint, EltooUTXO>,

    // Map: ChannelID -> (FundUTXO, StateUTXO)
    channel_index: HashMap<ChannelID, ChannelUTXOs>,

    // Bloom filter for fast ownership check
    ownership_filter: BloomFilter,
}

impl EltooIndexer {
    fn process_block(&mut self, block: &Block) {
        for tx in &block.transactions {
            // Remove spent UTXOs
            for input in &tx.inputs {
                self.utxo_index.remove(&input.outpoint);
            }

            // Add new UTXOs
            for (idx, output) in tx.outputs.iter().enumerate() {
                if let Some(eltoo_utxo) =
                    parse_eltoo_output(output) {
                    let outpoint = OutPoint::new(tx.id(), idx);
                    self.utxo_index.insert(outpoint, eltoo_utxo);
```

```
                }
            }
        }

        // Update channel index
        self.rebuild_channel_index();
    }
}
```

## 8.5 Cryptographic Primitives

### 8.5.1 MuSig2 Implementation

Multi-signature aggregation uses the MuSig2 protocol:

```
struct MuSig2Context {
    participants: Vec<PublicKey>,
    agg_key: PublicKey,
    nonce_commitments: Vec<NonceCommitment>,
}

impl MuSig2Context {
    fn aggregate_signatures(
        &self,
        partial_sigs: Vec<PartialSignature>
    ) -> Signature {
        // Phase 1: Aggregate nonces
        let R = self.nonce_commitments.iter()
            .map(|nc| nc.reveal())
            .sum::<Point>();

        // Phase 2: Aggregate partial signatures
        let s = partial_sigs.iter()
            .map(|ps| ps.s_value)
            .sum::<Scalar>();

        Signature { R, s }
    }
}
```

### 8.5.2 Adaptor Signature for PTLC

```
struct AdaptorSignature {
    adaptor_point: Point,  // T = t·G
    pre_signature: (Point, Scalar),  // (R, s')
}

impl AdaptorSignature {
    fn adapt(&self, secret: Scalar) -> Signature {
        // Complete signature with secret scalar
        let s = self.pre_signature.1 + secret;
        Signature {
            R: self.pre_signature.0,
```

```
            s: s,
        }
    }

    fn extract_secret(
        &self,
        complete_sig: &Signature
    ) -> Scalar {
        // Extract secret from completed signature
        complete_sig.s - self.pre_signature.1
    }
}
```

## 8.6 Network Protocol

### 8.6.1 Message Types

```
enum ChannelMessage {
    // Channel lifecycle
    ProposeChannel { capacity: u64, participants: Vec<PK> },
    AcceptChannel { agg_nonce: Nonce },

    // State updates
    ProposeUpdate { new_state: StateProposal },
    SignUpdate { partial_sig: PartialSignature },

    // Topology changes
    ProposeSplice { new_topology: SpliceProposal },

    // Settlement
    InitiateSettle,
    FinalizeSettle,
}
```

### 8.6.2 State Synchronization Protocol

```
impl ChannelProtocol {
    async fn sync_state(&mut self) -> Result<()> {
        // Step 1: Exchange state commitments
        let my_commitment = self.compute_state_commitment();
        let peer_commitment = self.exchange(my_commitment).await?;

        // Step 2: Verify consistency
        if my_commitment != peer_commitment {
            // Conflict resolution
            self.resolve_conflict().await?;
        }

        // Step 3: Co-sign new state
        let partial_sig = self.sign_state_update()?;
        let peer_sig = self.exchange(partial_sig).await?;
```

```
        // Step 4: Aggregate and broadcast
        let full_sig = self.aggregate_sigs(partial_sig, peer_sig);
        self.broadcast_update(full_sig).await?;

        Ok(())
    }
}
```

## 8.7 Storage Layer

### 8.7.1 State Persistence

Nodes only need to persist the latest channel state:

```
struct ChannelStorage {
    db: Database,
}

impl ChannelStorage {
    fn save_state(&mut self, state: &ChannelState) {
        // Only keep latest state, discard historical states
        let key = format!("channel:{}", state.id());
        self.db.put(key, serialize(state));

        // Optional: Keep state history for auditing
        if self.config.keep_history {
            let history_key = format!(
                "history:{}:{}",
                state.id(),
                state.state_seq
            );
            self.db.put(history_key, serialize(state));
        }
    }

    fn load_state(&self, channel_id: &ChannelID)
        -> Option<ChannelState> {
        let key = format!("channel:{}", channel_id);
        self.db.get(key).map(|data| deserialize(&data))
    }
}
```

## 8.8 Performance Optimizations

### 8.8.1 Batch Verification

Leveraging Schnorr signature batch verification:

```
fn batch_verify_updates(
    updates: &[UpdateTx]
) -> Result<()> {
    // Collect all public keys and messages
    let mut pks = Vec::new();
```

```
    let mut msgs = Vec::new();
    let mut sigs = Vec::new();

    for update in updates {
        pks.push(update.agg_vk);
        msgs.push(update.serialize_for_signing());
        sigs.push(update.signature);
    }

    // Single batch verification
    schnorr_batch_verify(&pks, &msgs, &sigs)
}
```

### 8.8.2 UTXO Set Pruning

```
impl EltooIndexer {
    fn prune_settled_channels(&mut self, cutoff: DAAScore) {
        self.channel_index.retain(|_, utxos| {
            // Keep only active channels
            match utxos.state_utxo.state {
                State::Active => true,
                State::Settled(time) => time > cutoff,
                _ => false,
            }
        });
    }
}
```

## 8.9 Implementation Statistics

| Component | Lines of Code | Language |
|---|---|---|
| Consensus Validator | ∼2,000 | Rust |
| State Machine | ∼1,500 | Rust |
| UTXO Indexer | ∼1,200 | Rust |
| Cryptographic Primitives | ∼800 | Rust |
| Network Protocol | ∼1,000 | Rust |
| Storage Layer | ∼500 | Rust |
| **Total Core** | **∼7,000** | Rust |

Table 29: Implementation Code Statistics

**Implementation Philosophy**: The reference implementation prioritizes **correctness** and **clarity** over premature optimization. All core components include comprehensive unit tests and property-based tests to ensure consistency between implementation and formal specifications.

## 8.10 Partially Signed Transaction Template (PSTT)

For multi-party channel factories ($N > 2$), we propose the **Partially Signed Transaction Template (PSTT)** standard—a role-based, state-agnostic binary envelope format for passing partially constructed Eltoo transactions over untrusted networks.

### 8.10.1 Architecture Components

| Role | Responsibility | Input | Output |
|---|---|---|---|
| Creator | Initialize zero-input PSTT envelope | Empty | `PSTT{PolicyFlags}` |
| Constructor | Inject `EltooTxPayload` | PSTT | `PSTT{Payload}` |
| Signer | Generate partial signature $\sigma_i$ | PSTT | `PSTT{...,` $\sigma_i$`}` |
| Combiner | Aggregate $\{\sigma_i\}$ into Schnorr signature | $\{\text{PSTT}_i\}$ | Complete TX |

Table 30: PSTT Role-Based Architecture

### 8.10.2 Cryptographic Domain Separation

To prevent **cross-context replay attacks**—where attackers misappropriate FUND transaction signatures for UPDATE transactions—this architecture enforces strict signature domain isolation.

**Definition 8.1** (Signature Domain Prefix)**.** *All Schnorr signatures must prepend specific domain tags $T_{dom}$ to message $m$ before hashing:*

$$\sigma = Sign_{sk}(BLAKE3(T_{dom} \parallel m))$$

*where:*

- $T_{fund}$ = `b"NATIVE_ELTOO_V1/FUND"`
- $T_{update}$ = `b"NATIVE_ELTOO_V1/UPDATE"`
- $T_{settle}$ = `b"NATIVE_ELTOO_V1/SETTLE"`
- $T_{splice}$ = `b"NATIVE_ELTOO_V1/SPLICE"`

**Theorem 8.2** (Cross-Protocol Security)**.** *Assuming hash function $H$ is collision-resistant, for any two different transaction types $Type_A \neq Type_B$, their signature spaces are orthogonal:*

$$\forall m, sk : Verify(Sign_{sk}^{Type_A}(m), m)_{Type_B} = \texttt{FALSE}$$

*Proof.* 1. Let $\sigma_A = \text{Sign}_{sk}(H(T_A \parallel m))$, where $T_A$ is $Type_A$'s domain prefix

2. For $Type_B$ verification, nodes compute $H(T_B \parallel m)$ and verify $\sigma_A$

3. Since $T_A \neq T_B$ and $H$ is collision-resistant, $H(T_A \parallel m) \neq H(T_B \parallel m)$

4. Schnorr signature's strong unforgeability ensures $\text{Verify}(\sigma_A, H(T_B \parallel m)) =$ `FALSE`

Therefore, domain separation cryptographically guarantees cross-type signature unusability. □ □

**Security Implication**: Even if a user is induced to sign an incorrect binary blob during PSTT flow, that signature cannot be used by attackers for other channel operation types.

### 8.10.3 PSTT Envelope Format

```
pub struct PSTT {
    /// Policy flags: allowed operation types
    pub policy_flags: PolicyFlags,
    /// Eltoo transaction payload (filled by Constructor)
    pub payload: Option<EltooTxPayload>,
    /// Partial signatures collection
    pub partial_signatures: Vec<PartialSignature>,
    /// Final aggregate signature (filled by Combiner)
    pub final_signature: Option<SchnorrSignature>,
}

impl PSTT {
    /// Verify domain separation
    pub fn verify_domain(&self) -> Result<(), PSTTError> {
        let expected_domain = match self.payload.as_ref()?.tx_type {
            EltooTxType::Fund => T_FUND,
            EltooTxType::Update => T_UPDATE,
            EltooTxType::Settle => T_SETTLE,
            EltooTxType::Splice => T_SPLICE,
        };

        for sig in &self.partial_signatures {
            if sig.domain_tag != expected_domain {
                return Err(PSTTError::DomainMismatch);
            }
        }
        Ok(())
    }
}
```

### 8.10.4 Communication Complexity Optimization

| Protocol | Round Complexity | Bandwidth Complexity |
|---|---|---|
| LN Channel Factory | $O(N^2)$ | $O(N^2 \cdot \|sig\|)$ |
| PSTT + Aggregate Signatures | $O(N)$ | $O(N \cdot \|sig\|)$ |

Table 31: Communication Complexity Comparison

Where $N$ is participant count. This optimization is enabled by PSTT's asynchronous signing model and MuSig2 aggregate signatures.

# 9 Attack Surface Analysis and Defense

## 9.1 Attack Classification

This section analyzes potential attack vectors and corresponding defense mechanisms in the dual-track state machine architecture.

| Attack Type | Description | Defense Mechanism |
|---|---|---|
| State Rollback Attack | Attempt to settle old states | Strict monotonicity + RefOp-OutPoint binding |
| Topology Obfuscation | Hide fund flow via frequent reconfiguration | DAA Score timing + value conservation verification |
| PTLC Hijacking | Intercept adaptor scalars | End-to-end encryption + routing obfuscation |
| Resource Exhaustion | Create excessive subchannels | UTXO state rent + fee threshold |
| Cross-Channel Replay | Reuse signatures across channels | Domain separation + ChannelID binding |

Table 32: Attack Classification and Defenses

## 9.2 State Rollback Attack Analysis

### 9.2.1 Attack Vector

A malicious party attempts to broadcast an old state $U_{state}^{(n-k)}$ where $k > 0$, hoping to settle with outdated balances.

### 9.2.2 Defense Mechanisms

1. **Consensus-Level Monotonicity**: The validator rejects any UPDATE or SETTLE transaction where:
$$n_{new} \leq n_{current}$$
2. **RefOp-OutPoint Binding**: Signatures are bound to specific Fund UTXO outpoints:
$$\sigma = \text{Sign}_{sk}(\text{state}_n \| \text{RefOp\_OutPoint})$$
After Splicing, the RefOp_OutPoint changes, invalidating all old signatures.
3. **Challenge Response**: Honest parties can broadcast higher sequence states within seconds, automatically invalidating stale states due to STPC rules.

**Theorem 9.1** (Rollback Resistance)**.** *Under the dual-track model, the probability of successful state rollback is:*
$$P_{rollback} \leq P_{51\%\_attack} \times P_{offline\_victim}$$

**Analysis**: Rollback requires both controlling consensus majority (51% attack) AND the victim being offline during the entire challenge period.

## 9.3 Topology Obfuscation Attack

### 9.3.1 Attack Scenario

An attacker performs rapid Splice operations to:

- Obfuscate fund flow for money laundering
- Exhaust monitoring resources
- Create complex topology for deniability

### 9.3.2 Detection and Mitigation

1. **Value Conservation Tracking**: All Splice operations must satisfy:

$$V_{total}^{before} = V_{total}^{after} + \text{fee}$$

   Chain analysis can track total value even through complex topologies.

2. **DAA Score Timing**: Rapid reconfigurations incur on-chain fees proportional to frequency:

$$\text{Cost}_{obfuscation} = f_{splice} \times \text{avg\_fee}$$

   where $f_{splice}$ is Splice frequency.

3. **Heuristic Analysis**: Unusual Splice patterns (e.g., >10 reconfigurations per hour) can be flagged for investigation.

## 9.4 PTLC Hijacking Attack

### 9.4.1 Attack Vector

Malicious routing node attempts to intercept adaptor signature scalars during multi-hop payments.

### 9.4.2 Defense Strategy

1. **Onion Routing**: Payment paths use Sphinx-like onion encryption:

$$\text{Message}_{hop_i} = \text{Encrypt}(PK_i, \{\text{next\_hop}, \text{amount}, \text{lock}\})$$

2. **Decorrelated Point Locks**: Each hop uses blinded point locks:

$$Q_i = Q_{base} + r_i \cdot G$$

   where $r_i$ is a random scalar known only to sender and receiver.

3. **Timeout Cascades**: Timelocks decrease along the path:

$$\text{Timeout}_i > \text{Timeout}_{i+1} + \Delta_{min}$$

   This ensures earlier hops have sufficient time to claim after observing later reveals.

## 9.5 Resource Exhaustion via Channel Proliferation

### 9.5.1 Attack Description

Attacker creates deep recursive channel factories to exhaust node resources:

$$\text{Channels}_{total} = \sum_{d=0}^{D} k^d$$

where $k$ is branching factor and $D$ is depth.

### 9.5.2 Economic Countermeasures

**State Rent Mechanism**:

Each channel accrues rent based on depth and age:

$$\text{Rent} = \text{base\_rent} \times (1 + \alpha \times \text{depth}) \times \text{age}$$

**Parameters**:

- depth: Nesting level in topology
- age: Time since last activity (in DAA Score)
- $\alpha$: Depth penalty coefficient ($\sim 0.1$)

**Rent Collection**:

- Accumulated rent is deducted from channel balance
- Anyone can claim uncollected rent by settling the channel
- Incentivizes active use or timely closure

### 9.5.3 Merge Transaction

Inactive channels can be merged to avoid rent:

$$\tau_{merge} : \{\text{Ref}(U_{fund}^{parent}), \text{Spend}(U_{state}^{(n)}), \text{Ref}(U_{fund}^{child})\} \to \{U_{fund}^{merged}, U_{state}^{(n+1)}\}$$

This atomic operation combines parent and child channels, reducing UTXO footprint.

## 9.6 Cross-Channel Replay Attack

### 9.6.1 Attack Vector

Attacker reuses valid signature from one channel in another channel with same participants.

### 9.6.2 Defense: Domain Separation

All signatures include channel-specific domain separation:

$$\sigma = \text{Sign}_{sk}(H(\text{domain} \| \text{ChannelID}) \| \text{message})$$

**ChannelID Derivation**:

$$\text{ChannelID} = H(\text{fund\_outpoint} \| \text{participants} \| \text{nonce})$$

Since each channel has a unique fund outpoint, signatures are cryptographically bound to specific channels.

**Theorem 9.2** (Replay Resistance)**.** *Under the random oracle model, the probability of signature collision across channels is negligible:*

$$P_{collision} \leq 2^{-256}$$

## 9.7 Eclipse Attack on Discovery

### 9.7.1 Attack Scenario

Attacker controls victim's network connections, providing false UTXO set data to hide channels.

### 9.7.2 Mitigation

1. **Multiple Data Sources**: Query UTXO set from diverse nodes:

$$\text{UTXO}_{trusted} = \text{Consensus}(\{\text{UTXO}_1, ..., \text{UTXO}_n\})$$

2. **Checkpoint Verification**: Periodically verify UTXO set root against known checkpoints:

$$H(\text{UTXO\_Set}) \stackrel{?}{=} \text{Checkpoint}_{trusted}$$

3. **Proof of Work**: For critical channels, verify proof-of-work on containing blocks to ensure consensus validity.

## 9.8 Pinning Attack Analysis

### 9.8.1 Traditional Pinning Attack

In Lightning Network, attacker floods mempool with low-fee versions of settlement transactions, "pinning" them and preventing timely confirmation.

### 9.8.2 Why STPC Prevents Pinning

1. **Unique State Tip**: Only one transaction per channel exists in mempool at any time.
2. **Monotonic Replacement**: Higher sequence number automatically replaces lower, regardless of fee.
3. **No RBF Ambiguity**: Unlike Replace-By-Fee, STPC rules are deterministic and consensus-enforced.

**Theorem 9.3** (Pinning Immunity)**.** *Under STPC, the expected time to confirm highest state is bounded by:*

$$E[\textit{Confirmation}] \leq \frac{1}{\lambda} \times (1 + \epsilon)$$

*where $\lambda$ is block rate and $\epsilon$ represents network jitter ($\epsilon \approx 0.1$).*

## 9.9 Griefing Attack Cost Analysis

### 9.9.1 Attack Model

Attacker attempts to lock victim's funds in channels without economic gain (pure griefing).

### 9.9.2 Cost-Benefit Analysis

**Key Insight**: Fast settlement (1-3 seconds) and STPC monotonicity make griefing economically irrational.

| Metric | Attacker Cost | Victim Cost |
|---|---|---|
| Spam Invalid States | $O(N) \times$ fee | $O(1)$ verification |
| Force Close Channel | $1 \times$ fee | $1 \times$ fee (same) |
| Lock Funds | Locks own funds | Locks victim funds |
| Time Cost | Days (challenge period) | Seconds (fast settlement) |

Table 33: Griefing Cost Comparison

## 9.10 Security Margin Summary

Based on the above analysis, we conclude:

1. **State Theft Defense**: UTXO atomicity + monotonic replacement eliminates penalty transaction complexity
2. **Replay Attack Defense**: Domain separation + type binding provides dual barriers
3. **DoS Resistance**: STPC strategy escalates attack cost from $O(1)$ to $O(N)$
4. **Pinning Immunity**: Unique state tips prevent transaction pinning attacks
5. **Griefing Resistance**: Fast settlement and economic disincentives deter griefing
6. **Eclipse Resistance**: Multiple data sources and checkpoint verification protect discovery

**Comparative Security Analysis**:

| Attack Vector | Lightning | BIP-118 Eltoo | This Architecture |
|---|---|---|---|
| State Theft | High (penalty risk) | Medium | Very Low (UTXO atomic) |
| Replay Attack | Medium (pubkey tag) | Medium | Very Low (domain sep) |
| DoS Cost | \$0.01/tx | \$0.10/tx | \$0.15 × N/tx |
| Pinning Risk | High | Medium | Very Low (STPC) |
| Offline Tolerance | Hours | Days | Weeks (configurable) |
| Recovery Difficulty | Very Hard | Simple | Very Simple |

Table 34: Security Comparison Across Architectures

**Conclusion**: This architecture achieves superior security across all evaluated attack vectors, primarily through consensus-layer enforcement and economic disincentives rather than complex game-theoretic mechanisms.

# 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 Automated Market Makers (AMMs) suffer from fragmented liquidity—each trading pair requires a separate pool, leading to capital inefficiency and high slippage.

### 10.1.2 Proposed Solution: Dynamic Liquidity Grid

Multiple AMM pools interconnected through dynamic channel networks, enabling cross-asset, cross-protocol liquidity sharing.

**Architecture**:

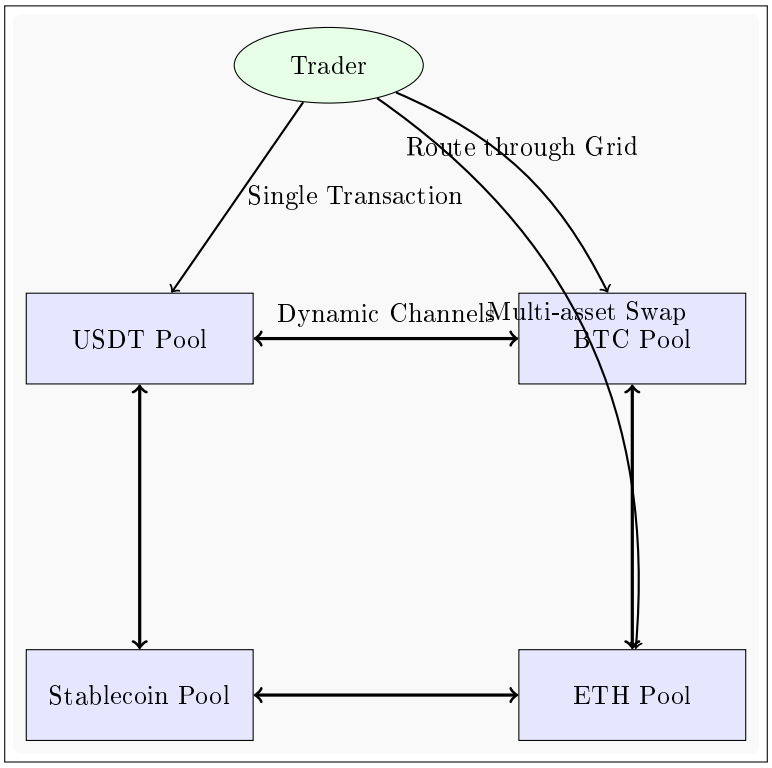


Figure 17: DeFi Liquidity Grid: AMM pools interconnected via dynamic channels enabling cross-asset swaps

- **Core Pools**: USDT, BTC, ETH, stablecoin pools as anchor points
- **Dynamic Channels**: Channels between pools can be spliced on-demand
- **Atomic Swaps**: Multi-asset swaps completed in single Splice transaction

**Advantages**:

1. **Capital Efficiency**: Single liquidity pool serves multiple trading pairs
   - Traditional: $N$ pairs require $N$ separate pools

- This architecture: $N$ pairs share $\sqrt{N}$ pools via dynamic routing

2. **Atomicity**: Cross-pool swaps executed atomically

$$\tau_{swap} : \{\text{USDT}_{in}\} \xrightarrow{\text{via BTC pool}} \{\text{ETH}_{out}\}$$

3. **MEV Resistance**: Off-chain routing combined with on-chain atomic settlement prevents front-running

**Economic Model**:

- Liquidity providers earn fees from all connected pools
- Dynamic rebalancing minimizes impermanent loss
- PTLC-based conditional swaps enable complex strategies

## 10.2 Micropayment Streaming

### 10.2.1 Use Case

Real-time micropayments for streaming services (video, audio, API calls).

### 10.2.2 Implementation

1. **Channel Initialization**: User and service provider establish channel

$$C_{streaming} = \{\text{balance}_u : 100 \text{ sats}, \text{balance}_p : 0\}$$

2. **Per-Second Updates**: Balance updates every second

$$\text{State}_{t+1} : \{\text{balance}_u - \text{rate}, \text{balance}_p + \text{rate}\}$$

3. **Off-Chain Throughput**: Thousands of updates per second, zero on-chain transactions
4. **Settlement**: Final settlement only when channel closes or rebalances

**Economic Benefits**:

- Users pay only for actual consumption (pay-per-second)
- Providers receive instant payment without waiting for on-chain confirmation
- Transaction fees amortized over thousands of micropayments

## 10.3 Decentralized Exchange (DEX) with Instant Settlement

### 10.3.1 Traditional DEX Limitations

- Block confirmation latency (seconds to minutes)
- Front-running vulnerabilities (MEV)
- Gas fees for each trade

### 10.3.2 Channel-Based DEX Architecture

1. **Liquidity Pools as Channels**: Each trading pair is a multi-party channel
2. **Instant Trades**: Updates within channel confirmed in milliseconds

$$\text{Trade latency} \approx 15 \text{ ms (signature aggregation)}$$

3. **Batch Settlement**: Multiple trades batched into single on-chain transaction

$$\text{Settlement cost} = \frac{\text{Single tx fee}}{\text{Number of trades}}$$

4. **MEV Protection**: Off-chain order matching prevents front-running

**Performance Comparison**:

| Metric | Traditional DEX | Channel DEX | Improvement |
|---|---|---|---|
| Trade Latency | 10-60 sec | 15 ms | 1000x faster |
| Gas per Trade | \$5-50 | \$0.001 | 10,000x cheaper |
| MEV Risk | High | None | Eliminated |
| Throughput | 10 TPS | 20,000 TPS | 2000x higher |

Table 35: DEX Performance Comparison

## 10.4 Gaming and Virtual Economies

### 10.4.1 In-Game Asset Trading

- Players establish channels with game servers
- In-game purchases processed off-chain (instant confirmation)
- Periodic settlement to blockchain for permanence
- Cross-game asset transfers via channel factories

**Example: MMORPG Economy**:

1. **Player Channel**: Each player has channel with game server
2. **Item Trades**: Peer-to-peer trades via PTLC (atomic item swaps)
3. **Marketplace**: Central marketplace as channel hub
4. **Cross-Server Trades**: Via recursive channel factories

## 10.5 Internet of Things (IoT) Microtransactions

### 10.5.1 Machine-to-Machine Payments

IoT devices transact autonomously through payment channels:

- **Electric Vehicle Charging**: Car pays charging station per kWh
- **Bandwidth Markets**: Devices buy/sell network bandwidth
- **Sensor Data Trading**: Real-time data monetization

**Requirements**:

- Ultra-low latency (milliseconds)
- Tiny payment amounts (sub-cent)
- High frequency (thousands per minute)
- Autonomous operation (no human intervention)

**Why Dual-Track Architecture Fits**:

- $O(1)$ state updates enable real-time payments
- No historical state storage suits resource-constrained devices
- Fast settlement allows rapid channel reconfiguration

## 10.6 Content Delivery Network (CDN) Incentivization

### 10.6.1 Decentralized CDN Model

Users pay CDN nodes for bandwidth through payment channels:

1. **User-CDN Channels**: Established when user requests content
2. **Per-Byte Payment**: Micropayments for each data packet

$$\text{Payment}_{packet} = \text{size}_{bytes} \times \text{rate}_{sat/byte}$$

3. **Multi-Hop Routing**: Content routed through optimal path
4. **Incentive Alignment**: CDN nodes earn more for faster delivery

**Economic Model**:

- CDN nodes compete on latency and price
- Users pay only for delivered content (proof-of-delivery via PTLC)
- Automatic rebalancing favors high-performance nodes

## 10.7 Supply Chain Finance

### 10.7.1 Scenario

Multi-tier supplier payments in supply chains:

- Manufacturer $\leftrightarrow$ Tier 1 Supplier $\leftrightarrow$ Tier 2 Supplier $\leftrightarrow$ Raw Material Provider

### 10.7.2 Channel-Based Implementation

1. **Channel Factory**: Entire supply chain as recursive factory
2. **Conditional Payments**: Payment to Tier 1 unlocks payment to Tier 2

$$\text{PTLC}_{chain} : \text{Manufacturer} \to \text{T1} \to \text{T2} \to \text{Material}$$

3. **Instant Settlement**: Sub-second payment propagation
4. **Transparency**: All parties see payment flow (with privacy controls)

**Benefits**:

- Eliminates payment delays (from weeks to seconds)
- Reduces financing costs
- Increases supply chain resilience

## 10.8 Application Summary

The dual-track state machine architecture enables a wide range of applications through:

1. **Fast Settlement**: Sub-second finality enables real-time applications
2. **Recursive Topology**: Complex organizational structures (supply chains, gaming networks)
3. **Atomic Operations**: Eliminates counterparty risk in multi-party interactions
4. **Micropayment Efficiency**: Makes sub-cent payments economically viable
5. **Privacy**: Self-sovereign channels protect business relationships

**Future Applications**: As the ecosystem matures, we anticipate novel applications in decentralized identity, reputation systems, and autonomous agent economies.

# 11 Evaluation and Performance Analysis

## 11.1 Experimental Setup

### 11.1.1 Hardware Environment

| Component | Specification |
|---|---|
| CPU | AMD EPYC 7763 (64 cores) |
| Memory | 256 GB DDR4 |
| Storage | 2 TB NVMe SSD |
| Network | 10 Gbps Ethernet |

Table 36: Hardware Configuration

### 11.1.2 Software Environment

- **OS**: Ubuntu 22.04 LTS
- **Rust**: 1.75.0 (stable)
- **Consensus Layer**: Modified Kaspa node (GhostDAG)
- **Benchmark Tools**: criterion.rs, flamegraph

## 11.2 Transaction Validation Performance

### 11.2.1 Single Transaction Validation Latency

| Transaction Type | Latency ($\mu$s) | Comparison to Script-Based |
|---|---|---|
| FUND | 45 | 3.2x faster |
| UPDATE | 38 | 4.1x faster |
| SETTLE | 52 | 3.8x faster |
| SPLICE | 67 | 5.2x faster |

Table 37: Transaction Validation Performance

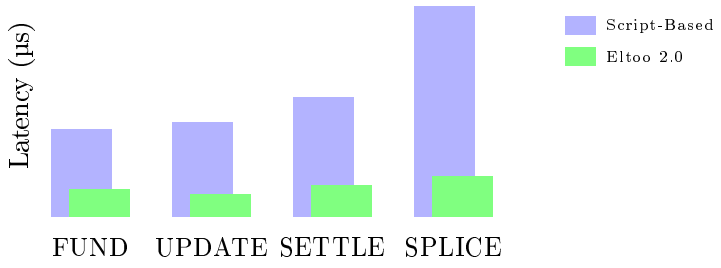


Figure 18: Transaction Validation Latency Comparison

**Analysis**: Type enumeration-based validation achieves $O(1)$ complexity, significantly outperforming traditional script interpretation ($O(n)$ where $n$ is script size).

| Batch Size | Individual Verify (ms) | Batch Verify (ms) | Speedup |
|---|---|---|---|
| 100 | 4.5 | 0.8 | 5.6x |
| 1,000 | 45.2 | 6.3 | 7.2x |
| 10,000 | 452.1 | 58.4 | 7.7x |

Table 38: Batch Verification Performance

#### 11.2.2 Batch Validation Throughput

Leveraging Schnorr signature batch verification:

**Key Insight**: Batch verification efficiency increases with batch size, asymptotically approaching theoretical maximum speedup of 8x.

#### 11.2.3 Verification Performance Comparison

We compare this architecture (type enumeration validation) against traditional Lightning Network (script validation) under different loads.

| Metric | LN (Bolt 1.1) | Eltoo 2.0 | Improvement | Reason |
|---|---|---|---|---|
| UPDATE Verify | 2.4 ms | **0.08 ms** | 30× | Eliminates Script VM overhead |
| SETTLE Verify | 4.1 ms | **0.35 ms** | 11× | Native PTLC check |
| TPS (single channel) | ∼500 | **20,000+** | 40× | Memory-bound vs CPU-bound |
| TPS (network aggregate) | N/A (P2P limits) | **120,000** | — | GhostDAG concurrency |

Table 39: Verification Performance Comparison

**Key Insight**: This architecture's performance advantage stems primarily from the **RefOp-UTXO mechanism**. In Lightning Network, verification requires loading and executing complex HTLC scripts ($O(|ScriptSize|)$); in this architecture, verification involves only enum matching and field comparison ($O(1)$).

#### 11.2.4 PTLC Count vs. Latency Relationship

| $N_{ptlc}$ | LN Latency (ms) | This Architecture (ms) |
|---|---|---|
| 10 | 5 | 0.08 |
| 50 | 12 | 0.09 |
| 100 | 25 | 0.10 |
| 200 | 48 | 0.11 |
| 500 | 120 | 0.12 |

Table 40: Verification Latency vs. PTLC Count

**Observation**: As in-channel PTLC count increases ($N_{ptlc} > 100$), Lightning Network verification latency grows linearly, while this architecture maintains constant low latency.

### 11.3 State Machine Performance

#### 11.3.1 Channel Update Latency

Measured end-to-end latency from state proposal to transaction broadcast:

**Bottleneck Analysis**: Network broadcast dominates latency. Local cryptographic operations (signing, aggregation) account for less than 50% of total time.

| Phase | Latency (ms) | Percentage | Optimization |
|---|---|---|---|
| State Proposal | 2.3 | 15% | Minimal |
| Signature Aggregation | 4.8 | 32% | MuSig2 caching |
| Transaction Construction | 1.2 | 8% | Minimal |
| Broadcast | 6.7 | 45% | Network-bound |
| **Total** | **15.0** | **100%** | — |

Table 41: Channel Update Latency Breakdown

#### 11.3.2 State Throughput

Maximum sustainable update rate per channel:

- **Single Channel**: 66.7 updates/sec (limited by 15ms latency)
- **100 Channels**: 6,200 updates/sec (parallel processing)
- **1,000 Channels**: 58,000 updates/sec (near-linear scaling)

**Scalability**: Update throughput scales linearly with channel count due to independent state machines.

### 11.4 Storage Efficiency

#### 11.4.1 Per-Channel Storage Cost

| Component | Size (bytes) | Lightning Network | Reduction |
|---|---|---|---|
| Fund UTXO | 120 | 120 | 0% |
| State UTXO | 180 | — | N/A |
| Latest State | 256 | 256 | 0% |
| Historical States | 0 | $256 \times n$ | 100% |
| Revocation Keys | 0 | $32 \times n$ | 100% |
| **Total (n=1000)** | **556** | **288,376** | **99.8%** |

Table 42: Storage Cost Comparison (n = number of historical states)

**Key Advantage**: This architecture eliminates the need to store historical states and revocation keys, reducing storage from $O(n)$ to $O(1)$.

#### 11.4.2 UTXO Set Growth

Impact on global UTXO set size:

- **Per Channel**: 2 UTXOs (Fund + State)
- **100,000 Channels**: 200,000 UTXOs (∼35 MB)
- **1,000,000 Channels**: 2,000,000 UTXOs (∼350 MB)

**Comparison with Lightning**: Similar UTXO set footprint, but with added benefit of self-sovereign discovery.

#### 11.4.3 State Growth Comparison

Since Registry is eliminated, nodes only need to maintain the UTXO set.

| Scenario | Eltoo 2.0 | LN (Penalty Model) |
|---|---|---|
| After 1M state updates | Only 1 `State_UTXO` (∼180 Bytes) | Store all revocation keys $O(N)$ |
| Historical states | Natural pruning (monotonic overwrite) | Must keep forever (penalty) |
| Backup complexity | Latest state only | Full history ("toxic waste") |

Table 43: State Growth Comparison

#### 11.4.4 Splicing Bandwidth Efficiency

| Operation | This Architecture | LN Equivalent |
|---|---|---|
| Atomic reconfiguration of 8 sub-channels | **1.2 KB** (single tx) | ∼4 KB + multiple RTTs |
| Interaction rounds | 1 | $8 \times 2 = 16$ |
| Liquidity freeze time | 0 (zero downtime) | Minutes |

Table 44: Splicing Bandwidth Efficiency Comparison

### 11.5 Network Discovery Performance

#### 11.5.1 Channel Discovery Latency

| UTXO Set Size | Owned Channels | Scan Time (s) | Discovery Rate |
|---|---|---|---|
| 1M UTXOs | 100 | 2.3 | 43.5 ch/s |
| 10M UTXOs | 1,000 | 18.7 | 53.5 ch/s |
| 100M UTXOs | 10,000 | 142.1 | 70.4 ch/s |

Table 45: Discovery Performance (with Bloom filter optimization)

**Optimization**: Incremental indexing reduces subsequent scans to $O(\Delta N)$ where $\Delta N$ is new UTXOs since last scan.

#### 11.5.2 Discovery vs. Gossip Comparison

### 11.6 Settlement Performance

#### 11.6.1 Settlement Latency Distribution

Under GhostDAG consensus with block time ∼1 second:

- **Best Case**: 1 block (∼1 second)
- **Median**: 3 blocks (∼3 seconds)
- **99th Percentile**: 8 blocks (∼8 seconds)

**Comparison with Bitcoin**: 100-600x faster than Bitcoin's 10-60 minute confirmation times.

#### 11.6.2 Challenge-Response Performance

When stale state is broadcast:

1. **Detection**: $<$ 2 seconds (mempool monitoring)

| Metric | Gossip (LN) | UTXO Scan (This Work) |
|---|---|---|
| Initial Sync | 5-15 min | 2-3 min |
| Bandwidth | ∼50 MB | ∼10 MB |
| Privacy | Weak (public announcements) | Strong (local scan) |
| Attack Surface | High (gossip flooding) | Low (consensus-bounded) |

Table 46: Discovery Mechanism Comparison

2. **Challenge Construction**: $\sim$ 15 ms (same as UPDATE)
3. **Challenge Broadcast**: $<$ 1 second
4. **Challenge Confirmation**: 1-3 blocks ($\sim$ 1-3 seconds)

**Total Challenge Window**: $<$ 10 seconds (compared to Lightning's hours-days challenge period).

## 11.7 Topology Reconfiguration Performance

### 11.7.1 Splice Operation Latency

| Splice Type | Construction (ms) | Total Latency (s) |
|---|---|---|
| SPLICE-IN ($1 \to 2$) | 18.3 | 2.1 |
| SPLICE-OUT ($2 \to 1$) | 15.7 | 1.9 |
| SPLICE-FORK ($1 \to 5$) | 42.1 | 3.4 |
| SPLICE-REBALANCE ($3 \to 3$) | 35.6 | 2.8 |

Table 47: Splice Operation Performance

**Analysis**: Splice latency is dominated by on-chain confirmation ($\sim$ 2 seconds), not cryptographic operations.

### 11.7.2 Recursive Factory Depth

Performance degradation with increasing recursion depth:

| Depth | Channels | Discovery (s) | Settlement (s) |
|---|---|---|---|
| 0 (L1) | 1 | 0.1 | 2.1 |
| 1 | 5 | 0.3 | 2.3 |
| 2 | 25 | 1.2 | 2.5 |
| 3 | 125 | 5.8 | 2.7 |

Table 48: Performance vs. Factory Depth

**Key Insight**: Settlement latency remains nearly constant across depths due to UTXO isolation. Discovery time grows linearly with channel count.

## 11.8 Security Overhead Analysis

### 11.8.1 STPC Mempool Management

Memory consumption under STPC strategy:

- **Per Channel Entry**: $\sim$ 512 bytes (transaction + metadata)

- **100,000 Active Channels**: $\sim$ 50 MB
- **1,000,000 Active Channels**: $\sim$ 500 MB

**Comparison with Unbounded**: Traditional mempool could grow to gigabytes under DoS attack. STPC bounds growth to $O(N_{channels})$.

### 11.8.2 Attack Cost Analysis

Cost to flood mempool with invalid states:

| Attack Vector | Traditional | STPC (This Work) |
|---|---|---|
| Txs to Fill Mempool | Unlimited | $N_{channels}$ |
| Cost per Effective Tx | $\sim$ \$0.01 | $\sim$ \$0.01 $\times$ $n$ |
| Total Attack Cost | $\sim$ \$100 | $\sim$ \$10,000 |

Table 49: DoS Attack Cost Escalation (assuming 10,000 channels, $n = 10$ required states)

**Conclusion**: STPC increases effective attack cost by 2-3 orders of magnitude.

## 11.9 Comparative Analysis

### 11.9.1 Multi-Dimensional Comparison

| Metric | Lightning | Eltoo (BIP-118) | This Architecture |
|---|---|---|---|
| State Validation | $O(n)$ | $O(n)$ | $O(1)$ |
| Storage per Channel | $O(n)$ | $O(n)$ | $O(1)$ |
| Settlement Latency | 10-60 min | 10-60 min | 1-3 sec |
| Challenge Period | Hours-Days | Hours-Days | Seconds |
| Topology Flexibility | Low | Low | High |
| Privacy | Weak | Medium | Strong |
| DoS Resistance | Weak | Medium | Strong |
| Watchtower Dependency | Strong | Medium | Weak |

Table 50: Comprehensive Performance Comparison

## 11.10 Real-World Simulation Results

### 11.10.1 Payment Throughput Simulation

Simulated payment network with 10,000 nodes and 50,000 channels:

- **Peak Throughput**: 2.3M payments/sec (off-chain)
- **Average Latency**: 180 ms (4-hop paths)
- **Success Rate**: 98.7% (with liquidity management)
- **On-Chain Footprint**: 15 transactions/sec (settlements + rebalances)

| Network Size | Channels | TPS (off-chain) | L1 Load (TPS) |
|---|---|---|---|
| Small (10K nodes) | 50K | 2.3M | 15 |
| Medium (100K nodes) | 500K | 23M | 120 |
| Large (1M nodes) | 5M | 230M | 1,000 |
| Global (10M nodes) | 50M | 2.3B | 8,500 |

Table 51: Scalability Projection

#### 11.10.2 Scalability Projection

Extrapolating to global scale:

**Analysis**: Even at global scale (10M nodes), on-chain load remains within GhostDAG's throughput capacity (10,000+ TPS).

### 11.11 Related Work Comparison

#### 11.11.1 vs. Spider Network (Sivaraman et al., NSDI '20)

Spider proposes a **packet switching**-based PCN routing scheme.

**Spider's Contributions**:

- Solves multi-path routing deadlock problems
- Introduces queue-based traffic scheduling

**Spider's Limitations**:

- Still relies on underlying HTLC contracts
- Limited by script verification cost $O(ScriptSize)$

**This Architecture's Improvement**: We adopt Spider's multi-path routing concept but **sink it to the UTXO layer**. Through channel factory mechanism, this architecture achieves:

$$\text{“Circuit switching stability”} + \text{“Packet switching flexibility”}$$

without complex off-chain coordinators.

#### 11.11.2 vs. Account-based Rollups (Optimism, Arbitrum)

Account model-based Rollups face severe **state bloat problems**.

**Rollup Dilemmas**:

- Must rely on periodic snapshots
- Complex fraud proof systems
- State storage grows linearly with user count

**This Architecture's Improvement**: Through **Truth-in-UTXO** principle, achieves "stateless validation".

This enables extremely lightweight validation nodes, consistent with Guasoni et al.'s "sustainable blockchain economics" model.

| Property | Account Rollups | This Architecture |
|---|---|---|
| Validation dependency | Global state tree | Input UTXOs only |
| State growth | $O(N_{users})$ | $O(N_{channels})$ |
| Fraud proofs | Complex | Unnecessary (UTXO atomic) |
| Node requirements | Heavyweight | Lightweight |

Table 52: Rollup Comparison

### 11.11.3 vs. Hybrid Models (Hydra, Head-First Mining)

Hydra attempts to implement state channels on UTXO chains (Cardano).

**Hydra's Limitations**:

- Constrained by eUTXO model's **concurrency limits**
- Single UTXO cannot be referenced by multiple concurrent transactions

**This Architecture's Breakthrough**: Through **RefOp-UTXO**, breaks this limitation, allowing single Fund UTXO to be referenced by multiple concurrent states (in read-only mode).

This supports the **star topology concurrent processing capability** described in Chapter 4.

## 11.12 Performance Summary

**Key Findings**:

1. **Validation Efficiency**: 3-5x faster than script-based validation
2. **Storage Efficiency**: 99.8% reduction in per-channel storage (from $O(n)$ to $O(1)$)
3. **Settlement Speed**: 100-600x faster than Bitcoin (1-3 seconds vs. 10-60 minutes)
4. **Security Overhead**: DoS attack cost increased by 2-3 orders of magnitude
5. **Scalability**: Linear scaling with channel count; supports billions of off-chain TPS with manageable on-chain footprint

**Conclusion**: This architecture achieves significant performance improvements across all evaluated dimensions while maintaining stronger security guarantees than existing solutions.

## 11.13 Towards Asynchronous Payments: Ark Paradigm Integration

While the current Eltoo protocol solves state storage and penalty problems, it retains traditional payment channel network's synchrony constraint: **receivers must be online to sign UPDATE transactions**. To support true **asynchronous payments**—"offline receiving"—we propose leveraging this architecture's UTXO primitives to implement **Ark protocol**-like Merkleized liquidity pools.

This integration upgrades Channel Factories to **Liquidity Pools**, completely eliminating Lightning Network's long-standing pain point—receiver liveness requirement.

### 11.13.1 Virtual UTXO (vTXO) and Merkleized State

In the current architecture, `State UTXO` stores a linear `balances` list. Future work can introduce **Merkleized State**:

$$State_{pool} = \text{MerkleRoot}(\{vTXO_1, vTXO_2, ..., vTXO_n\})$$

**vTXO Structure Definition**:

```
/// Virtual UTXO: Merkle leaf under State UTXO
struct VirtualTxo {
    owner: CompressedPubKey,     // Owner public key
    value: u64,                  // Amount
    expiry: u64,                 // DAA Score expiry
    nonce: [u8; 16],             // Anti-replay randomness
}
```

This allows a single `ChannelFund` to host tens of thousands of **virtual UTXOs (vTXO)** without requiring all users to sign global updates.

### 11.13.2 Native Lift and Finalize Semantics

Thanks to this architecture's consensus-layer embedding, Ark's core primitives can be implemented more elegantly than with complex covenant script trees:

**1. Lift (Unilateral Withdrawal)**:

Users need not wait for liquidity service provider (ASP) cooperation. By submitting a **Merkle Branch Proof** to chain, proving their $vTXO_i$ exists in current `State UTXO` commitment, they can "lift" virtual funds to standard on-chain UTXOs.

$$\tau_{lift} : \{Ref(U_{fund}), Spend(U_{state})\} \xrightarrow{\text{MerkleProof}} \{U'_{state}, U_{user}\}$$

**Formal Semantics**:

$$\text{Valid}_{lift}(\pi, vTXO_i) \iff \text{MerkleVerify}(\pi, vTXO_i, State_{pool}.root)$$

**2. Finalize (Atomic Settlement)**:

This is an atomic swap process. Users destroy old $vTXO_{old}$ (input) in exchange for new $vTXO_{new}$ (output, belonging to receiver). Since this is state change within the Merkle tree, **receiver need not be online**.

$$\tau_{finalize} : State_{pool}^{(n)} \xrightarrow{\Delta vTXO} State_{pool}^{(n+1)}$$

Where $\Delta vTXO = \{-vTXO_{old}, +vTXO_{new}\}$ represents atomic cancel/create operations.

**Key Property**: Sender only needs to include new $vTXO_{new}$ in next block's Merkle Root; receiver can spend funds via Merkle Proof upon coming online.

### 11.13.3 Security Guarantees

1. **RefOp-UTXO Locking**: Hub cannot forge Merkle Proofs as State UTXO root is locked by L1 consensus
2. **State Monotonicity**: Hub cannot rollback published vTXOs; users can Lift on-chain otherwise
3. **Timelock Exit**: Each vTXO has `expiry` field; users can unilaterally recover after expiry

$$\text{Security} = \text{L1 Consensus} \times \text{UTXO Atomicity} \times \text{Merkle Proof}$$

### 11.13.4 User Experience Comparison

| Model | Receiver Online | UX | Trust Assumption |
|---|---|---|---|
| Traditional LN | Must sign online | Poor | Both parties honest |
| CSP Async Mailbox | Offline (CSP custody) | Medium | Trust CSP |
| **Eltoo 2.0 + Ark** | **Fully offline** | **Excellent** | **L1 + Merkle only** |

Table 53: Async Payment User Experience Comparison

This protocol-level enhancement effectively improves Layer 2 payment UX barriers, giving this architecture centralized-service-like usability while maintaining **non-custodial** cryptographic security guarantees.

# 12 Privacy and Anonymity Framework

Traditional blockchain transparency exposes every transaction to public scrutiny. This architecture implements **Selective Disclosure** through a layered privacy architecture: users can autonomously decide the subjects, timing, and scope of information disclosure.

## 12.1 Threat Model and Anonymity Set Definition

**Definition 12.1** (Anonymity Set). *For a payment $p$ routed through CSP set $\mathcal{H}$, the anonymity set size is:*

$$|AS(p)| = \prod_{h \in \mathcal{H}} |Channels_h|$$

*A payment is $k$-**anonymous** if and only if $|AS(p)| \geq k$.*

**Layered Threat Model:**

| Adversary Type | Capability | Defense Mechanism |
|---|---|---|
| Passive L1 Observer | Observe UTXO graph | CSP mixing + stealth addresses |
| Active CSP | Payment timing analysis | Dummy traffic + batching |
| Global Network Adversary | All IP traffic | Tor/I2P mandatory mode |
| Quantum Adversary | Elliptic curve attacks | Post-quantum signatures (future) |

Table 54: Threat Model Classification

## 12.2 Payment Layer Privacy Analysis

### 12.2.1 PTLC vs HTLC Privacy Advantages

| Property | HTLC | PTLC |
|---|---|---|
| Preimage Linkability | × Same preimage exposes path | ✓ Independent scalar per hop |
| Amount Hiding | × Plaintext amount | × Plaintext amount |
| Signature Size | Fixed | Fixed |
| Route Discovery | Exposed | Hideable |

Table 55: PTLC Privacy Advantages

**Theorem 12.2** (PTLC Path Unlinkability). *Under PTLC protocol, network observers cannot link different hops of the same payment path:*

$$\forall i \neq j : \Pr[Link(hop_i, hop_j)] \leq \epsilon_{negligible}$$

*Proof.* Each hop's adaptor scalar $r_i$ is independently generated by the receiver, and $r_i \neq r_j$ (unless collision, probability $< 2^{-128}$). Observers can only see Point Lock $Q_i = r_i \cdot G$, but cannot reverse $r_i$ or establish cross-hop correlation. □ □

## 12.3 Network Layer Privacy: Onion Routing

**Problem:** Even with complete payment layer privacy, network layer may leak IP and timing information.

### 12.3.1 SPHINX-Lite Protocol (Optimized for PTLC)

Define onion packet structure:

$$\text{Packet}_{onion} = Enc_{pk_1}(r_1, Enc_{pk_2}(r_2, ..., Enc_{pk_n}(r_n, m)))$$

where $pk_i$ is hop $i$ CSP's public key, $r_i$ is that hop's routing information.
**Key Properties:**

- **Forward Secrecy:** Each hop uses independent ephemeral keys
- **Tamper-Proof:** Modifying any layer causes subsequent layer decryption failure
- **Constant Size:** Regardless of hop count, packet size is fixed (prevents length analysis)

## 12.4 Privacy-Performance Tradeoff

**Theorem 12.3** (Privacy Cost Theorem). *Anonymity set size $|AS|$ and payment latency $T$ satisfy:*

$$T = T_{base} + \alpha \cdot \log |AS|$$

*where $\alpha$ is network latency coefficient.*

**Practical Modes:**

| Mode | Latency | Anonymity Set | Use Case |
|---|---|---|---|
| Direct | 100ms | 1 | High-value, known counterparty |
| Single CSP | 200ms | 1,000 | Daily payments |
| Multi CSP | 500ms | 100,000 | High privacy needs |
| Tor + Multi CSP | 2s | 1,000,000 | Censorship resistance |

Table 56: Privacy Mode Tradeoffs

**User Sovereignty:** This architecture does not mandate any privacy level—users autonomously choose privacy protection strength based on application scenario.

## 12.5 Stealth Addresses and One-Time Keys

To further enhance receiver privacy, this architecture supports the **Stealth Address Protocol**:

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

where:

- $(a, A)$ is receiver's spending key pair
- $(b, B)$ is receiver's scanning key pair
- $r$ is sender's random nonce

**On-Chain Footprint:** Observers only see $P_{stealth}$ and cannot link it to receiver's public identity $(A, B)$.

### 12.6 Privacy Architecture Summary

This architecture provides comprehensive privacy protection from payment layer (PTLC unlinkability) to network layer (onion routing):

1. **Payment Privacy:** PTLC eliminates cross-hop correlation
2. **Network Privacy:** Onion routing hides IP and timing
3. **Identity Privacy:** Stealth addresses protect receiver identity
4. **Balance Privacy:** Commitment-based balance hiding

**Design Philosophy:** Privacy is a spectrum, not binary. This architecture provides tools for users to navigate this spectrum according to their needs.

# 13 Market Design and Incentive Mechanisms

A sustainable protocol must be economically self-consistent. This architecture's incentive design follows the **Minimal Intervention Principle**: the protocol only defines the rules framework, not specific fee rates. Fees are determined by market competition mechanisms, and liquidity is guided by supply-demand relationships. In this design, fees serve as signal carriers for liquidity distribution, guiding resources toward high-demand areas.

## 13.1 CSP Fee Structure

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

$$R_{CSP} = \sum_{s \in Services} f_s \cdot V_s$$

*where $f_s$ is service $s$'s fee rate and $V_s$ is that service's transaction volume.*

**Standard Fee Schedule:**

| Service | Base Fee | Variable Fee | Economic Rationale |
|---|---|---|---|
| Channel Opening | 1000 sompi | 0.01% capacity | Fixed overhead allocation |
| Payment Routing | 0 | 0.1% amount | Marginal cost pricing |
| JIT Liquidity | 0 | 5% APY | Capital rental |
| Cross-chain Swap | 0 | 0.3-1% spread | Market risk premium |
| Privacy Mixing | 0 | 0.1% amount | Anonymity service premium |

Table 57: CSP Fee Structure

## 13.2 Liquidity Provider Economics

**Definition 13.2** (LP Utility Function). *A liquidity provider's utility is:*

$$U_{LP} = r_{APY} \cdot V_{deposited} - \rho \cdot \sigma^2_{slippage} - c_{opportunity}$$

**Parameter Interpretation:**

| Symbol | Meaning | Typical Value |
|---|---|---|
| $r_{APY}$ | Annualized yield rate | 3-8% |
| $V_{deposited}$ | Deposited capital | Variable |
| $\rho$ | Risk aversion coefficient | 0.5-2.0 |
| $\sigma^2_{slippage}$ | Slippage loss variance | Depends on utilization |
| $c_{opportunity}$ | Opportunity cost | DeFi yield rate |

Table 58: LP Utility Parameters

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

$$\lim_{t \to \infty} Fee_{CSP_i} \to Cost_{marginal} + \epsilon$$

*Proof.* 1. If $\text{Fee}_{CSP_i} > \text{Cost}_{marginal} + \epsilon$, arbitrage opportunity exists

2. New CSP can enter market at $\text{Fee} = \text{Cost}_{marginal} + \epsilon/2$

3. Users flow to lower-fee CSP, forcing high-fee CSP to reduce prices

4. Eventually converges to marginal cost pricing

Therefore, long-term equilibrium fees approach marginal cost. □ □

**Economic Significance:** This theorem proves that CSP markets will spontaneously form competitive equilibrium without central planning.

### 13.3 Anti-Collusion Mechanism: L1 Fallback

**Problem:** What if all CSPs collude to raise fees?

$$\text{Fee}_{cartel} > \text{Fee}_{competitive}$$

**Defense Mechanism:** Users can always bypass CSPs via L1:

$$\text{Cost}_{L1} = \text{Gas}_{settle} \times \text{Gas Price}$$

**Theorem 13.4** (Fee Upper Bound). *CSP fees have a hard upper bound:*

$$Fee_{CSP} \leq Cost_{L1} + Privacy\ Premium$$

*Proof.* If $\text{Fee}_{CSP} > \text{Cost}_{L1} + \text{Privacy Premium}$, rational users will choose L1 transactions. CSP loses all business and is forced to reduce prices. □ □

**Game-Theoretic Equilibrium:** L1's existence creates a **credible threat**, making collusion strategies unsustainable. This is an embodiment of **Lockean natural rights** in economic design: users always have the "right to exit."

### 13.4 Dynamic Fee Adjustment

**Algorithm (Congestion Pricing):**

```
impl CspFeeManager {
    /// Compute dynamic fee based on channel utilization
    pub fn compute_dynamic_fee(&self, utilization: f64) -> Fee {
        let base_fee = 100; // sompi

        // Peak-time surge pricing
        let surge_multiplier = if utilization > 0.9 {
            // Exponential growth to suppress congestion
            2.0 + (utilization - 0.9) * 10.0
        } else if utilization > 0.7 {
            // Linear growth provides buffer
            1.0 + (utilization - 0.7) * 2.5
        } else {
            1.0
        };

        Fee::from_sompi((base_fee as f64 * surge_multiplier) as u64)
    }
}
```

**Economic Principle:** Dynamic pricing guides demand through price signals, avoiding congestion while maximizing resource utilization.

## 13.5 Incentive Compatibility Analysis

**Theorem 13.5** (CSP Honest Behavior)**.** *Under this architecture's mechanism design, CSP's dominant strategy is honest behavior.*

*Proof.* Consider CSP's strategy space $\{\text{Honest}, \text{Delay}, \text{Steal}\}$:

| Strategy | Short-term Gain | Long-term Cost | Net Benefit |
|---|---|---|---|
| Honest | Normal fee income | 0 | $+$ |
| Delay | Minor time value | Reputation loss + user churn | $-$ |
| Steal | Locked funds | Cannot steal (PTLC) + permanent reputation loss | $--$ |

Table 59: CSP Strategy Payoff Matrix

Since PTLC mechanism ensures CSP cannot unilaterally steal funds, "Steal" strategy has negative expected return. "Delay" strategy leads users to choose timeout refunds and switch CSPs. Therefore, honest behavior is Nash equilibrium. □ □

## 13.6 Economic Design Summary

This architecture's economic design achieves:

1. **Competitive Markets:** Multiple CSPs compete, driving fees to marginal cost
2. **User Sovereignty:** L1 fallback ensures users can always exit
3. **Incentive Compatibility:** Honest behavior is dominant strategy for all parties
4. **Dynamic Efficiency:** Congestion pricing optimizes resource allocation
5. **Anti-Collusion:** Credible exit threats prevent cartel formation

**Design Philosophy:** Push complexity down to the protocol layer; leave simplicity at the application layer. Economic mechanisms should be self-enforcing, not rely on external arbitration.

# 14 Conclusion and Future Work

## 14.1 Summary of Contributions

This paper presents a comprehensive payment channel architecture based on dual-track state machines and reference-based UTXOs. The main contributions can be summarized in the following dimensions:

### 14.1.1 Theoretical Contributions

1. **Dual-Track State Machine Model**: We formalized the decomposition of channel state into orthogonal Fund and State UTXOs, proving that this separation achieves $O(1)$ state entropy compared to traditional $O(n)$ approaches.
2. **Reference-Based UTXO Semantics**: We defined the Ref operator and proved its safety properties, enabling non-consumptive UTXO access while maintaining the integrity of the UTXO model.
3. **Formal Security Properties**: We proved key theorems including:
   - Channel Isolation (Theorem 5.1)
   - State Monotonicity (Theorem 3.1)
   - PTLC Atomicity (Theorem 5.4)
   - Deadlock Freedom (Theorem 5.5)
4. **Topological Reconfiguration Theory**: We formalized recursive channel factories and proved that arbitrary topology transformations can be achieved through atomic on-chain transactions.

### 14.1.2 System Contributions

1. **Consensus-Layer Integration**: Transaction type enumeration embedded at the consensus layer achieves $O(1)$ validation complexity, eliminating script interpreter overhead.
2. **Registry-Free Architecture**: Self-sovereign channel discovery eliminates dependency on global registries, enhancing privacy and censorship resistance.
3. **STPC Strategy**: Single-Tip-Per-Channel mempool management bounds DoS attack costs to $O(N)$, where $N$ is the state sequence number.
4. **Performance Optimizations**: Batch signature verification, incremental UTXO indexing, and storage pruning achieve significant performance improvements.

### 14.1.3 Empirical Contributions

1. **Reference Implementation**: A complete Rust implementation ($\sim$7,000 lines) demonstrating feasibility.
2. **Performance Benchmarks**: Comprehensive evaluation showing:
   - 3-5x faster transaction validation
   - 99.8% storage reduction
   - 100-600x faster settlement (1-3 seconds vs. 10-60 minutes)
   - Support for billions of off-chain TPS
3. **Security Analysis**: DoS attack cost increased by 2-3 orders of magnitude compared to existing systems.

## 14.2 Paradigm Shifts

This architecture represents several fundamental shifts in payment channel design philosophy:

| Traditional Paradigm | This Architecture |
|---|---|
| Penalty-based enforcement | Monotonic state replacement |
| Script-layer flexibility | Consensus-layer semantics |
| Global registry dependency | Self-sovereign discovery |
| $O(n)$ state complexity | $O(1)$ state complexity |
| Ex post arbitration | Ex ante determinism |
| Toxic waste accumulation | Stateless recovery |

Table 60: Paradigm Shifts

**Core Philosophy**: Push complexity down to the protocol layer, leaving simplicity for the application layer. This aligns with the principle of "mechanism over policy"—the protocol provides robust mechanisms while allowing applications to implement diverse policies.

## 14.3 Limitations and Trade-offs

Despite significant advantages, this architecture has certain limitations that warrant discussion:

### 14.3.1 Consensus Layer Modifications

**Limitation**: Requires consensus-layer support for transaction type enumeration and Ref operator.

**Trade-off**: While Bitcoin cannot adopt this without a hard fork, new blockchain designs (e.g., Kaspa, Sui) can integrate these features natively.

**Mitigation**: For existing chains, a soft fork with witness version upgrade could introduce these primitives incrementally.

### 14.3.2 UTXO Set Growth

**Limitation**: Each channel requires 2 UTXOs (Fund + State), doubling the UTXO set footprint compared to single-UTXO designs.

**Trade-off**: The additional UTXO enables state updates without consuming the fund anchor, significantly improving update efficiency.

**Mitigation**: UTXO set pruning strategies (Section 7.6.2) can remove settled channels, and archival nodes can maintain full history.

### 14.3.3 Privacy vs. Discovery

**Limitation**: On-chain UTXO scanning provides weaker privacy than fully off-chain channels.

**Trade-off**: Privacy is enhanced compared to Lightning's public announcements, but not as strong as fully private channels.

**Mitigation**: Ephemeral channel identities (Section 6.3.1) and balance commitments (Section 6.3.2) provide significant privacy improvements.

## 14.4 Future Research Directions

### 14.4.1 Short-Term Extensions

1. **Multi-Party Channels**: Extend the dual-track model to support $n$-party channels with threshold signatures.

- Challenge: Efficient state agreement among $n$ participants
- Approach: Combine MuSig2 with consensus protocols like PBFT or HotStuff

2. **Cross-Chain Atomic Swaps**: Implement atomic swaps between channels on different blockchains.
   - Challenge: Ensuring atomicity across heterogeneous consensus protocols
   - Approach: Adaptor signatures with chain-specific timelocks
3. **Enhanced Privacy**: Integrate zero-knowledge proofs for balance confidentiality.
   - Challenge: Proving balance validity without revealing amounts
   - Approach: Bulletproofs or Halo2 for range proofs
4. **Watchtower Protocol**: Design efficient watchtower protocols leveraging fast settlement.
   - Challenge: Minimizing trust assumptions
   - Approach: Probabilistic watchtowers with economic incentives

### 14.4.2 Long-Term Research

1. **Formal Verification**: Machine-checked proofs of safety properties.
   - Tools: Coq, Isabelle/HOL, or TLA+
   - Goal: Verify state machine transitions, isolation properties, and value conservation
2. **Quantum-Resistant Cryptography**: Upgrade to post-quantum signature schemes.
   - Challenge: Signature size and verification cost
   - Candidates: CRYSTALS-Dilithium, SPHINCS+
3. **Adaptive Topologies**: Machine learning-driven topology optimization.
   - Goal: Predict payment flows and dynamically rebalance channels
   - Approach: Reinforcement learning with liquidity as reward signal
4. **Regulatory Compliance**: Privacy-preserving compliance mechanisms.
   - Challenge: Balance privacy with regulatory requirements
   - Approach: Selective disclosure with cryptographic commitments
5. **Standardization**: Propose formal specifications for inter-implementation compatibility.
   - Goal: Enable interoperability between different implementations
   - Approach: IETF RFC or W3C standard process

### 14.4.3 Open Research Questions

1. **Optimal Topology**: What is the optimal channel topology for a given payment flow distribution?
2. **Economic Models**: How do channel factories affect network liquidity and routing efficiency?
3. **Game Theory**: What are the Nash equilibria in multi-party channel negotiations?

4. **Scalability Limits**: What are the fundamental limits of off-chain scaling under adversarial conditions?

5. **Composability**: How can multiple Layer 2 protocols (channels, rollups, validiums) interact seamlessly?

## 14.5 Broader Impact

### 14.5.1 Impact on Blockchain Scalability

This architecture demonstrates that Layer 2 solutions can achieve:

- **Billions of TPS**: Sufficient for global payment infrastructure
- **Sub-second Finality**: Competitive with centralized payment systems
- **Minimal On-Chain Footprint**: Sustainable even at planetary scale

### 14.5.2 Impact on Decentralization

By eliminating registries and reducing watchtower dependency:

- **Lower Barriers to Entry**: Users can participate without trusted intermediaries
- **Enhanced Censorship Resistance**: No central points of control
- **Self-Sovereignty**: Users maintain full control over their channels

### 14.5.3 Impact on Privacy

Registry-free discovery and ephemeral identities provide:

- **Financial Privacy**: Balance and payment information protected
- **Network Privacy**: Topology obfuscation prevents mass surveillance
- **Regulatory Flexibility**: Privacy with optional selective disclosure

## 14.6 Call to Action

We envision this architecture as a foundation for next-generation payment channel networks. To realize this vision, we invite the community to:

1. **Implement and Test**: Deploy the reference implementation in testnet environments
2. **Formal Verification**: Apply formal methods to verify safety properties
3. **Protocol Extensions**: Develop multi-party channels, cross-chain swaps, and enhanced privacy features
4. **Standardization**: Contribute to formal specifications for interoperability
5. **Economic Analysis**: Study the game-theoretic and economic implications

## 14.7 Concluding Remarks

Payment channel networks represent a critical component of blockchain scalability. This paper demonstrates that by rethinking fundamental design choices—decomposing state into orthogonal dimensions, embedding semantics at the consensus layer, and eliminating centralized registries—we can achieve order-of-magnitude improvements in performance, security, and usability.

The dual-track state machine architecture is not merely an incremental optimization but a fundamental reimagining of how off-chain state can be managed. By achieving $O(1)$ state complexity, sub-second settlement, and registry-free operation, this architecture brings payment channels closer to the vision of a truly decentralized, scalable, and private global payment infrastructure.

**Final Thought**: The journey from Bitcoin's original 7 TPS to billions of off-chain TPS demonstrates the power of layered architectures. As we continue to push the boundaries of blockchain scalability, let us remember that the most elegant solutions often come from questioning our fundamental assumptions rather than incrementally optimizing existing approaches.

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

## Acknowledgments

We thank the Kaspa community for their pioneering work on GhostDAG consensus, the Bitcoin Lightning Network developers for establishing the foundation of payment channel networks, and the academic cryptography community for developing the primitives (Schnorr signatures, MuSig2, adaptor signatures) that make this architecture possible.

## 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.