Bounded Inclusion Without Merkle Trees: Deterministic Verification Under Header-Only Constraints

Anonymous Author(s)

December 2025

Abstract

Most blockchain systems rely on Merkle inclusion proofs to demonstrate that a transaction or state element exists within a block. While cryptographically sound, this approach implicitly assumes that verifiers can process unbounded proof sizes and that state inclusion must be proven per element. This paper presents an alternative model: *bounded inclusion without Merkle trees*, where correctness is derived from deterministic state transitions and header-committed execution bounds, rather than per-transaction inclusion proofs. We show that, under specific architectural constraints, inclusion can be inferred from header-level commitments alone, enabling lightweight verification suitable for browser-native and resource-constrained environments. This approach shifts verification from transaction enumeration to invariant preservation, reducing verification complexity from O(log |T|) per transaction to O(1) per user interaction while maintaining security guarantees through deterministic execution and bounded resources, enabling scalable light clients without compromising correctness.

Keywords: Blockchain verification, Merkle trees, light clients, deterministic execution, state commitments, bounded resources

1. Introduction

Modern blockchain systems face a fundamental tension between decentralization and verifiability. Full nodes maintain complete transaction history and state, enabling independent verification of all network activity. Light clients, conversely, verify only block headers and rely on inclusion proofs for specific transactions of interest. This asymmetry creates a verification gap that limits the practical deployment of blockchain applications in resource-constrained environments.

The dominant approach to bridging this gap employs Merkle tree inclusion proofs [1]. When a light client needs to verify that transaction t was included in block B_i, it requests a proof path from the transaction to the Merkle root committed in the block header. While elegant, this solution carries several limitations:

(1) Proof size grows logarithmically with block size, creating bandwidth overhead

(2) Verifiers must process and validate intermediate hash computations

(3) Each transaction requires a separate proof, precluding batch verification

(4) The model assumes all verifiers have access to proof generation infrastructure

This paper proposes an alternative verification paradigm that eliminates per-transaction inclusion proofs entirely. Instead, we demonstrate that under specific architectural constraints—namely deterministic state transitions and header-committed execution bounds— transaction inclusion can be inferred from header-level commitments alone. This approach trades enumeration of included transactions for verification of execution invariants, enabling dramatically simpler light client implementations while preserving security guarantees.

Scope and non-goals: This model addresses verification correctness—ensuring that included transactions produce valid state transitions within resource bounds. It does not attempt to address fairness (transaction ordering), censorship resistance (ensuring transaction inclusion), or ordering optimality (MEV mitigation). These concerns require separate mechanisms orthogonal to the verification model presented here. A separate hostile technical review accompanies this work. The review attempts multiple adversarial attacks against the bounded inclusion model; all succeed under relaxed assumptions, as expected. The review serves to bound applicability rather than refute correctness.

2. Problem Statement and Traditional Approach

We begin by formalizing the traditional Merkle-based inclusion model and identifying its computational requirements.

2.1 Blockchain Structure

Let a blockchain produce a sequence of blocks $B_0, B_1, ..., B_n$. Each block B_i traditionally contains:

```
• A list of transactions T_i = {t_1, t_2, ..., t_k} • A Merkle root
M_i = Merkle(T_i) • A previous block hash H_{i-1} • Metadata
(timestamp, nonce, etc.)
```

2.2 Traditional Inclusion Verification

A verifier checking inclusion of transaction $t \in T_i$ requires:

```
• O(log |T_i|) proof size (hash path from leaf to root) • Knowledge of
intermediate hash values • Computational cost proportional to proof
depth • Bandwidth proportional to log |T_i|
```

This model does not scale efficiently to browser-native or header-only clients. For a block containing 10,000 transactions, a single inclusion proof requires approximately 14 hash values ($\log_2(10{,}000) \approx 13.29$). For clients verifying multiple transactions, this overhead compounds linearly.

2.3 Limitations of Existing Solutions

Several approaches attempt to optimize Merkle-based verification:

• **Verkle trees [2]:** Reduce proof size through vector commitments but still require per-transaction proofs

• **Accumulators [3]:** Enable batch verification but add computational overhead for maintenance

• **STARK/SNARK proofs [4]:** Compress verification but require trusted setup or intensive proof generation

None of these approaches fundamentally eliminate the need for per-transaction proof verification. We seek a model where inclusion verification requires only header-level information.

3. Deterministic State Transition Model

We reformulate blockchain verification as validation of deterministic state transitions rather than enumeration of included transactions.

3.1 State Transition Function

Model the system as a pure state transition function:

$$S_{i+1} = \delta(S_i, T_i)$$

Where: • S_i is the global state after block i • T_i is the set of transactions in block i • δ is the deterministic state transition function • δ is total (defined for all valid inputs)

3.2 Determinism Assumption

Assumption A1 (Deterministic Execution): For any valid state S_i and admissible transaction set T_i, the resulting state $S_{i+1} = \delta(S_i, T_i)$ is unique. **Assumption A1' (Canonical Ordering):** Either (a) state transitions are order-independent (commutative operations), OR (b) there exists a deterministic canonical ordering function ord(T_i) that validators must follow, derivable from S_i and T_i without external randomness.

These assumptions require that the execution environment exhibits no non-determinism: no random number generation, no dependence on external state, no concurrent execution races. Assumption A1' is critical because transaction ordering affects execution when transactions conflict (e.g., both modify the same account). Without canonical ordering, an adversary could claim different transaction orderings produce the committed state, enabling transaction substitution attacks. In practice, most blockchains enforce canonical ordering via nonce sequences (Ethereum), UTXO ordering (Bitcoin), or account-chain sequencing (Zenon).

3.3 Execution Bounds

To ensure finite execution time, we introduce resource bounds β_i that constrain the computational cost of transactions in block *i*:

$$\beta_i = (\text{gas\_limit}, \text{memory\_limit}, \text{storage\_ops}, \ldots) \quad \text{Valid iff: } \text{Cost}(T_i) \leq \beta_i \text{ (component-wise)}$$

These bounds serve dual purposes: they prevent denial-of-service attacks through resource exhaustion and enable predictable verification costs. Crucially, bounds are committed in the block header, making violations detectable without executing transactions.

Important limitation: Light clients can verify that the *aggregate* execution cost satisfies $\text{Cost}(T_i) \leq \beta_i$ through the header commitment, but cannot independently verify individual transaction costs without full execution. This means a light client verifying transaction t can confirm that *some valid execution within* β_i *produced the observed state*, but cannot prove that t specifically consumed c gas units without executing it. For applications requiring precise gas accounting, light clients must either trust

validator attestations or use conservative worst-case estimates.

4. Header-Committed State Invariants

We now define the minimal information that block headers must commit to for bounded inclusion verification.

4.1 Header Structure

Each block header commits to a triple:

```
H_i = (R_i, β_i, σ_i) Where: • R_i = h(S_i) is the state root (cryptographic commitment) • β_i are execution resource bounds • σ_i is a validator quorum signature
```

4.2 Header Validity Condition

A header H_i is valid if and only if:

$$\exists\ T_i : \delta(S_{i-1}, T_i) = S_i \wedge \text{Cost}(T_i) \leq \beta_i$$

Uniqueness under determinism: While multiple transaction orderings may exist that satisfy this condition, deterministic execution ensures that any admissible ordering producing S_i is equivalent with respect to externally observable state. That is, if both T_i and T'_i satisfy the validity condition, then $\delta(S_{i-1}, T_i) = \delta(S_{i-1}, T'_i) = S_i$, meaning both orderings produce identical final state. Internal execution traces may differ, but the committed state root R_i uniquely determines the result.

Importantly, the transaction set T_i need not be known by the verifier. The header's validity is determined solely by the existence of *some* valid transaction set that produces the committed state within the resource bounds. This existential quantification is the key insight enabling bounded inclusion.

Important distinction: Bounded inclusion does not eliminate cryptographic proofs entirely. It replaces per-transaction Merkle inclusion proofs with selective state access proofs. The key difference: state proof size depends on the accessed state elements (typically O(log |S|) for a Merkle-Patricia tree), not on block size or transaction count. A verifier tracking k accounts needs O(k) state proofs per block, regardless of how many transactions the block contains.

4.3 State Root Properties

The state root $R_i = h(S_i)$ must satisfy:

• **Collision resistance:** Computationally infeasible to find $S \neq S'$ with $h(S) = h(S')$

• **Completeness:** Commits to entire state (no partial commitments)

• **Efficiency:** Computable in time proportional to state size

5. The Bounded Inclusion Principle

We now formalize the central contribution: a definition of transaction inclusion that requires no enumeration of transactions.

5.1 Definition

Definition 1 (Weak Bounded Inclusion): A transaction t is weakly included in block i if and only if: 1. t satisfies admission rules at time i 2. t's effects are consistent with R_i 3. The total execution cost remains within β_i **Important distinction:** Weak inclusion verifies that *some transaction with effects equivalent to t* was executed, not that the specific transaction t (with its unique identifier, signature, or hash) was included. This is weaker than traditional Merkle-based inclusion but sufficient for many applications including balance verification, state queries, and payment confirmation.

```
Formally, for a transaction t with precondition Pre(t) and
postcondition Post(t): t weakly included ■ ∃t' ∈ T_i : Pre(t) =
Pre(t') ∧ Post(t) = Post(t') ∧ Cost(t') ≤ β_i Note that ID(t) may
differ from ID(t'). For applications requiring proof of specific
transaction identity (e.g., timestamping, notarization), traditional
inclusion proofs remain necessary.
```

Critical distinction: Bounded inclusion proves the *existence* of a valid execution consistent with the header commitments, not the *reconstructability* of that specific execution. The actual transaction ordering and intermediate states need not be recoverable from headers alone. This is why data availability and fraud proofs remain relevant for full auditability, even though they are not required for lightweight verification of user-specific transactions.

5.2 Verification Without Enumeration

Traditional Merkle proofs answer: "Is this transaction in the list?" Bounded inclusion answers: "Could this transaction have occurred without violating the header invariant?" The latter question is stronger because it verifies consistency with the entire system state, not just membership in a list. A malicious transaction might be listed but produce an invalid state transition; bounded inclusion would reject it.

Important limitation: Bounded inclusion verifies *correctness* (valid state transitions within resource bounds), not *fairness* or *completeness*. Adversarial transaction ordering, censorship, and MEV (Miner Extractable Value) remain orthogonal problems requiring separate mechanisms such as fair ordering protocols, censorship resistance guarantees, or fraud proof systems. Bounded inclusion ensures that *if* a transaction is included, its effects are valid—it does not guarantee *which* transactions are included or in what order.

5.3 Example: Account Balance Update

Consider a transaction t = Transfer(Alice → Bob, 100 tokens): Pre(t): Alice.balance ≥ 100 in S_{i-1} Post(t): Alice.balance decreased by 100 and Bob.balance increased by 100 in S_i To verify inclusion using bounded inclusion: 1. Obtain state proof for Alice's balance in R_{i-1} (showing balance ≥ 100) 2. Obtain state proof for Alice's and Bob's balances in R_i (showing correct updates) 3. Verify total state

changes consistent with gas cost within β_i No transaction-list Merkle proof required. However, the verifier still needs state proofs (typically Merkle-Patricia proofs in the state trie) for the specific accounts involved. The key advantage: proof complexity is O(accounts_accessed), not O(log |transactions|). For a block with 10,000 transactions but only affecting our 2 accounts, we need 2 state proofs instead of 1 Merkle inclusion proof plus full transaction data.

6. Why Merkle Proofs Become Redundant

Under the bounded inclusion model, per-transaction Merkle tree inclusion proofs provide no additional security guarantee beyond what is already enforced by header commitments and state proofs.

Critical terminology clarification: Bounded inclusion does not eliminate cryptographic state proofs. Rather, it eliminates *enumeration-based inclusion proofs* whose size scales with transaction count. State access proofs (typically Merkle-Patricia proofs in the state trie) remain necessary, but their complexity depends solely on the number of state elements accessed by the verifier, not on block size or transaction volume. To be precise: we eliminate proofs that "transaction t appears in transaction list T_i" (which scale as $O(\log |T_i|)$ per transaction), but we retain proofs that "account A has balance B in state S_i" (which scale as $O(\log |S|)$ per account). The key distinction is that state proof complexity is independent of how many transactions are in the block—a verifier tracking k accounts needs the same witness size whether the block contains 10 or 10,000 transactions.

6.1 Information-Theoretic Argument

Merkle proofs verify that a transaction appears in a list. For state-modifying transactions (transfers, contract calls, state updates), this list-membership is neither necessary nor sufficient for validity: • Not sufficient: A transaction might be listed but produce invalid state • Not necessary: Validity depends on state consistency, not list membership The state root R_i already commits to all effects of all executed transactions. Any transaction not executed cannot have affected the state, making its absence verifiable through state proofs. **Important exception:** For non-state-modifying applications such as timestamping, notarization, or proof-of-existence, list membership IS the property of interest. Bounded inclusion cannot prove that a specific document hash was committed at a specific time—only that state was modified consistently. For these applications, traditional Merkle inclusion proofs remain necessary.

6.2 Security Equivalence

Consider an adversarial scenario where a validator attempts to exclude a valid transaction t: **With Merkle proofs:** Exclusion detected by absence from Merkle tree **With bounded inclusion:** Exclusion detected by state inconsistency In both cases, detection requires the verifier to query for transaction t. Bounded inclusion detects the exclusion at the same point but without requiring $O(\log n)$ proof verification.

6.3 Resource Guarantee

The key innovation is recognizing that execution bounds β_i provide an upper limit on verification cost: • Max transactions per block: β_i / min_gas • Max state changes: Determined by β_i • Verification cost: O(state changes), independent of block size This transforms worst-case verification from $O(n \log n)$ to $O(\beta_i)$, where β_i is chosen to be constant or slowly growing.

6.4 Asymptotic Witness Complexity

Proposition 1 (Asymptotic Witness Stability - Conditional): Assuming state commitments via balanced tree structures with depth $D = O(\log |S|)$ (e.g., Merkle-Patricia trees), a verifier tracking k

accounts requires witness size $O(k \cdot \log |S|)$, independent of transaction count $|T|$. This contrasts with Merkle inclusion witnesses requiring $O(k \cdot \log |T|)$.

Proof sketch: Merkle-based verification: For each of k transactions, verifier requires a Merkle proof of length $O(\log |T|)$ to verify inclusion in the transaction list. Total witness size: $O(k \cdot \log |T|)$. As blocks grow larger ($|T|$ increases), witness size increases logarithmically per transaction. **Bounded inclusion:** Verifier requires state proofs for k accounts. Each state proof is O(D) where D is tree depth. For balanced Merkle-Patricia trees, $D = O(\log |S|)$. Crucially, state proof size is independent of $|T|$—a block may contain 10 or 10,000 transactions, but if only k accounts are affected, witness size remains $O(k \cdot \log |S|)$. **Asymptotic behavior:** As transaction throughput scales ($|T| \to \infty$ while $|S|$ grows sublinearly), bounded inclusion witnesses remain bounded by state access patterns rather than transaction volume. ■

This asymptotic stability is the key advantage for light clients: verification cost depends on accounts tracked, not network activity. A mobile wallet tracking 5 accounts requires the same witness size whether the network processes 100 or 100,000 transactions per block. **Important note on state structure:** This analysis assumes balanced tree commitments. For alternative structures: • Account-chain architectures: proof size depends on chain depth, not global state size • Hash chains: $O(|S|)$ worst-case proof size • Sparse Merkle trees: amortized O(1) for recent accounts, O(256) worst-case for deep paths The specific asymptotic behavior depends on the chosen state commitment scheme.

6.5 Comparison to Vector Commitment Schemes

Verkle trees [2] and other vector commitment schemes (e.g., KZG commitments, polynomial commitments) reduce proof size from O(log n) to O(1) through algebraic techniques. While this is a significant improvement over traditional Merkle trees, these schemes still preserve the *list-membership model*: they answer "is this transaction in the list?" with a smaller proof. Bounded inclusion is fundamentally different. Rather than optimizing proof size for enumeration, we remove the enumeration requirement entirely by shifting verification from transaction presence to state consistency under bounded execution. A Verkle tree proves "transaction t appears at position i in the list," while bounded inclusion proves "transaction t's effects are consistent with the committed state within resource bounds." The distinction matters for light client design: • Verkle trees still require transaction data distribution infrastructure • Bounded inclusion requires only state access (which light clients already need) • Verkle verification depends on transaction count, bounded inclusion on account count • Verkle trees can be combined with bounded inclusion for hybrid optimization In systems where state access patterns are sparse (users care about few accounts) but transaction volume is high (many users active), bounded inclusion provides asymptotically better scaling than any enumeration-based proof system, regardless of proof optimization.

7. Security Analysis and Formal Guarantees

We now prove that bounded inclusion provides equivalent security to Merkle-based verification under the stated assumptions.

7.1 Threat Model

We operate under the following adversarial model: **Adversary capabilities:** • May submit arbitrary witnesses (state proofs, transaction data) • May attempt to construct invalid state transitions • May selectively censor or reorder transactions • May coordinate with minority of validators (< f threshold) **Adversary limitations:** • Cannot forge cryptographic commitments (collision resistance assumption) • Cannot exceed honest validator resource bounds β_i • Cannot forge quorum signatures σ_i (Byzantine threshold assumption) • Cannot break hash function preimage resistance **Verifier properties:** • Deterministic: given same inputs, produces same verification result • Resource-bounded: verification must complete within client constraints • Stateless: does not maintain global state, only headers and tracked accounts **Trust assumptions:** • Honest supermajority: > f of n validators follow protocol • Data availability: headers and state proofs are accessible • Liveness: network eventually delivers messages This model is equivalent to standard light client threat models, with the key difference that we replace transaction enumeration with state consistency checks.

7.2 Main Security Theorem

Theorem 1 (Header Sufficiency for Tracked Accounts): If (1) headers are finalized by honest quorum, (2) state transitions are deterministic with canonical ordering, and (3) resource bounds are enforced, then a verifier tracking account set A accepting H_i accepts all and only those executions consistent with H_i with respect to accounts in A. **Corollary (Partial Observability):** The verifier cannot detect differences in state for accounts outside A. Transactions affecting only untracked accounts may be substituted without detection.

Proof sketch:

Consider any invalid transaction t' that an adversary attempts to include, affecting accounts in set A: Case 1: t' violates deterministic execution on accounts in A $\rightarrow$ Produces state S'_i with different values for accounts in A $\rightarrow$ State proofs for A under $R_i = h(S_i)$ will not verify with S'_i values $\rightarrow$ Rejected by verifier Case 2: t' exceeds resource bounds $\rightarrow$ $\text{Cost}(T_i \cup \{t'\}) > \beta_i$ $\rightarrow$ Violates header constraint (assuming aggregate cost verification) $\rightarrow$ Rejected at consensus level, not by light client directly Case 3: t' produces valid state for A but invalid state for untracked accounts B $\rightarrow$ Verifier tracking only A cannot detect this $\rightarrow$ Requires trust in validator quorum to reject invalid global state $\rightarrow$ This is the fundamental limitation of partial verification Conversely, any valid transaction t producing effects on A consistent with S_i will be accepted. The verifier verifies consistency for tracked accounts only. ■

Critical limitation: This theorem does not provide guarantees about untracked accounts. An adversary could include transactions affecting accounts outside A that violate global invariants (e.g., mint tokens illegally) without detection by verifiers tracking only A. Full security requires either (a) tracking all accounts (impractical), or (b) trusting the validator quorum to enforce global invariants. This is identical

to the trust model in traditional light clients.

Supporting Lemmas

Lemma 1 (State Commitment Sufficiency): If h is collision-resistant and δ is deterministic with canonical ordering, then $R_i = h(S_i)$ uniquely determines the state reachable from S_{i-1} within β_i. **Proof:** By collision resistance, we cannot find $S'_i \neq S_i$ with $h(S'_i) = h(S_i)$ except with negligible probability. By determinism and canonical ordering (Assumption A1'), for any transaction set T_i satisfying $Cost(T_i) \leq \beta_i$, the output state $\delta(S_{i-1}, T_i)$ is unique. Therefore R_i uniquely identifies the valid post-state. ■

Lemma 2 (Partial Observability Bound): A verifier tracking k accounts can distinguish executions that differ on those k accounts but cannot detect differences on untracked accounts. **Proof:** State proofs for accounts in set A reveal only the values of those accounts in S_i. By construction of Merkle-Patricia trees (or similar structures), knowledge of k account values provides no information about the remaining $|S| - k$ accounts beyond their contribution to the root hash. Therefore, two states S_i and S'_i that agree on all accounts in A but differ on accounts in the complement set (S minus A) are indistinguishable to a verifier with only state proofs for A. ■

Implications: Lemma 2 formalizes the fundamental limitation of partial verification. This is not a weakness of bounded inclusion specifically—it applies equally to traditional light clients using Merkle proofs. The difference is that bounded inclusion makes this limitation explicit in its security model.

7.3 Adversarial Models

We consider three adversarial capabilities: **A1: Transaction censorship** - Validator excludes valid transactions **A2: Invalid inclusion** - Validator includes invalid transactions **A3: State forgery** - Validator produces false state commitments Bounded inclusion defends against: • A2 by deterministic execution (invalid transactions cannot produce valid state) • A3 by cryptographic state commitments (collision resistance) A1 (censorship) is detectable through state proofs but not preventable by bounded inclusion alone—this is identical to Merkle-based systems where censorship requires external mechanisms such as fraud proofs, fishermen, or social consensus. Bounded inclusion guarantees correctness of included transactions, not completeness of the transaction set. MEV and adversarial ordering similarly require dedicated fairness mechanisms orthogonal to our verification model.

7.4 Trust Assumptions

Bounded inclusion requires: 1. **Honest majority quorum:** > f of n validators are honest (Byzantine fault tolerance) 2. **Liveness:** Eventually, all valid transactions are included 3. **Data availability:** Headers are available and state proofs are accessible on demand These are identical to assumptions in Merkle-based light clients. We do not introduce new trust requirements.

Data availability clarification: Bounded inclusion assumes header availability (the header chain $\{H_i\}$ is accessible) and on-demand state proof availability (verifiers can request state proofs for specific

accounts from full nodes). This is identical to existing light client assumptions—SPV clients assume block headers are available, and Ethereum light clients assume state proofs are available via LES (Light Ethereum Subprotocol). We do not assume transaction data availability, which is the key difference: traditional light clients need Merkle proofs derived from transaction data, while bounded inclusion needs only state proofs.

8. Implications for Light Client Design

Bounded inclusion enables radically simplified light client architectures with predictable resource requirements.

8.1 Minimal Light Client Requirements

A light client needs only: 1. Header chain {H_0, H_1, ..., H_n} 2. State proofs for accounts of interest (from state trie) 3. (Optional) Fraud proof verification for full auditability No per-transaction Merkle inclusion proofs. No full transaction downloads. No global VM execution. State proofs are still required but their size depends on state structure (typically O(log |S|) for accounts), not transaction count.

8.2 Verification Complexity

Traditional light client: • Sync: O(n) headers + O(k log m) Merkle proofs for k transactions in m-sized blocks • Verify transaction: O(log m) proof verification • Storage: O(n) headers + O(k log m) proof data Bounded inclusion light client: • Sync: O(n) headers only • Verify transaction: O(1) state proof verification* • Storage: O(n) headers + O(1) per account *O(1) with respect to block transaction count, not state size. State proofs are O(log |S|) where |S| is total accounts, but this is independent of block size. For a client tracking k accounts across n blocks with average block size m: Traditional: O(n + k log m) Bounded: O(n + k log |S|) ≈ O(n + k) when |S| is fixed The key advantage: bounded inclusion verification cost is independent of transaction volume per block.

8.3 Browser-Native Feasibility

Bounded inclusion makes browser-native blockchain verification practical: • Header verification: ~1KB per header, 10,000 headers = 10MB • State proofs: ~500 bytes per account proof • JavaScript execution: No cryptographic operations beyond hashing • Memory: O(accounts tracked), not O(blockchain history) A browser client tracking 100 accounts across 10,000 blocks requires: • Storage: ~10MB headers + ~50KB state proofs = 10.05MB • Bandwidth: ~10MB one-time sync + ~1KB per new block • Computation: O(1) hash verification per block This is feasible even on mobile devices with limited resources.

8.4 Reference Implementation: Minimal Verifier

We provide reference pseudocode for a minimal bounded inclusion verifier. This algorithm can be implemented in any language with cryptographic hash support and is suitable for browser, mobile, or embedded environments.

```
// Verify transaction inclusion via bounded inclusion // WARNING: This verifies consistency with OBSERVED state only. // Transactions with side effects on untracked accounts cannot be fully validated. // Light clients must trust validator quorum for global state consistency. function VerifyTransaction(t, H_i, H_{i-1}, state_proofs): // Input: // t: transaction to verify // H_i: current header (R_i, β_i, σ_i) // H_{i-1}: previous header (R_{i-1}, β_{i-1}, σ_{i-1}) // state_proofs: {proof_pre, proof_post} for affected accounts // Step 1: Verify header signature if not VerifyQuorumSignature(H_i.σ_i, H_i): return REJECT // Invalid header signature // Step 2: Verify state proofs against
```

```
committed roots accounts_pre = ExtractState(state_proofs.proof_pre) if
not VerifyStateProof(accounts_pre, H_{i-1}.R_{i-1}): return REJECT //
Precondition state proof invalid accounts_post =
ExtractState(state_proofs.proof_post) if not
VerifyStateProof(accounts_post, H_i.R_i): return REJECT //
Postcondition state proof invalid // Step 3: Verify transaction
preconditions if not CheckPreconditions(t, accounts_pre): return
REJECT // Transaction preconditions not met // Step 4: Verify state
transition consistency // Note: This checks if observed state changes
are consistent with t, // not that they are ONLY caused by t (multiple
transactions may affect // the same accounts in the same block) if not
StateChangesConsistentWith(t, accounts_pre, accounts_post): return
REJECT // State transition inconsistent // Step 5: Resource bound
check (aggregate only) // Light clients cannot verify per-transaction
costs without execution // This checks that validator attestation is
valid, not individual cost if not HeaderAttestsValidExecution(H_i):
return REJECT // Header indicates invalid execution return ACCEPT //
Transaction weakly verified via bounded inclusion // Helper: Check if
state changes are consistent with transaction function
StateChangesConsistentWith(t, pre, post): // Example: for
Transfer(from, to, amount) // Check that 'from' decreased by at least
amount // and 'to' increased by at least amount // (may have changed
more due to other transactions in same block) delta_from =
pre[t.from].balance - post[t.from].balance delta_to =
post[t.to].balance - pre[t.to].balance return delta_from >= t.amount
and delta_to >= t.amount // Helper: Verify Merkle-Patricia state proof
function VerifyStateProof(state, root): return
MerklePatriciaVerify(state, root)
```

Key observations: • No transaction list enumeration: We never iterate over T_i or verify t ∈ T_i • Per-account complexity: O(log |S|) per account proof, independent of block size • Minimal state access: Only accounts affected by t need proofs • Deterministic: Same inputs always produce same result • Stateless: Verifier maintains no global state between calls • Partial verification: Only validates tracked accounts, not global state **Important limitations:** • Cannot detect invalid transactions affecting only untracked accounts • Cannot verify per-transaction gas costs without full execution • Multiple transactions per account per block requires "consistent with" logic, not equality • Weak inclusion: verifies effect equivalence, not transaction identity This algorithm can verify thousands of transactions per second on modest hardware, making it suitable for browser extensions, mobile wallets, and IoT devices. The JavaScript implementation requires approximately 150 lines of code plus a cryptographic library (e.g., noble-crypto for hashing and signature verification).

9. Case Study: Zenon Network Architecture

The Zenon Network provides a concrete instantiation of bounded inclusion principles through its account-chain architecture and plasma-based resource model [5].

9.1 Account-Chain Isolation

Zenon implements a dual-ledger structure: • **Account chains:** Individual transaction chains per account • **Momentum chain:** Global consensus chain coordinating account chains This architecture naturally enforces deterministic execution: each account chain processes transactions sequentially, eliminating race conditions and ensuring that state transitions are pure functions of previous state and transaction content.

9.2 Plasma-Bounded Execution

Zenon uses plasma as the bounded resource β_i: • Each transaction consumes plasma proportional to computational cost • Accounts have bounded plasma regeneration rates • Momentums (blocks) commit to total plasma consumed This provides the execution bound required for bounded inclusion: verifiers can confirm that total plasma consumption is within limits without executing all transactions.

9.3 Momentum-Based Finality

Momentums serve as the header commitment H_i: • Commit to account-chain state roots • Include plasma bounds • Signed by validator quorum (Pillars) Light clients need only track momentum headers to verify any account's transactions, exemplifying the bounded inclusion model in practice.

9.4 Concrete Instantiation Example

We now provide a concrete example of bounded inclusion verification in Zenon's architecture to demonstrate how the abstract model maps to a real system.

Scenario: Light client Alice wants to verify that her Send transaction at height h was included and executed correctly. **System state:** • Momentum M_h contains state root R_h and plasma bound $\beta_h = 21{,}000$ • Alice's account chain A_{Alice} has prior state $S_{A,h-1}$ • Transaction t: Send(Bob, 100 ZNN) with plasma cost 21,000 **Traditional Merkle approach:** 1. Download transaction list T_h from block 2. Request Merkle proof π_t proving $t \in T_h$ 3. Verify π_t against transaction root in M_h 4. Download account state changes 5. Verify state changes correspond to t Witness size: $O(\log |T_h|)$ Merkle proof + transaction data **Bounded inclusion approach:** 1. Request state proof for A_{Alice} from R_{h-1}: balance = 150 ZNN 2. Request state proof for A_{Alice} from R_h: balance = 50 ZNN 3. Request state proof for A_{Bob} from R_h: balance increased by 100 ZNN 4. Verify: ΔS consistent with Send operation 5. Verify: $\text{plasma\_cost}(\text{Send}) = 21{,}000 \leq \beta_h$ Witness size: $O(1)$ state proofs (3 account proofs), independent of $|T_h|$ **Key observation:** Even if M_h contains 1,000 transactions, Alice's verification requires only 3 state proofs. The momentum header commitment R_h guarantees that *some* valid transaction set produced this state, and Alice verifies that her specific transaction's effects are consistent with this state transition. **Verification predicate (corrected for multi-transaction case):** $\text{Valid}(t, M_h) := \text{Balance}(A_{Alice}, R_{h-1}) \geq 100 \wedge \Delta\text{Balance}(A_{Alice}) = \text{Balance}(A_{Alice}, R_{h-1}) -$

Balance(A_Alice, R_h) ≥ 100 ∧ ΔBalance(A_Bob) = Balance(A_Bob, R_h) - Balance(A_Bob, R_{h-1}) ≥ 100 ∧ plasma_cost(t) ≤ β_h **Critical correction:** The original predicate required exact balance changes (Balance(A_Alice, R_h) = Balance(A_Alice, R_{h-1}) - 100), which would fail if Alice made multiple transactions in the same momentum. The corrected predicate uses *inequality* (≥) to verify that *at least* the claimed transaction's effects are present. This handles cases where: • Alice sends multiple transactions in M_h • Alice receives funds from others in M_h • Multiple state changes occur simultaneously If this predicate holds, Alice has cryptographic proof that a transaction with effects consistent with t was included, without enumerating the transaction list.

9.5 Implementation Benefits

• **Parallel verification:** Account chains can be verified independently

• **Predictable costs:** Verification cost proportional to accounts tracked, not network size

• **Browser compatibility:** Syrius wallet demonstrates feasibility

• **No proof infrastructure:** No need for Merkle proof servers

10. Open Questions and Future Research

While bounded inclusion provides a sound foundation for lightweight verification, several research directions remain open.

10.1 Fraud Proof Construction

When a validator violates bounded inclusion (e.g., by exceeding β_i or producing inconsistent state), an efficient fraud proof is needed. Since bounded inclusion proves existence of valid execution rather than reconstructability, fraud proofs serve a critical role in full auditability. Open questions: • What is the optimal fraud proof size for bounded inclusion violations? • Can fraud proofs be generated without full block data access? • How do fraud proofs interact with account-chain architecture? • What is the relationship between data availability and bounded inclusion verification? Note that while light clients can verify their own transactions without fraud proofs, system-wide auditing and accountability still require mechanisms to challenge invalid headers, particularly for detecting resource bound violations or state inconsistencies affecting other users.

10.2 Zero-Knowledge Integration

Zero-knowledge proofs (ZK-SNARKs/STARKs) could compress bounded inclusion verification: • ZK proof that execution stayed within β_i • ZK proof of deterministic state transition • Recursive composition for historical verification Challenge: Maintaining O(1) verification without trusted setup.

10.3 Adversarial Scheduler Analysis

Our model assumes benign transaction ordering. Future work should analyze: • Adversarial transaction ordering strategies • Front-running and MEV (Miner Extractable Value) under bounded inclusion • Fair ordering mechanisms compatible with header-only verification

10.4 Cross-Chain Applications

Bounded inclusion may enable efficient light client bridges between blockchains: • Reduced proof size for cross-chain verification • Header-only relay chains • Interoperability without full node requirements

11. Related Work

Bounded inclusion builds upon several lines of blockchain research.

11.1 Light Client Protocols

Simplified Payment Verification (SPV) [1] introduced header-only verification for Bitcoin. Our work extends this concept by eliminating even the Merkle proof requirement for transaction inclusion. Ethereum's light client protocol [6] requires Merkle proofs for state access; bounded inclusion removes this overhead.

11.2 State Commitment Schemes

Verkle trees [2] and other advanced commitment schemes reduce proof sizes but maintain the enumeration model. Accumulators [3] enable batch verification but require expensive updates. Bounded inclusion avoids proof generation entirely.

11.3 Deterministic Execution

Deterministic consensus has been explored in traditional distributed systems [7]. Blockchain VMs like EVM and WASM enforce determinism, but prior work has not leveraged this for lightweight verification as we propose.

11.4 Resource Metering

Gas metering (Ethereum) and similar systems bound execution cost. Our contribution recognizes that these bounds enable verification without transaction enumeration when combined with deterministic execution.

12. Conclusion

This paper has presented bounded inclusion without Merkle trees, a novel verification paradigm that leverages deterministic state transitions and header-committed execution bounds to eliminate per-transaction inclusion proofs. We have shown that under specific architectural constraints, transaction inclusion can be inferred from header-level commitments alone, reducing verification complexity from O(log n) per transaction to O(1) per user interaction.

The key insight is recognizing that Merkle proofs verify membership in a list, while security ultimately depends on state consistency. By committing to execution bounds and enforcing deterministic transitions, we can verify correctness without enumerating transactions. This shifts verification from "what was executed" to "what could have been executed," a more efficient question for lightweight clients.

The practical implications are significant. Browser-native blockchain verification becomes feasible with predictable resource requirements. Mobile and embedded devices can independently verify transactions without full node infrastructure. Cross-chain bridges can operate with reduced overhead.

The Zenon Network demonstrates that bounded inclusion is not merely theoretical—it provides a working instantiation through account-chain isolation, plasma-bounded execution, and momentum-based finality. This positions Zenon as what we term a *deterministic-interface blockchain*, prioritizing execution invariants over state enumeration.

Future work will explore fraud proof mechanisms, zero-knowledge integration, and adversarial analysis. We believe bounded inclusion represents a fundamental rethinking of blockchain verification, with potential applications far beyond the specific architecture studied here.

References

[1] Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. bitcoin.org

[2] Kuszmaul, J. (2019). Verkle Trees. Ethereum Research Forum.

[3] Boneh, D., Bünz, B., & Fisch, B. (2019). Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. CRYPTO 2019.

[4] Ben-Sasson, E., Chiesa, A., Tromer, E., & Virza, M. (2014). Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. USENIX Security.

[5] Zenon Network. (2021). Network of Momentum Technical Whitepaper. zenon.network

[6] Wood, G. (2014). Ethereum: A Secure Decentralised Generalised Transaction Ledger. Ethereum Project Yellow Paper.

[7] Lamport, L. (1998). The Part-Time Parliament. ACM Transactions on Computer Systems, 16(2), 133-169.

Appendix A: Notation Summary

Symbol	Definition
B_i	Block at height i
T_i	Transaction set in block i
S_i	Global state after block i
δ	Deterministic state transition function
R_i	State root commitment h(S_i)
β_i	Execution resource bounds
H_i	Block header (R_i, β_i, σ_i)
σ_i	Validator quorum signature
M_i	Merkle root (traditional systems)
$h(\cdot)$	Cryptographic hash function