A Bounded Verification Architecture for Dual-Ledger Systems

A verification-first framework for offline-resilient, edge-native participation

Abstract

This paper presents a theoretical architecture for verification under resource bounds in dual-ledger distributed systems—architectures that separate parallel transaction processing (often as a block-lattice or DAG) from sequential commitment ordering (typically a blockchain), enabling independent verification of canonical ordering without reconstructing full execution traces. We establish requirements for verifiers operating under network partition, storage constraints (O(log N)), and limited computation. The architecture employs genesis-anchored trust roots, bounded inclusion proofs, and explicit refusal semantics as foundational primitives. We introduce two novel contributions: (1) *adaptive retention horizons* that dynamically adjust based on verifier query patterns, and (2) an optional *privacy-enhanced verifier variant* using succinct proofs for state frontiers where confidentiality is required. Treating verification as architecturally primary, we invert traditional blockchain assumptions and provide concrete algorithms with quantitative bounds. The central question: what verification properties must hold if resource-bounded, offline-resilient edge verifiers are to participate meaningfully in dual-ledger systems?

1. Purpose and Scope

1.1 What This Paper Contributes

This work synthesizes exploratory research into a unified verification architecture for dual-ledger systems while introducing two novel primitives: adaptive retention horizons and optional privacy-enhanced verifier variants. We provide concrete algorithms, quantitative bounds, and practical implementation guidance while maintaining theoretical rigor.

1.2 Explicit Non-Goals

This architecture does not address: (1) Byzantine agreement under arbitrary network conditions; (2) Producer-level censorship resistance; (3) Data availability beyond retention horizons; (4) Global finality proofs for bounded verifiers; (5) Sybil resistance; (6) Economic incentive alignment. These require orthogonal protocol layers.

2. System Model

2.1 Dual-Ledger Architecture

A dual-ledger system maintains two structurally distinct ledgers: a *transaction ledger* (block-lattice or DAG) for parallel state transitions, and a *commitment ledger* (blockchain) for canonical ordering. This decouples execution from commitment, enabling verification of ordering without reconstructing execution traces.

Figure 1 (Conceptual): Transaction Ledger (parallel account chains) → Commitment Ledger (sequential blocks with Merkle roots). Each commitment C_i contains: prev_hash(C_{i-1}), merkle_root(transactions), producer_signatures, height *i*.

2.2 Producers Versus Verifiers

Producers: Generate commitments, maintain full state (typically ≥100GB), participate in consensus. **Verifiers**: Validate commitments with bounded resources (≤10MB storage, intermittent connectivity). Verifiers operate under strict constraints: storage O(log N) for N commitments, bandwidth O(log M) for M transactions per commitment, computation feasible in browsers using standard Web Crypto API primitives.

3. Genesis-Anchored Trust Root

3.1 Genesis as Trust Primitive

The genesis commitment G_0 is the irreducible trust foundation. Its hash $H(G_0)$ serves as the network identifier. Lineage verification confirms: $\forall\ C_i, \exists$ path $\{C_0=G_0, C_1, ..., C_i\}$ where $H(C_j)$ is embedded in C_{j+1}. This establishes chain identity but not validity—a verifier knows "this chain descends from G_0" but not "this chain represents valid execution.".

4. Bounded Verification Primitives

Architectural Foundation: This architecture does not require zero-knowledge proofs or privacy-preserving cryptography as foundational primitives. The core model relies on deterministic, bounded verification of proof-carrying state transitions without re-execution. Privacy-enhancing techniques such as zero-knowledge proofs may be composed where confidentiality is desired, but they are not architecturally necessary for the verification guarantees established herein.

4.1 Header-Only Verification

Verifiers synchronize commitment headers only. Storage: $O(k \cdot \log N)$ where k is header size (~500 bytes), N is chain length. For $N=10^6$ commitments: ~500MB if storing all, or ~5MB with checkpointing every 100 blocks.

4.2 Bounded Inclusion Proofs

Merkle proofs demonstrate transaction T_x inclusion in commitment C_i. Proof size: $O(\log M)$ where M = transactions per commitment. For $M=10^4$: ~320 bytes (SHA256, 10 hashes). Algorithm:

```
def verify_inclusion(tx_hash, proof, merkle_root):
    current = tx_hash
    for (sibling_hash, direction) in proof:
        if direction == 'left':
            current = hash(sibling_hash + current)
        else:
            current = hash(current + sibling_hash)
    return current == merkle_root
```

4.3 Adaptive Retention Horizons (Novel)

Traditional retention horizons are static (e.g., "keep proofs for 30 days"). We introduce *adaptive horizons* that adjust based on verifier query patterns. Let Q(t) = query frequency at time t, S_{max} = storage bound. The adaptive horizon $H_{adaptive}$ satisfies:

$$H_{adaptive}(t) = \max\{ h : \Sigma_{i=t-h}^{t} [Q(i) \cdot \text{proof\_size}(i)] \leq S_{max} \}$$

High-query accounts receive longer retention. Implementation uses exponentially-weighted moving average: $Q(t) = \alpha \cdot Q_{current} + (1-\alpha) \cdot Q(t-1)$, $\alpha=0.3$. This enables efficient resource allocation—frequently queried data persists longer without manual configuration. For example, with 1MB storage: ~1000 proofs for high-frequency accounts, ~100 for low-frequency.

4.4 Refusal Semantics: Consolidated

When verification fails, return explicit refusal codes: EXPIRED (proof beyond horizon), OUT_OF_SCOPE (query outside frontier), INSUFFICIENT_BANDWIDTH, UNKNOWN. This prevents silent degradation. Refusal is architecturally correct—it signals honest operation under constraints.

5. Verifier Lifecycle

5.1 Bootstrapping Algorithm

Pseudocode for initial sync:

```
    def bootstrap(genesis_hash, peer_network):
        headers = fetch_headers(peer_network, start=0, end=latest)
        if hash(headers[0]) != genesis_hash:
            return REJECT("Invalid genesis")

        for i in range(1, len(headers)):
            if headers[i].prev_hash != hash(headers[i-1]):
                return REJECT("Broken lineage at height", i)

        frontier = headers[-1].height
        proofs = fetch_proofs_for_accounts(watched_accounts, frontier)
        return SUCCESS(frontier, proofs)
```

Cost: O(N) header downloads (~500N bytes), O(log M) proofs per watched account. For $N=10^5$, 10 accounts: ~50MB initial sync.

5.2 Proof Refresh and Expiration

Proofs expire when frontier - proof_height > retention_horizon. Adaptive horizons (Section 4.3) reduce refresh frequency for low-priority accounts. Upon expiration, verifier regresses frontier or requests new proofs. Pull-based model: verifiers control their security boundaries.

Figure 2 (Conceptual): Lifecycle flowchart: [Bootstrap from Genesis] → [Verify Headers] → [Fetch Proofs] → [Track Frontier] → [Refresh Loop] → {Proofs Valid? Yes: Continue | No: Refuse/Refresh}.

6. Composition Under Constraints

6.1 Composability

Primitives compose if costs remain bounded: O(log N) + O(log M) = O(log(N·M)). Header verification + inclusion proofs + adaptive horizons compose because none introduce multiplicative overhead.

6.2 The Impossibility Theorem

Theorem: A verifier with storage S cannot verify history length N if proof_size(N) > S. **Proof:** Inclusion proofs for N commitments require at least $\log_2(N)$ hashes per proof. For k proofs: space ≥ k · $\log_2(N)$ · hash_size. As N → ∞, space → ∞. Therefore, bounded verifiers accept finite scope. ■

Quantitative Example: With S=100KB, hash_size=32 bytes: max verifiable N ≈ 2^{20} ≈ 10^6 commitments (assuming 10 proofs). Beyond this, verifiers must employ checkpointing or accept trust assumptions.

7. Human Interface as Security Boundary

7.1 Epistemic State Surfaces

The user interface is part of the security and trust boundary—not merely user experience design—because verification without comprehensible presentation fails to enable informed human judgment. The UI must display distinct states: **VERIFIED** (green, within frontier), **UNVERIFIED** (yellow, received but unconfirmed), **EXPIRED** (gray, beyond horizon), **REFUSED** (red, out of scope), **UNKNOWN** (blank). Scope metadata is mandatory: "Balance as of height H, frontier F, horizon R.".

Figure 3 (Conceptual UI Mockup): [Account Balance: 100 ZNN | Status: VERIFIED (green) | Height: 1,234,567 | Frontier: 1,234,600 | Horizon: 30 days | Refresh: Available].

8. Optional Privacy-Enhanced Verifier Variant

8.1 Motivation

Standard bounded verification uses Merkle proofs (O(log M) size) with full transparency. Where confidentiality is required, succinct cryptographic proofs could compress state frontier verification to O(1) size while hiding execution details. We propose an *optional privacy-enhanced variant* where producers generate succinct proofs of state frontier correctness using zero-knowledge techniques.

8.2 Construction

Let S_i = state at commitment C_i. Producer generates succinct proof: π_i = Prove(S_{i-1}, transactions, S_i, "S_i is valid transition"). Verifier checks: Verify(π_i, C_i.state_root) $\rightarrow$ {Accept, Reject}. Proof size: ~200 bytes (using schemes like Groth16), verification: ~5ms.

8.3 Trade-offs and Composability

Advantages: O(1) proof size, constant verification time, transaction privacy. **Disadvantages:** High producer overhead (seconds per proof), trusted setup for some schemes, reduced transparency (proof hides execution details). **Critical clarification:** Privacy-enhancing proof usage is entirely optional and compositional—it does not weaken the bounded verification guarantees established in prior sections but instead trades execution transparency for constant proof size and confidentiality. **Hybrid approach:** Use succinct proofs for state frontiers where privacy is desired, Merkle proofs for individual transaction inclusion where transparency is preferred—balancing efficiency, transparency, and privacy requirements.

8.4 Implementation Frameworks

This could integrate with: (1) **Cosmos SDK**: Add IBC light client with optional succinct state proofs; (2) **Substrate**: Custom pallet for proof verification; (3) **Browser environments**: WASM-compiled verifier using proof system libraries. No timeline claims—purely architectural feasibility.

9. Architectural Limitations

This architecture cannot provide: (1) **Global finality proofs** for bounded verifiers—requires observing producer quorums; (2) **Censorship resistance** without additional mechanisms; (3) **Universal data availability** beyond retention horizons; (4) **Fork choice** for verifiers—lineage verification confirms identity, not consensus legitimacy.

10. Related Work

10.1 Bitcoin SPV and Extensions

Nakamoto's SPV (2008) introduced header-only verification. BIP-157/158 Neutrino (2017) added compact block filters. Our work extends SPV to dual-ledger architectures where transaction and commitment structures are separated, and introduces adaptive retention (Section 4.3) absent in SPV designs.

10.2 FlyClient and Probabilistic Sampling

FlyClient (2020) achieves O(log log N) proof complexity via probabilistic sampling. Our adaptive horizons could compose with FlyClient's sampling for long-range verification. Key difference: FlyClient targets single-ledger PoW chains; we address dual-ledger systems with explicit refusal semantics.

10.3 Mina and Recursive SNARKs

Mina (2020) compresses entire chain state to ~22KB using recursive zero-knowledge proofs. Our optional privacy-enhanced variant (Section 8) is less aggressive—O(1) proofs for state frontiers only, not full recursion. Trade-off: Mina requires substantial producer computation (~minutes per block); our hybrid approach limits succinct proofs to frontiers where privacy is desired, preserving transparency for individual transactions.

10.4 Ethereum Light Clients and Data Availability

Ethereum's Nimbus (2021) and recent data availability sampling work (Celestia, 2023) address similar problems. Celestia's DAS enables light clients to verify data availability probabilistically. Our architecture could integrate DAS for transaction ledger verification while maintaining bounded commitment ledger verification—orthogonal concerns that compose.

10.5 Rollups and Layer-2 Systems

Optimistic (Arbitrum, 2021) and validity rollups (zkSync, StarkNet, 2022) separate execution from settlement. Our dual-ledger model parallels this but operates at Layer-1. A rollup could employ bounded verification for its own state—verifying rollup commitments using our primitives without trusting the L1 entirely.

11. Conclusion

11.1 Contributions

This paper synthesizes bounded verification principles and introduces two novel primitives: (1) adaptive retention horizons for efficient resource allocation, and (2) optional privacy-enhanced verifier variants for O(1) state frontier proofs where confidentiality is desired. We provide concrete algorithms, quantitative bounds (O(log N) storage, O(log M) proofs), and implementation guidance. The architecture inverts traditional blockchain priorities—verification is foundational, execution is secondary.

11.2 Honest Constraints Over Illusions

Verification under resource constraints is inherently limited. This architecture chooses explicit refusal over silent degradation, honest boundaries over convenient approximations. If verification is to mean anything, it must acknowledge what cannot be verified as clearly as what can.

11.3 Future Work

Several directions remain: (1) **Game-theoretic incentives** for proof refresh—how do producers price proof generation? (2) **Integration with DAS**—combining data availability sampling with bounded verification. (3) **Adaptive horizon optimization**—machine learning models to predict query patterns more accurately. (4) **Formal verification**—mechanized proofs of impossibility theorem and composition properties. (5) **Implementation studies**—empirical evaluation in Cosmos SDK or Substrate without claiming production readiness.

References

[1] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.
[2] Corallo, M., et al. (2017). BIP-157/158: Compact block filters for light clients.
[3] Bünz, B., et al. (2020). FlyClient: Super-light clients for cryptocurrencies. IEEE S&P.
[4] Mina Protocol. (2020). Mina: A succinct blockchain using recursive zk-SNARKs.
[5] Status Research. (2021). Nimbus: Ethereum light client implementation.
[6] Celestia Labs. (2023). Data availability sampling for modular blockchains.
[7] Offchain Labs. (2021). Arbitrum: Optimistic rollup protocol.
[8] Matter Labs. (2022). zkSync 2.0: ZK rollup architecture.
[9] Gabizon, A., et al. (2019). PLONK: Permutations over Lagrange bases for SNARKs.
[10] Parity Technologies. (2023). Substrate: Modular blockchain framework.
[11] Cosmos Network. (2023). IBC light client verification specification.
[12] Wood, G. (2014). Ethereum: A secure decentralized transaction ledger.
[13] Buterin, V., et al. (2023). EIP-4844: Proto-danksharding for data availability.
[14] Ben-Sasson, E., et al. (2018). Scalable, transparent SNARKs via polynomial commitments.
[15] Bunz, B., et al. (2021). Proof-carrying data from accumulation schemes.