SWMT: A Sliding Window Merkle Tree with Delayed Writes for Scalable Blockchain State Management

NIANZU SHENG, Hefei Institutes of Physical Science, Chinese Academy of Sciences, China and University of Science and Technology of China, China
TONG ZHOU*, Hefei Institutes of Physical Science, Chinese Academy of Sciences, China and Anhui Zhongke Jingge Technologies Co., Ltd, China
HE ZHAO*, Hefei Institutes of Physical Science, Chinese Academy of Sciences, China and University of Science and Technology of China, China
XIAOFENG LI, Hefei Institutes of Physical Science, Chinese Academy of Sciences, China and University of Science and Technology of China, China
JINLIN XU, Hefei Institutes of Physical Science, Chinese Academy of Sciences, China and University of Science and Technology of China, China

The state tree used for blockchain state management has emerged as a core performance bottleneck, primarily due to the latency involved in generating cryptographic commitments during block production and validation. To address this issue, we present a delayed write mechanism that decouples commitment computation from block execution. We introduce the Sliding Window Merkle Tree (SWMT), a cyclic in-memory cache that temporarily buffers recent state updates, deferring their integration into the state tree. SWMT uses a 32-bit window field to efficiently manage cache lifespan and support rapid commitment generation. To handle fluctuating transaction loads, we propose a TCP-inspired congestion control strategy that adapts caching granularity to dynamic workload intensity. Our design can be integrated with existing MPT and Verkle tree structures, while preserving proof succinctness and verifiability. Evaluation on 6 million real Ethereum blocks demonstrates up to a 61.3× reduction in commitment latency, a 6.3× speedup in block execution. These results highlight a practical and deployable approach to enhancing blockchain throughput and scalability.

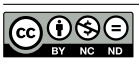
CCS Concepts: • **Information systems** → **Data structures**; *Distributed storage*; • **Security and privacy** → *Distributed systems security*.

Additional Key Words and Phrases: Blockchain, Authenticated data structure, Scalability

ACM Reference Format:
Nianzu Sheng, Tong Zhou, He Zhao, Xiaofeng Li, and Jinlin Xu. 2026. SWMT: A Sliding Window Merkle Tree with Delayed Writes for Scalable Blockchain State Management. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 87 (February 2026), 25 pages. https://doi.org/10.1145/3786701

*Corresponding authors: He Zhao and Tong Zhou.

Authors' Contact Information: Nianzu Sheng, nianzu@mail.ustc.edu.cn, Hefei Institutes of Physical Science, Chinese Academy of Sciences, Hefei, Anhui, China and University of Science and Technology of China, Hefei, Anhui, China; Tong Zhou, tzhou@hfcas.ac.cn, Hefei Institutes of Physical Science, Chinese Academy of Sciences, Hefei, Anhui, China and Anhui Zhongke Jingge Technologies Co., Ltd, Hefei, Anhui, China; He Zhao, zhaoh@hfcas.ac.cn, Hefei Institutes of Physical Science, Chinese Academy of Sciences, Hefei, Anhui, China and University of Science and Technology of China, Hefei, Anhui, China; Xiaofeng Li, xfli@hfcas.ac.cn, Hefei Institutes of Physical Science, Chinese Academy of Sciences, Hefei, Anhui, China and University of Science and Technology of China, Hefei, Anhui, China; Jinlin Xu, jlxu@hfcas.ac.cn, Hefei Institutes of Physical Science, Chinese Academy of Sciences, Hefei, Anhui, China and University of Science and Technology of China, Hefei, Anhui, China.



ACM 2836-6573/2026/2-ART87
https://doi.org/10.1145/3786701

1 Introduction

Blockchain [22] enables trustless data management and has shown significant potential in powering decentralized applications (DApps). It has driven the growth of sectors such as decentralized finance (DeFi) [21], non-fungible tokens (NFTs) [31], blockchain gaming, and real-world asset (RWA) tokenization [35]. For example, as of July 2025, Ethereum [33] had a market capitalization of approximately $300 billion [9]. Its thriving ecosystem included a total value locked (TVL) of around $80 billion in DeFi protocols [34], and over $124 billion in stablecoin supply [15]. These applications rely on the storage and management of on-chain states—such as account balances and smart contract data—which are maintained by the state tree, a fundamental component of the blockchain infrastructure.

A state tree is an authenticated data structure (ADS) [27], which efficiently organizes and indexes all on-chain states. An ADS produces succinct cryptographic commitments to a dataset, enabling any party to verify the authenticity of specific data elements using compact proofs. In each block, state updates regenerate a commitment (namely, the state root), which serves as a global snapshot of the blockchain's current state. This commitment is embedded in the block header and propagated throughout the network. Upon receiving a new block, each node independently re-executes the transactions and computes its own commitment. If the result matches the commitment declared in the block, the state transition is deemed valid, thereby ensuring consistency of the state across the network.

The state tree also enables stateless nodes—lightweight participants that do not store the full state. These nodes synchronize blocks and transactions, and rely on state data provided by full nodes (i.e., nodes that maintain the entire blockchain state) to execute transactions without maintaining a local state. With the help of proofs generated by the state tree, stateless nodes can verify the correctness of state data provided externally. This mechanism further promotes decentralization by allowing broader participation without requiring every node to store the full state.

However, the state tree critically limits blockchain scalability. Ethereum currently adopts a Merkle Patricia Trie (MPT) [33] for key-value storage, where each state is mapped to a leaf node, and the path from the root to the leaf is derived from its associated key. When a state update occurs, the system must update the path from the affected leaf to the root, which involves multiple disk I/Os. Studies have shown that operations on the state tree account for approximately 80% of block execution time [19]. Existing optimizations, such as binary trie [5] and Verkle trees [17], primarily aim to reduce state proof sizes in order to lower bandwidth costs for stateless nodes. However, these techniques often introduce additional computational overhead. For instance, verkle trees replace hash-based commitments with elliptic curve-based vector commitments, achieving even smaller proofs, though with increased computational complexity.

As the system scales and transaction complexity continue to grow, the burden of updating the state tree is becoming increasingly significant. Ethereum employs a gas limit to restrict the maximum number of operations per block. To support more complex applications and higher transaction density, this gas limit has been gradually increased in recent years [11], resulting in more frequent state changes. In addition, core developers have proposed shortening the block interval [1] to further reduce network confirmation latency. These trends imply that the state tree may need to handle a larger number of updates within a shorter time window, making its performance a critical bottleneck to improving throughput.

To address this bottleneck, we propose a delayed write mechanism inspired by write-ahead logging (WAL) in database systems. Instead of immediately applying each block's state updates to the state tree, we temporarily buffer them in a transient Merkle tree. Multiple blocks are then committed in batches, while new transient trees are constructed in parallel. This effectively decouples state

root generation from block production and validation, thereby improving system throughput while preserving consistency.

Despite its benefits, this mechanism poses two key challenges:

First, to ensure consistency across nodes, each block must still generate a temporary state root. This requires the system to maintain multiple transient state trees, which increases the complexity of state proofs and may negatively impact the performance of stateless nodes. To address this, we design a Sliding Window Merkle Tree (SWMT) structure that caches recent state updates. Each tree node in the SWMT includes a 32-bit window field that acts as a time indicator, marking when each subtree was last updated. By referencing this field, nodes can identify the oldest subtrees, enabling fast pruning of stale data. Through periodic pruning of tree nodes, SWMT supports efficient state caching and root generation across multiple blocks.

Second, the volume of state updates exhibits significant temporal variation, depending on transaction workload and application behavior. If updates are merged based on a fixed number of blocks, it may lead to underutilized transient tree capacity during low activity periods or congestion during high activity periods, both of which can degrade performance. To mitigate this, we introduce a congestion control mechanism inspired by TCP, built on top of the SWMT. The system dynamically adjusts the block height range represented by each bit in the window field. When updates are sparse, each bit spans more blocks to enhance caching capacity; when updates are dense, the span is reduced to distribute state changes more evenly, thereby balancing memory pressure and pruning workload.

Our contributions are:

- We propose a delayed write mechanism that decouples state root generation during block execution, effectively alleviating performance bottlenecks;
- We design a SWMT that caches recent state updates and enables fast pruning of stale subtrees through a 32-bit temporal window;
- We introduce a TCP-inspired congestion control mechanism that improves SWMT capacity utilization by ensuring a balanced distribution of state updates;
- We implement our mechanism atop both MPT and Verkle trees, achieving integration and significant performance gains on real Ethereum mainnet data. Our evaluation shows up to a 61.3× reduction in commitment generation time and a 6.3× reduction in block execution latency.

2 Background

2.1 Ethereum Blockchain System

Ethereum is a decentralized system maintained by a large number of nodes that collectively agree on a canonical chain of blocks through a consensus protocol. To produce a new block, the proposer selects a set of transactions, executes them to update the global state, and computes the corresponding state root. This root, embedded in the block, acts as a succinct cryptographic commitment to the resulting state and is broadcast to the network along with the block.

Each receiving node must re-execute the transactions against its local state to independently compute the resulting state root. The block is considered valid only if the locally computed root matches the one declared in the block; otherwise, it is rejected.

Therefore, state root computation is required during both block production and validation, making it a critical performance bottleneck.

2.2 Accounts and State Model

Ethereum defines two types of accounts: externally owned accounts (EOAs) and smart contract accounts, both uniquely identified by a 20-byte address. An EOA maintains key attributes such as

a balance and a transaction counter (nonce), while a contract account additionally stores a code hash that identifies its associated contract code. All account information is stored indexed by the account address.

During execution, a smart contract can also access and modify its own storage, which is organized as a mapping located by a pair of values: the contract address and a slot key. Therefore, Ethereum's global state can be abstracted as two types of mappings:

- **Account state:** `address` → `account`. Includes fields such as the account balance, nonce, and code hash.
- **Contract storage state:** `(address, slot key)` → `value`. Represents the value stored in each storage slot of a contract.

Ethereum supports two types of transactions: native token transfers and contract calls, i.e., invocations of smart contracts. A native token transfer affects only the balance and nonce of the involved accounts. In contrast, a contract call triggers the execution of smart contract logic. Smart contracts run on the Ethereum Virtual Machine (EVM), which takes the current global state and block context (such as timestamp and block height) as input. Since the EVM does not include any source of randomness, its execution is deterministic. Given the same state and transaction input, it will always produce the same results and state changes.

2.3 Authenticated Data Structures in Blockchain

In blockchain systems, ADSs—such as MPT, Verkle Tree, and Sparse Merkle Tree [12]—serve as the underlying key-value databases to manage the blockchain state. Each block's state is uniquely identified by the root commitment of the ADS, which corresponds to an immutable snapshot. Verifiers can validate the existence and correctness of arbitrary states based on the commitment and the corresponding proof.

During block execution, the system loads the state snapshot using the root commitment of the previous block, ensuring that all read operations are based on a stable view. As the state transition process is deterministic, nodes with identical inputs will derive consistent state updates. By comparing the newly generated root commitments, nodes can efficiently verify each other's execution results, thereby achieving consensus.

2.3.1 Definition. To formalize the behavior of such ADSs, we define a **Snapshot-ADS**. Let the state set be:

$$S = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}, \quad \forall\, i \neq j,\ k_i \neq k_j$$

The distinct keys in S imply the existence of a deterministic order (e.g., lexicographical order). The ADS structure orders the keys to guarantee a unique root commitment. The Snapshot-ADS provides the following three interfaces:

(1) **Commit**$(S) \rightarrow C_{\text{root}}$: Given the set S, returns a succinct commitment C_{root}.
(2) **Prove**$(C_{\text{root}}, k_i) \rightarrow \pi_i$: Generates a proof π_i for the key k_i with respect to the ADS structure represented by C_{root}.
(3) **Verify**$(C_{\text{root}}, k, v, \pi_i) \rightarrow \{\text{true}, \text{false}\}$: Verifies whether the key-value pair (k, v) is included in the committed set using the provided proof and the commitment.

2.3.2 Properties. A Snapshot-ADS must satisfy three properties:

(1) **Verifiability:** For any $(k, v) \in S$, the **Verify** function should return `true`; otherwise, it should return `false`. This also ensures consistent state views and deterministic query results among nodes.

(2) **Write Consistency:** Identical state sets yield identical commitments:

$$\forall S_1, S_2, \quad S_1 = S_2 \Rightarrow \textbf{Commit}(S_1) = \textbf{Commit}(S_2)$$

This property ensures that all nodes can verify the consistency of each other's execution results.

(3) **Key-Value Semantics:** State updates follow overwrite semantics. Let S be the current state set and S_u an update set. The merged state is $S' = S \cup S_u$, where for any key k:

$$S'(k) = \begin{cases} S_u(k), & \text{if } k \in \text{dom}(S_u) \\ S(k), & \text{if } k \in \text{dom}(S) \wedge k \notin \text{dom}(S_u) \\ \perp, & \text{otherwise} \end{cases}$$

Here, $\text{dom}(S)$ denotes the set of all keys in S, and $\perp$ represents a null value indicating that the key is absent from both S and S_u.

It follows that updates satisfy the associativity property:

$$(S_1 \cup S_2) \cup S_3 = S_1 \cup (S_2 \cup S_3) \tag{1}$$

This holds because each key always reflects its most recent assignment regardless of merge order.

Additionally, the following identity holds:

$$S_1 \cup (S_2 \setminus S_1) = S_2 \quad \text{if } S_1 \subset S_2 \tag{2}$$

Here, $\setminus$ denotes the set difference operator, which removes all key-value pairs whose keys are present in the left-hand set. Intuitively, this ensures that merging the disjoint remainder $S_2 \setminus S_1$ back into S_1 exactly reconstructs S_2.

The Snapshot-ADS properties will be used to demonstrate the correctness of our design.

2.4 Merkle Patricia Trie

The MPT is an authenticated data structure that combines properties of both Trie (a prefix tree for key lookup) and Merkle trees (which provide cryptographic integrity), and generates a succinct root commitment over key-value pairs (see Figure 1). MPT structures consist of two types of internal elements: path nodes and leaf nodes. Each key is first decomposed into a sequence of symbols, which determines a unique prefix path in the Trie, where the corresponding value is stored in the leaf node. Since the symbol sequence has a well-defined order, the construction of the Trie is deterministic for any given set. On top of this structure, the MPT incorporates a hashing mechanism: each leaf node encodes its value using a hash function, and each path node recursively embeds the hash values of its child nodes, ultimately producing the root commitment.

Because each node in the MPT is identified by its hash value, any update to the key–value set produces new tree nodes rather than altering existing ones. Consequently, each root hash corresponds to an immutable snapshot of the entire system state.

Based on this design, we can briefly explain how the MPT satisfies the requirements of a Snapshot-ADS.

Verifiability: For any root commitment C_{root} and key k, a lookup operation always traverses a deterministic path that leads to a unique leaf node. The hash values of all nodes along this path can be used to construct a proof. Given the collision resistance of the hash function, the verification process is reliable.

Write Consistency: As the Trie construction order is deterministic and each tree node's hash value is fixed by its content, identical state sets always produce the same tree structure and root commitment.

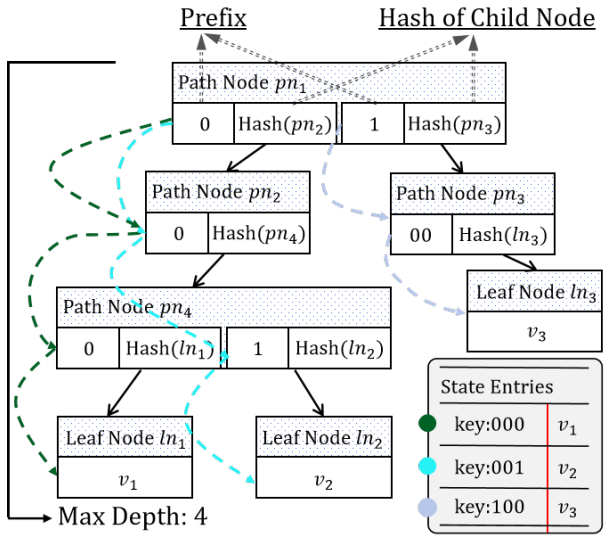


Fig. 1. The MPT structure.

Key-Value Semantics: When updating a key-value pair, the system updates all tree nodes along the corresponding path and recomputes their hash values, which propagate upward to the root node. Subsequent queries therefore return the most recent value.

It is worth noting that the above properties of the MPT rely on the determinism of all attributes within tree nodes. Once these attributes are fixed, their hash values are uniquely determined as well, ensuring that the tree structure and the final root commitment are deterministic. Based on this observation, we can formulate the following invariant:

Invariant 1. *If additional attributes are introduced into MPT nodes, the structure still satisfies Snapshot-ADS properties as long as these attributes are deterministically assigned during construction.*

2.5 Performance Bottleneck of ADS

We conducted preliminary experiments based on the Ethereum Go implementation. First, we tested the commitment computation time of MPT and Verkle tree under continuous data insertion, with each batch containing 5,000 entries. As shown in Figure 2, both Verkle tree and MPT exhibit increasing latency as the data volume grows, with the MPT's latency growing more rapidly. At the scale of 100 million entries, inserting 5,000 entries takes approximately 3 s for the MPT and 1 s for the Verkle tree. The difference in their growth rates mainly stems from their branching factor: the MPT is a 16-ary tree, whereas the Verkle tree is 256-ary.

We further analyzed the performance bottlenecks of the two structures. Specifically, we disentangled the impact of I/O and computation using two approaches: data prefetching at the database layer to mask I/O latency, and replacing elliptic curve point computations with zero-cost dummy results. Subsequently, we replayed approximately 4 million Ethereum mainnet blocks and measured the state tree commitment time under different configurations.

Figure 3(a) shows MPT performance without I/O latency, and Figure 3(b) compares two Verkle settings: removing I/O latency only, and removing both I/O latency and elliptic-curve operations. We then compared these configurations to estimate the relative contribution of each factor to the overall computation, thereby identifying the performance bottlenecks of the corresponding ADS.

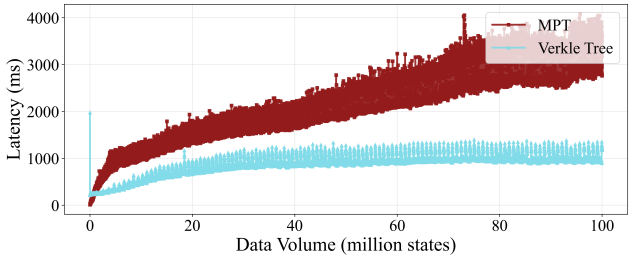
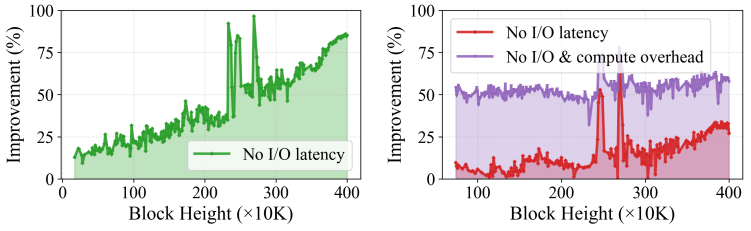


Fig. 2. End-to-end processing time of MPT, Verkle.



(a) MPT (b) Verkle

Fig. 3. Bottleneck analysis of MPT and Verkle.

Results indicate that MPT is I/O-bound: as block height grows, I/O operations account for up to 60–80%, which aligns with its structural makes multiple disk reads per access. In contrast, Verkle exhibits lower I/O latency (around 25%) due to its 256-ary tree structure, which reduces tree depth; however, its bottleneck shifts to CPU computation, where more than 30% of the time is spent on elliptic curve point operations. Both figures display two sharp spikes, corresponding to periods when Ethereum suffered attacks, which we will discuss in detail in Section 4.1.1.

The above results demonstrate that the commitment latency of the state tree increases continuously with data volume, and that the bottleneck sources differ substantially between the two ADS structures: MPT is limited by I/O, while Verkle is limited by cryptographic computation and I/O. This observation reinforces our design motivation—to decouple block generation from underlying state tree operations via a delayed write mechanism, thereby improving overall system performance regardless of whether the bottleneck originates from I/O or computation.

3 Design

3.1 Overview

We divide the generation of block commitments into three stages: the *execution commitment phase*, the *global state commitment phase*, and the *global state disclosure phase*, as illustrated in Figure 4. This process relies on two key components: a SWMT, which temporarily buffers recent state updates in memory, and a persistent global state tree that stores long-term committed state. Together, they form a two-tier state management structure that supports both high-throughput execution and eventual state finalization.

(1) Execution Commitment Phase. In this phase, the proposer node executes all transactions in the current block $Block_n$, applies the resulting state updates to the SWMT, and generates an SWMT-based state commitment, which is embedded in the block.

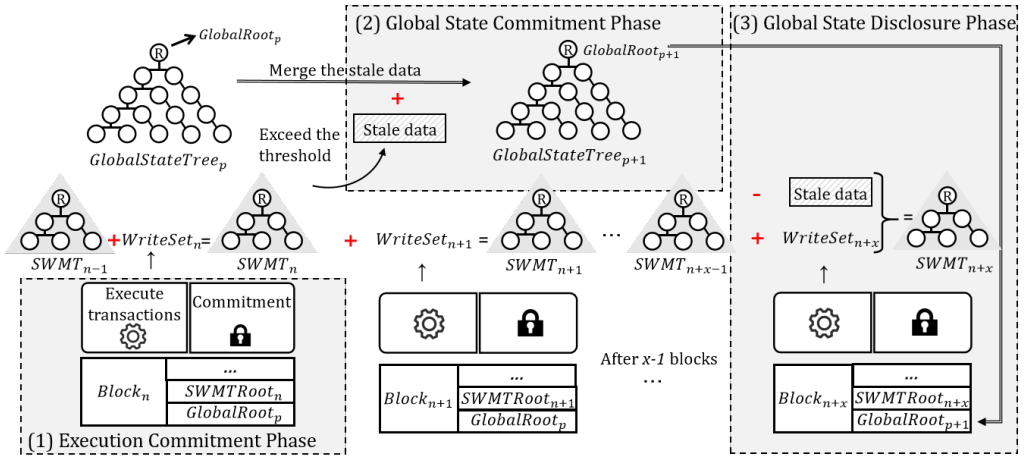


Fig. 4. Three-phase block commitment workflow.

(2) Global State Commitment Phase. After updating the SWMT, if its capacity exceeds a predefined threshold $LN_{\max}$ or available window bits become insufficient, a global state commitment event is triggered. The SWMT selects the least recently updated entries and inserts them into the global state tree. This phase can proceed in parallel with the execution commitment of subsequent blocks, enabling pipeline-style commitment generation. This parallelism stems from the Snapshot-ADS, where each block executes on an immutable snapshot identified by the previous block's state root, ensuring deterministic reads.

(3) Global State Disclosure Phase. Once the global state root is computed, this phase is triggered during the packaging of the next block. At this point, stale entries previously inserted into the global state tree are pruned from the SWMT, and new state updates from the current block are recorded. Both the global state root and the SWMT root are included in the block header.

To avoid race conditions between transaction execution and pruning operations, our design stipulates that pruning is performed only after the current block has been executed. This strategy ensures that the state view remains consistent throughout execution, thereby preventing concurrency conflicts.

3.2 SWMT

SWMT is a verifiable data structure based on the MPT, designed to track the most recent update time of state entries. As shown in Figure 5, it extends traditional MPT nodes by introducing two additional fields: `window` and `size`. The `window` field records the most recent update time of a node, while the `size` field represents the total number of leaf nodes in the corresponding subtree. These two fields serve distinct purposes: `size` helps control the overall size of the tree, while `window` facilitates identifying and pruning stale entries.

SWMT operates under two core protocols: the Sliding Window Protocol defines how the `window` and `size` fields are maintained and ensures that `window` values evolve cyclically over time. The Congestion Control Protocol adjusts the time span that each `window` bit represents, aiming to distribute state entries more evenly across window intervals.

3.2.1 Sliding Window Protocol. We treat block height as a temporal marker for state updates and manage updates using a fixed-size sliding window represented by a 32-bit bitfield. We define a

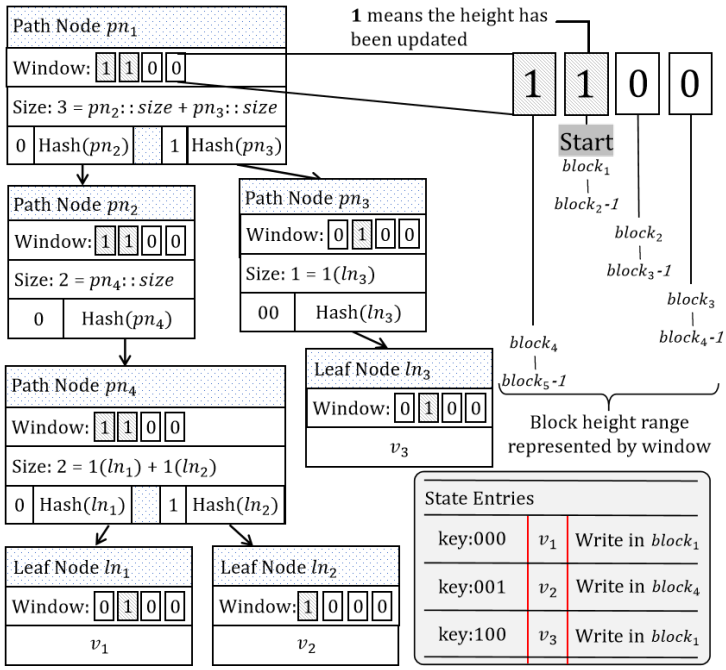


Fig. 5. The SWMT structure.

32-bit cyclic sliding window field $\mathcal{W}_t$ as:

$$\mathcal{W}_t = \langle I_0, I_1, \ldots, I_{31} \rangle, \quad I_i \in \{0, 1\}$$

Each bit I_i represents a contiguous range of block heights. Let $C = \{c_0, c_1, \ldots, c_{31}\}$ denote the number of blocks (i.e., the span) assigned to each bit. The window is cyclically reused with a current starting index f. Let B_0 denote the starting block height corresponding to bit I_f; then the latest representable block height is:

$$B_t = B_0 + \sum_{i=0}^{31} c_i$$

The block height range covered by each window bit is defined as:

$$u_i = B_0 + \sum_{j=0}^{(i-f) \bmod 32} c_{(f+j) \bmod 32}, \quad l_i = u_i - c_i \tag{3}$$

Thus, bit I_i corresponds to the interval $[l_i, u_i)$.

To map a given block height h to its corresponding window bit, we define:

$$\text{CalculateBit}(h) = i^* \quad \text{where } i^* \text{ satisfies } l_{i^*} \le h < u_{i^*} \tag{4}$$

State Insertion: Each tree node n in the SWMT maintains a 32-bit window field $w(n) \in \{0, 1\}^{32}$, represented as unsigned integer. The i-th bit indicates whether a state update occurred during interval I_i:

$$w_i(n) = \begin{cases} 1, & \text{if an update to } n \text{ occurred in } [l_i, u_i) \\ 0, & \text{otherwise} \end{cases}$$

As shown in Algorithm 1, when a state update $\langle k, v \rangle$ (insert, update, or delete) occurs at block height h, the corresponding leaf node resets its `window` field and sets the CalculateBit(h)-th bit to 1. Each leaf node retains only the most recently modified bit position, ensuring that the most recently updated states are not pruned. The `window` value is then propagated upward through the tree, with each parent node computing its own `window` field as the bitwise OR of its children's `window` fields. For clarity, the algorithm omits the path compression process.

Algorithm 1: Recursive Update with Sliding Window

```
Function Insert(node, path, v, bit):
    if path is empty then
        if node = null then
            node ← new leaf node;
        node.value ← v;
        node.window ← 1 ≪ bit;
        node.size ← 1;
        return node;
    nib ← path[0];
    rest ← path[1 :];
    if node = null then
        node ← new path node;
    child ← node.children[nib];
    node.children[nib] ← Insert(child, rest, v, bit);
    node.window ← ⋁_{c∈node.children}(c.window);
    node.size ← ∑_{c∈node.children}(c.size);
    return node;
Main(k, v, h, rootNode):
bit ← CalculateBit(h);
path ← Split k into a sequence of 4-bit nibbles;
rootNode′ ← Insert(rootNode, path, v, bit) ;
```

Node Pruning: When the SWMT becomes full—either because the current height $h > B_t$ (i.e., the window has completely wrapped around) or the total number of active leaf nodes exceeds the capacity $LN_{\max}$—pruning is initiated from the oldest bit f. As shown in Algorithm 2, the system recursively traverses the trie to delete all leaf nodes that are exclusively marked with this bit, and continues pruning bit by bit in cyclic order. Let f' denote the current pruning frontier (i.e., the most recently processed bit). The pruning process continues until the following condition is satisfied:

$$(f' - f) \bmod 32 \geq 4 \quad \text{and} \quad \text{size} < \delta \cdot LN_{\max} \tag{5}$$

where δ is a buffer ratio (e.g., $\delta = 0.90$) that reserves space for upcoming state insertions after pruning. Once pruning is complete, the bits from f to f' are reclaimed and reused for future block height ranges, enabling cyclic reuse of window space.

3.2.2 Congestion Control Protocol. Since the number of state operations per block naturally fluctuates, using a fixed-size window may cause low cache utilization under light workloads, or oversized windows under heavy load, which in turn increase the pressure on pruning. To adapt to such

Algorithm 2: Recursive Prune by Window Bit

```
Function Prune(node, bit):
    if node.window & (1 ≪ bit) = 0 then
        return node ;
    if node.isLeaf then
        return null ;                                   // Delete leaf
    foreach nib in node.children.keys() do
        if node.children[nib] ≠ null then
            child ← Prune(node.children[nib], bit);
            if child = null then
                Delete nib from node.children;
            else
                node.children[nib] ← child;
    if node.children is empty then
        return null;
    node.window ← ⋁_{c∈node.children}(c.window);
    node.size ← ∑_{c∈node.children}(c.size);
    return node;
Main(f, rootNode):
rootNode′ ← Prune(rootNode, f);
```

variability, we design a congestion-control protocol that dynamically adjusts the window size based on observable cache usage, thereby achieving more balanced cache distribution and improved overall utilization. The protocol integrates classical techniques from TCP congestion control, as follows:

(1) **Slow Start.** Initially, the block span assigned to each window bit increases exponentially—1, 2, 4, . . .—until it reaches a predefined threshold denoted by s_{thresh}.

(2) **Congestion Avoidance.** Once the span exceeds the threshold, it grows linearly, enabling the system to gradually adapt to higher throughput while maintaining control over memory growth.

We define the span allocation function $\text{SpanAlloc}(k)$ as:

$$\text{SpanAlloc}(k) = \begin{cases} 2^k, & \text{if } 2^k < s_{\text{thresh}} \\ s_{\text{thresh}} + (k - \lfloor \log_2 s_{\text{thresh}} \rfloor), & \text{otherwise} \end{cases}$$

(3) **Fast Recovery.** When the active leaf count exceeds $LN_{\max}$, the system enters fast recovery mode. It halves the current threshold:

$$s_{\text{thresh}} \leftarrow \left\lfloor \frac{s_{\text{thresh}}}{2} \right\rfloor \tag{6}$$

Then, it recalculates the span vector for the remaining unused portion of the window using the slow start rule.

Specifically, when pruning occurs at block height h_c, we compute the cutoff bit index as $\hat{i} = \text{CalculateBit}(h_c)$. The span vector is then updated as follows:

$$C'_i = \begin{cases} C_i, & \text{if } i \in \mathcal{P} \\ h_c - l_{\hat{i}}, & \text{if } i = \hat{i} \\ \mathsf{SpanAlloc}((i - \hat{i} - 1) \bmod 32), & \text{else} \end{cases} \quad \text{where } \mathcal{P} = \left\{ i \;\middle|\; \begin{array}{ll} (1)\ f \le i < \hat{i}, & \text{if } f \le \hat{i} \\ (2)\ i < \hat{i} \text{ or } i \ge f, & \text{if } f > \hat{i} \end{array} \right\} \quad (7)$$

This update ensures that all window bits prior to the congestion point remain unchanged, the cutoff bit is resized to precisely cover the block range up to h_c, and the remaining bits are reallocated using the exponential $\mathsf{SpanAlloc}(k)$ strategy.

3.3 Pipelined Commitment

To mitigate the performance bottleneck caused by global state root computation, we adopt a pipelined strategy that enables the parallel execution of global state root generation and SWMT root maintenance. Since the computation of the global state commitment is relatively time-consuming, it is allowed to span multiple blocks and may be disclosed after any subsequent block. However, due to runtime variation across participating nodes, the actual disclosure point can differ, potentially leading to inconsistent blocks being generated at the same height.

To address this, we introduce an interleaving mechanism between the Global State Commitment and Disclosure Phases. Specifically, the first commitment is triggered early using a small threshold to rapidly initialize the commitment pipeline. From the second round onward, a larger threshold is used, and each commitment is coupled with the disclosure of the result from the previous round.

Because block-level state updates are deterministic once a block is finalized, the block that triggers the threshold crossing is also deterministic. Consequently, all participating nodes disclose the same global state root at the same block height, ensuring network-wide consistency.

3.4 Support for Stateless and Light Clients

For Ethereum full nodes, which store the global state, block validation can be performed by executing transactions and verifying the resulting commitment. In contrast, **stateless clients** and **light clients** do not maintain the global state locally and must instead rely on the commitments derived from the global state tree (GST) to validate blocks. We discuss these two client types separately below.

Stateless Clients. Stateless clients do not store any local state; instead, they synchronize only blocks and transactions. Their main responsibility is to verify the correctness of the *read set* required for transaction execution and to ensure that the *write set* is correctly transformed into the resulting state commitment.

SWMT has a bounded size and does not grow indefinitely, making it feasible to store directly on stateless clients. Therefore, a stateless client only needs two types of proofs: (1) Proofs for read set entries that are not present in the SWMT, and (2) Transformation proofs from the pruned SWMT to the global state commitment. Thus, SWMT helps reduce both the proof size and verification workload for stateless clients.

Light Clients. Light clients neither store state nor execute blocks. Instead, they follow the blockchain's progress and trust the execution results provided by full nodes. Light clients are commonly used in scenarios such as cross-chain bridges [6], where it is necessary to verify the existence or non-existence of a transaction or state using commitment-based methods. This can be achieved by validating the target state entry e using the following expression:

$$\begin{aligned} \text{Existence:} \quad & e \in \text{SWMT} \vee (e \notin \text{SWMT} \wedge e \in \text{GST}) \\ \text{Non-existence:} \quad & e \notin \text{SWMT} \wedge e \notin \text{GST} \end{aligned}$$

3.5 Persistent Storage

In Ethereum, there are two types of full-state nodes: **archive nodes** and **full nodes**. Both store the complete global state. The key difference is that archive nodes additionally record historical states and can provide state proofs at any given block height.

Using the MPT as an example, an archive node records the tree nodes of the MPT at each block. Since MPT updates only affect the paths corresponding to the modified keys, the recorded state is *incremental*—i.e., it includes only the nodes affected by changes—rather than a full snapshot. Similarly, SWMT is a modified form of the MPT, and its state update mechanism is also incremental. Therefore, it is feasible to record the SWMT tree nodes for each block.

For full nodes, since the size of SWMT is bounded, the time required to compute the root commitment is relatively small. Hence, unlike archive nodes, there is no need to persist the full tree structure in the database. Instead, the data stored in SWMT can be organized as key-value pairs and periodically checkpointed to disk.

4 Discussion

4.1 Security Analysis

This section analyzes potential security threats that may arise under the SWMT mechanism. We identify a feasible manipulation-based attack on state distribution, termed the **Tsunami Attack**, and analyze its mechanism, mitigation strategies, and impact.

4.1.1 Attack Model and Mechanism. SWMT aims to evenly distribute state entries across all windows. However, this approximately uniform, "sea-level" style structure also introduces a potential attack surface: when a large number of state entries are concentrated in a few windows, it results in a skewed distribution resembling a tsunami-like concentration in certain windows.

An adversary can deliberately craft transactions such that the accessed states are heavily concentrated in specific windows. Since the congestion control protocol only adjusts allocations when SWMT reaches its capacity threshold, it lacks timely responsiveness to sudden localized state writes. This allows an attacker to amplify disparities in state distribution across windows by manipulating transaction inputs.

Furthermore, attackers may exploit vulnerabilities in the transaction fee mechanism to inject a large number of state write operations. Ethereum, for example, employs a gas limit to restrict operations within a single block. In its early history, due to improper gas configuration, attackers were able to create a large number of empty account states at low cost, leading to a state bloat attack (as shown in Section 5.2, the peak after 2.3 million corresponds to this attack). Attackers aim to increase node I/O costs, slow down sync and block propagation, and thereby degrade network availability. Although such vulnerabilities have been mitigated through EIP-150 [7], if similar mechanisms reappear, attackers could again generate massive state writes within a single block, resulting in uneven window distribution or even cache overflow.

4.1.2 Mitigation Strategies. To defend against the *Tsunami Attack*, we introduce two enforceable mechanisms.

Per-Bit Capacity Bound. We cap the number of entries assigned to each window bit with a fixed threshold $\theta \cdot LN_{\max}$ (e.g., $\theta = 0.05$). Once the limit is exceeded, subsequent writes in the next block are redirected to the next available bit. This avoids hotspot accumulation and preserves consistency across nodes.

Per-Block Capacity Bound. The blockchain system can further limit the number of state updates per block to no more than 10% of the maximum capacity of the SWMT, which is approximately equivalent to the amount of data pruned in a single trimming operation. If the write volume

continues to increase, the total cache capacity may be expanded through blockchain governance proposals (such as Ethereum Improvement Proposals, EIPs).

4.1.3 Cost and Impact. At block height 23,290,000 (September 2025), mounting an attack controlling 95% of transactions requires submitting high-fee transactions. The 95th percentile gas price at this block is approximately 3 Gwei (1 Gwei = 10^{-9} ETH). Given a block gas limit of 45 million and a per-state-write cost of 5,000 gas, each block can accommodate about 9,000 state writes. To inject 1 million entries, the attacker must control at least 112 consecutive blocks. Under these conditions, the total cost is estimated to be 15 ETH ($\approx$ \$60,000, assuming an ETH price of \$4,000 in September 2025).

In the worst-case scenario, an attack could force all state in the SWMT to be merged into the underlying state tree, causing block producers to delay block generation or produce empty blocks. However, the system must maintain a merging rate roughly comparable to the write rate; otherwise, even under normal conditions, it would be difficult to sustain the continuous pipelined rhythm of block production and global state commitment. Consequently, the merging delay remains within a time window comparable to that of the injected updates (i.e., 112 blocks). The per-bit capacity bound further reduces this impact. Even if users discover a vulnerability that allows a higher number of state writes within a single block, the system prevents amplification of such extreme write loads via the per-block capacity bound.

It is worth noting that this type of attack—manipulating transaction ordering or suppressing other transactions through the fee mechanism—has long existed in mainstream blockchain systems such as Ethereum and Bitcoin. It is therefore a general limitation rather than a specific vulnerability of the SWMT. In the worst case, SWMT may amplify the observable impact of these existing attacks to a limited extent.

4.1.4 Outlook. The window-based distribution of state writes in SWMT is fully observable and verifiable by all participant nodes, allowing for transparent detection of abnormal write concentration. This property opens the door to enforcing local policies or global consensus rules against adversarial behaviors. To assess the practical relevance of the *Tsunami Attack*, we conduct an experimental simulation in Section 5.5, demonstrating both its potential impact and the effectiveness of SWMT's built-in congestion control.

4.2 Correctness

In this section, we establish correctness by showing that SWMT composed with any Snapshot-ADS preserves its validity.

4.2.1 Proof of SWMT as a Snapshot-ADS. The SWMT, built upon the MPT, introduces a *window* field and a *size* field for both leaf nodes and path nodes. Pruning is triggered when either the window is full or the capacity limit is reached.

Based on Eq. 3 and Eq. 4, and given the predefined system parameters (B_0, f, C), we obtain the following invariant:

Invariant 2. *Given (B_0, f, C), the bit position within the window corresponding to each valid block height is uniquely determined.*

Moreover, block execution is deterministic given identical inputs; that is, the resulting key–value updates are fixed.

Based on the above, we can derive the following proof.

Lemma 4.1. *If pruning and congestion control do not occur, then given (B_0, f, C), the SWMT satisfies the Snapshot-ADS.*

Proof. We prove this by mathematical induction.

Base Case. At the initial state of the system, the SWMT is an empty tree. Since no keys have been written, any read operation returns a default null value. Given that the read state is deterministic, the resulting state transitions produced by block execution are also consistent. Assuming that (B_0, f, C) are fixed, the value of the `window` field for each newly inserted leaf node is uniquely derivable according to Invariant 2, and its `size` field is fixed to 1.

For each path node, its `window` is defined as the bitwise OR of the `window` fields of its children, and its `size` is the sum of the sizes of its children. These properties can be computed recursively in a bottom-up manner. Therefore, all `window` and `size` fields in the tree are deterministically defined.

Furthermore, by Invariant 1, if the value fields of all tree nodes (including `window` and `size`) are deterministic, then the SWMT satisfies the Snapshot-ADS property.

Inductive Step. Assume that the SWMT satisfies the Snapshot-ADS at block height n. Since the read operations in block $n + 1$ are deterministic, the state update set is also deterministic. According to Invariant 2, the window bit corresponding to the current block is uniquely determined. Therefore, for each modified or newly inserted leaf node, the `window` field is deterministic. Similarly, the `window` and `size` fields of all affected path nodes are deterministic.

By the induction hypothesis, the resulting SWMT after processing block $n + 1$ also satisfies the Snapshot-ADS property.

Conclusion. By mathematical induction, the lemma holds. □

Lemma 4.2. *After pruning and congestion control, the updated system parameters (B_0, f, C) are uniquely determined, and the resulting SWMT still satisfies the Snapshot-ADS property.*

Proof. **Base Case.** At the initial state ($B_0 = 0$, $f = 0$, and predefined C), the state update set of each block is deterministic (See Lemma 4.1). Therefore, the block height h_c at which the first pruning or congestion control is triggered is uniquely determined.

According to the pruning algorithm, pruning removes leaf nodes whose corresponding window bits are set to 1 in the bit position indicated by f. Since the `window` field of each existing leaf node is already uniquely determined, the set of pruned leaf nodes is also deterministic. This operation is equivalent to deleting data, and thus the resulting tree still satisfies the Snapshot-ADS property.

As defined in Eq. 5, after pruning, f' is determined. Since the new value B'_0 corresponds to the starting block represented by bit position f', it is therefore determined. If congestion control is triggered, the new congestion threshold s_{thresh} is derived from the previous threshold according to Eq. 6. Since h_c is fixed, Eq. 7 uniquely determines the updated distribution C'. Consequently, the updated parameters B'_0, f', and C' are uniquely determined.

Therefore, after pruning and congestion control, the updated parameters (B_0, f, C) are uniquely determined, and the Snapshot-ADS property of the SWMT remains preserved.

Inductive Step. Assume that after the n-th pruning or congestion control, the system state (B_0, f, C) is fixed, and SWMT satisfies the Snapshot-ADS property. Since the generation of the state-update set remains deterministic, the next block height $h_c^{(n+1)}$ at which pruning or congestion control is triggered is uniquely determined. According to Eqs. 5, 6, and 7, the updated values $B_0^{(n+1)}$, $f^{(n+1)}$, and $C^{(n+1)}$ are uniquely determined. Because pruning only deletes predetermined data, the SWMT continues to satisfy the Snapshot-ADS property.

Conclusion. At any time, the SWMT remains Snapshot-ADS compliant after pruning or congestion control, and (B_0, f, C) remain uniquely determined. The lemma holds. □

Theorem 4.3. *The SWMT with support for both sliding window and congestion control satisfies the definition of Snapshot-ADS.*

Proof. According to Lemma 4.2, after any round of pruning and congestion control, the system state (B_0, f, C) remains uniquely determined, and the tree continues to satisfy the Snapshot-ADS property. Combined with Lemma 4.1, this implies that the SWMT maintains the Snapshot-ADS property throughout its operation. Therefore, the theorem is proved. □

4.2.2 Proof of Compositional Snapshot-ADS Correctness. In system implementation, the SWMT can be combined with any Snapshot-ADS. During write operations, the system first applies updates to the SWMT. After completion, pruning may be triggered according to the policy, and the pruned data are written to the underlying Snapshot-ADS. During read operations, the system first queries the SWMT; if the key is not found, the query continues to the underlying Snapshot-ADS.

Given this priority, let S_{swmt} and S_{base} denote the key-value sets of the SWMT and the underlying Snapshot-ADS, respectively. The combined key-value set S, similar to an *update* operation, can thus be represented as follows:

$$S = S_{\text{base}} \cup S_{\text{swmt}}$$

Theorem 4.4. *The combination of SWMT and any Snapshot-ADS remains a valid Snapshot-ADS.*

Proof. **Verifiability.** For any key, the lookup first occurs in the SWMT. If the key exists, a unique value is returned; otherwise, the query is directed to the underlying Snapshot-ADS, which also returns a unique value. Since both lookups are performed within Snapshot-ADS structures, each query result is accompanied by a verifiable proof. Hence, the combined structure ensures the **Verifiability** property.

Write Consistency. During the write phase, each block's deterministic updates are first written into the SWMT. According to Theorem 4.3, the root commitment of the SWMT is deterministic. By Lemma 4.2, when pruning occurs, the set of pruned keys S_{Pruned} is uniquely determined. These pruned entries are then written into the underlying Snapshot-ADS. Since the underlying ADS also guarantees **Write Consistency**, the resulting root commitment is uniquely determined. Therefore, at any time, both the SWMT and the underlying ADS maintain deterministic key-value sets, and their combined root commitment remains uniquely defined.

Key-Value Semantics. Let S_u denote the set of new writes. According to Eq. 1, the key-value composition satisfies:

$$S_{\text{base}} \cup (S_{\text{swmt}} \cup S_u) = (S_{\text{base}} \cup S_{\text{swmt}}) \cup S_u$$

Thus, writing first to the SWMT is equivalent to writing directly to the combined key-value set.

During pruning, by Lemma 4.2, the pruned leaf node set S_{pruned} is uniquely determined. In one root commitment update, the two layers evolve into $(S_{\text{swmt}} \setminus S_{\text{pruned}})$ and $(S_{\text{base}} \cup S_{\text{pruned}})$, respectively. According to Eq. 2, the overall state after pruning is therefore:

$$\begin{aligned}(S_{\text{base}} \cup S_{\text{pruned}}) \cup (S_{\text{swmt}} \setminus S_{\text{pruned}}) &= S_{\text{base}} \cup [S_{\text{pruned}} \cup (S_{\text{swmt}} \setminus S_{\text{pruned}})] \\ &= S_{\text{base}} \cup S_{\text{swmt}}\end{aligned}$$

Hence, pruning does not alter the combined key-value set. Therefore, the overall system is semantically equivalent to a single key-value store, preserving consistent read-write behavior.

Conclusion. Since the combined structure ensures the **Verifiability**, **Write Consistency**, and **Key-Value Semantics** properties, the integration of SWMT with any Snapshot-ADS continues to satisfy the definition of a Snapshot-ADS. □

5 Evaluation

We evaluate SWMT through large-scale benchmarking, micro-level stress testing, fine-grained internal analysis, and dynamic workload simulations to assess its performance and adaptive efficiency.

5.1 Setup

Our experiments are based on Geth, the most widely adopted Ethereum client. As of version 1.15, Geth has fully implemented both the MPT and the Verkle Tree logic. It also incorporates various optimization techniques, such as parallel tree writes and the flat reader—a mechanism that accelerates state access by maintaining a flattened snapshot of state data and bypassing recursive tree traversal. We extend Geth by introducing a custom delayed write mode, which inserts an additional SWMT layer between the EVM and the underlying state tree storage. The SWMT is implemented as an MPT-based variant and serves as an independent caching and write-control layer. It can operate on top of either MPT or Verkle tree without modifying their underlying structures. We initialize the SWMT with a total capacity of 1,000,000 entries. These values are based on empirical memory constraints of consumer-grade hardware and tuned to balance memory usage and pruning frequency. During each pruning cycle, we retain the last 4 window bits and reserve 10% of the total cache capacity as buffer space.

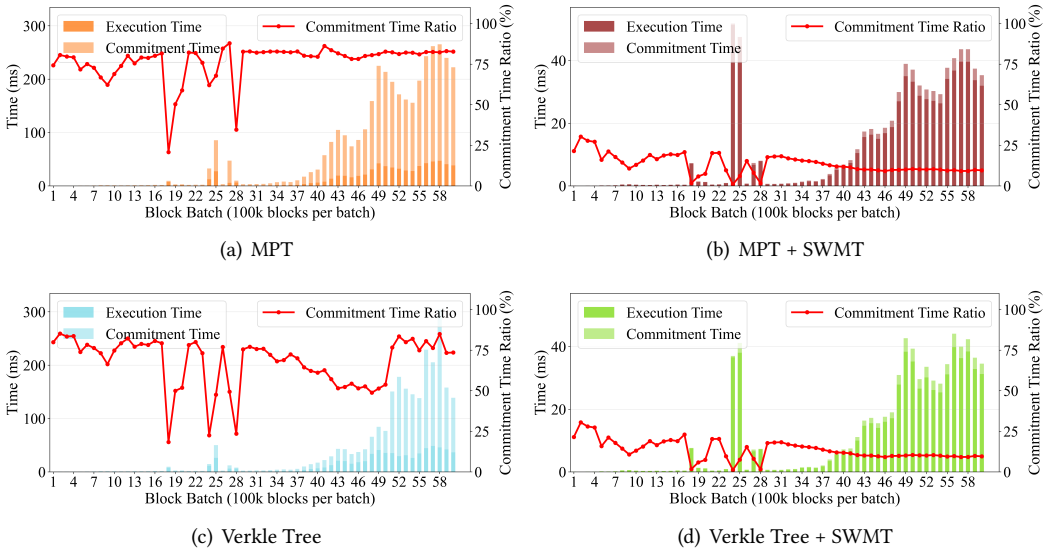
For the experimental environment, we use a consumer-grade host machine equipped with a 12-core processor running at 3.8 GHz and 64 GB of memory.

5.2 Macro Benchmark on Ethereum Mainnet

We conduct a comparative evaluation using 6 million real blocks from the Ethereum mainnet. Our evaluation compares the following four configurations:

(1) the baseline MPT,
(2) the Verkle tree,
(3) the MPT enhanced with the SWMT (MPT+SWMT), and
(4) the Verkle Tree enhanced with the SWMT (Verkle+SWMT).

Each configuration is tested through full synchronization. We record the average transaction execution time and the commitment computation latency, aggregated per 100,000 blocks.



(a) MPT (b) MPT + SWMT

(c) Verkle Tree (d) Verkle Tree + SWMT

Fig. 6. Overall performance.

As shown in Figure 6, the bar chart illustrates the distribution of verification time, while the line graph represents the proportion of time spent on commitment computation. Two significant anomalies are observed in the early stage. The first anomaly between blocks 1.7M and 1.8M corresponds to The DAO attack [32], which targeted a smart contract rather than the Ethereum system itself, causing a surge in transactions and temporary network congestion. The second anomaly, observed between blocks 2.3M and 2.8M, corresponds to the state bloat attack on Ethereum. During this period, the attacker exploited a vulnerability to create numerous empty accounts (as discussed in Section 4.1.1). This is also the primary reason for the significant increase in transaction execution time during that interval.

Excluding these exceptional cases, the ratio of transaction execution time to commitment computation time tends to stabilize. The transaction execution times for MPT and Verkle trees are roughly equivalent, as both utilize a snapshot mechanism to flatten all states into a separate view stored in the database. However, in SWMT mode, the execution times for both structures decrease, since SWMT enables cache hits during reads, thereby reducing state access latency.

Regarding commitment computation time, in the case of the MPT, more than 80% of the time is spent on commitment computation. After incorporating delayed root computation using SWMT, this proportion drops to approximately 10%, even though transaction execution time is also concurrently reduced. While the proportion of commitment time in Verkle trees fluctuates, it eventually stabilizes around 80% as well. Similarly, with delayed root computation, the proportion of commitment time in Verkle trees is also significantly reduced.

For example, between blocks 5.9 million and 6.0 million, the introduction of SWMT reduces MPT commitment time from 184ms to 3ms—a 61.3× reduction. The total processing time decreases from 222ms to 35ms, corresponding to a 6.3× reduction. This demonstrates that using SWMT to delay global state commitment computation offers a clear performance optimization advantage.

Furthermore, we observe that the performance characteristics of MPT and Verkle trees tend to converge when SWMT is applied. This convergence arises because the flattened snapshot mechanism masks the impact of read overhead, while SWMT hides the write-induced performance differences—effectively decoupling overall performance from the choice of underlying tree structure.

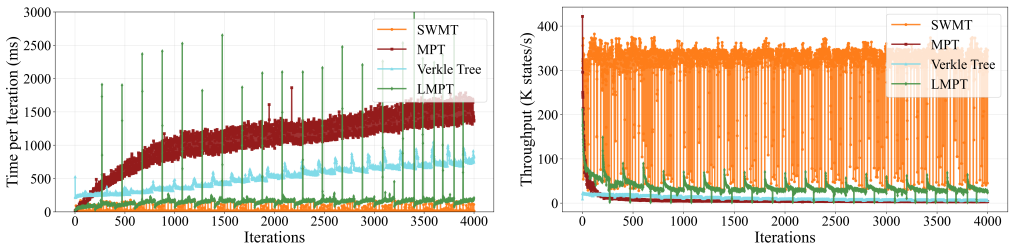
5.3 Micro Benchmark under Randomized Writes

We evaluate the performance of four ADSs — MPT, Verkle, LMPT, and SWMT — through randomized state-writing experiments. The experiment involves a total of 20 million state entries, where latency and throughput are collected every 5,000 writes to accurately measure how performance evolves as the data volume increases. LMPT [8] is a multilayer variant of MPT that introduces a caching mechanism. It employs three functionally separated Merkle trees to handle newly written states, buffered states, and the main state, respectively, thereby enabling high-frequency state updates with progressive merging and data caching. The LMPT evaluation is conducted using a publicly available benchmarking script[1].

We compare the time spent on state insertion and commitment computation for each structure. For SWMT, its automatic pruning time is also included in the total runtime. Figure 7(a) shows the total processing time, while Figure 7(b) presents system throughput in terms of state writes per second.

Experimental results show that the four structures exhibit significant differences in both latency and throughput. SWMT achieves the best overall performance, exhibiting a processing speed tens of times higher than both MPT and Verkle, and significantly outperforming LMPT. This advantage primarily stems from its bounded tree size and fully in-memory commitment computation that

[1]https://github.com/ChenxingLi/authenticated-storage-benchmarks



(a) End-to-end processing time under randomized writes (b) Throughput comparison under randomized writes

Fig. 7. Performance comparison under randomized writes.

avoids any disk I/O overhead. In terms of throughput, SWMT maintains high efficiency throughout the workload, with only performance drops during periodic pruning. This is because pruning involves adjustments to most tree paths, resulting in computational rather than I/O overhead, which can be further optimized through parallelization of tree-path computations.

Because MPT employs path compression, its tree depth increases rapidly under large-scale writes, resulting in noticeable latency growth. In contrast, the Verkle tree exhibits slower latency growth under similar conditions. LMPT demonstrates stage-wise performance fluctuations due to its layered merging mechanism: as the first subtree grows, performance gradually degrades, recovers after merging, and experiences a noticeable degradation peak when the second subtree merges into the third.

5.4 Fine-Grained Analysis of SWMT Behavior

To evaluate the practical effectiveness of SWMT, we conduct a simulation using 6 million real blocks from the Ethereum mainnet, replayed over an MPT-based state model. This setup closely mirrors the behavior of a production environment while allowing fully reproducible measurements. Our analysis focuses on three key aspects of SWMT's behavior: hit rate, cache behavior, and pruning dynamics.

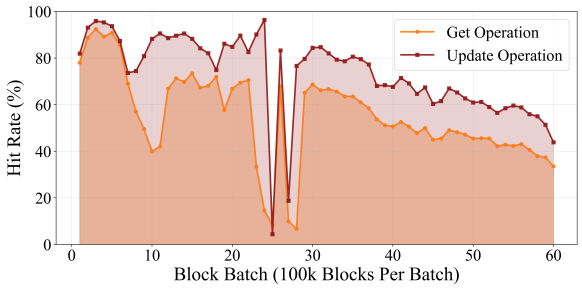
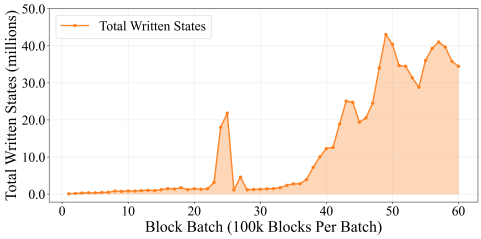


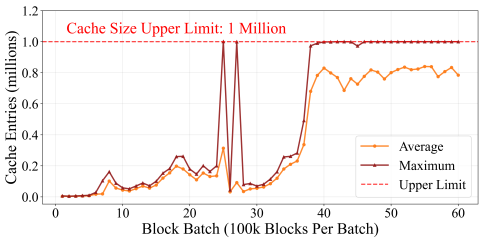
Fig. 8. Cache hit rates for `get` and `update` operations.

5.4.1 Hit Rate. Within SWMT, we measure the hit rates of both `get` and `update` operations, as shown in Figure 8. Apart from a few spikes caused by security attacks, the hit rates remain relatively stable. In most cases, `get` operations achieve a hit rate of around 40%, while `update` operations reach between 40% and 60%.

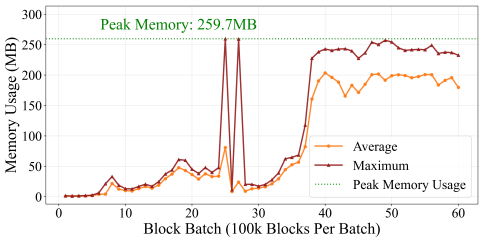
These results indicate that, in real Ethereum transactions, a significant portion of state access operations concentrate on recently updated entries. This locality pattern allows SWMT to effectively reduce the volume of data propagated to the global state.



(a) Total written data

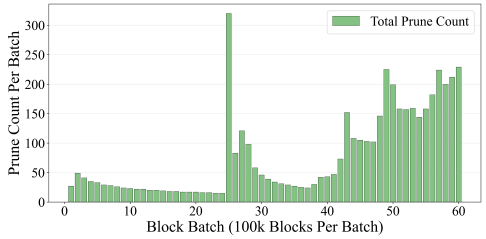


(b) Number of cached entries in SWMT

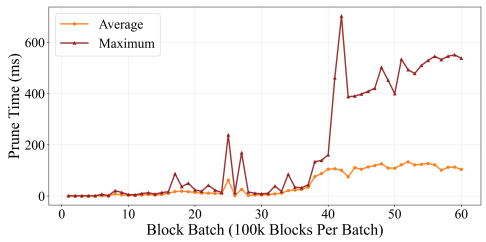


(c) Memory usage of SWMT

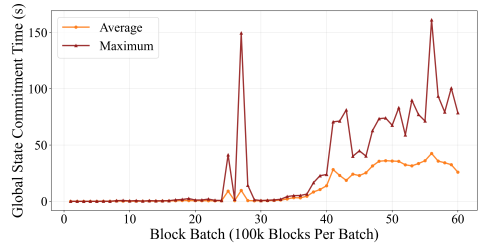
Fig. 9. SWMT caching behavior.



(a) Number of pruning operations



(b) Latency of the global state disclosure phase



(c) Global state commitment time

Fig. 10. Performance of Pruning.

5.4.2 Cached Data Volume and Memory Consumption. We record the total volume of data written, the number of cached state entries in SWMT, and their corresponding memory usage. Figure 9(a) shows the total volume of data written. Figure 9(b) reports the number of cached state entries in SWMT, including both the maximum and average counts. Figure 9(c) illustrates the memory usage of SWMT, also reporting both the maximum and average consumption.

We observe that the number of cached entries and memory usage closely correlate with the total volume of data written. When the data volume is low, both memory consumption and cached entry counts remain small because there are not enough frequently accessed entries to fill SWMT's capacity.

As the data volume increases, however, memory usage and cached entry counts stabilize at a bounded level, with their peaks remaining close to SWMT's designed upper limit. This behavior not only improves data utilization efficiency within SWMT, but also helps mitigate excessive memory

consumption and reduce commitment computation latency. Additionally, we observe that memory usage remains within 300 MB, well within the capacity of a typical consumer-grade machine.

5.4.3 State Pruning. As the data volume stabilizes in later stages, the frequency of pruning operations remains relatively constant, as shown in Figure 10(a).

After pruning is triggered, the global state disclosure phase exhibits an average latency of approximately 100 ms, and in most runs, remains below 600 ms (Figure 10(b)). Although pruning involves significant computational overhead within the SWMT itself, the resulting latency remains within acceptable limits when compared to the commitment latency of underlying structures such as MPT and Verkle trees.

As shown in Figure 10(c), the commitment computation for the global state tree typically completes within 40 seconds, with a maximum of 150 seconds. Based on the observed pruning frequency—approximately once every 500 blocks (roughly every 100 minutes)—based on Ethereum's average block interval of 12 seconds, the system has sufficient time between pruning events to complete the associated global state computations without introducing performance bottlenecks. Even if the block interval decreases to 1 second, the available time remains more than sufficient.

5.5 SWMT Behavior under Dynamic Workloads

To further evaluate the effect of the SWMT congestion window on performance, we simulate a series of dynamic load patterns by randomizing state write operations, and compare their behavior under the **Uncontrolled** mode, where congestion control is disabled. We evaluate multiple load patterns to capture both extreme load conditions and long-term scalability effects.

Based on Section 4.1.3, where the current maximum number of state writes per block is 9,000, we normalize the experimental reference load to 5000 writes per block, representing a realistic and reproducible baseline. In the experimental design, we define 300 blocks (about 1 hour) as a periodic observation unit to capture short-term dynamic behaviors, and continuously run for 7800 blocks (about 26 hours) to assess the system's stability and cumulative state effects during long-term operation. In the Uncontrolled setting, the window size is initialized to a fixed value of 6, which evenly distributes cache resources across 32 windows under 5,000 writes per block, yielding optimal performance.

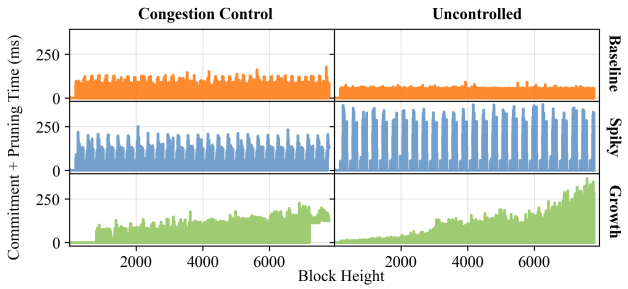


Fig. 11. Per-block commitment time.

We evaluate the performance of SWMT under the following three representative stress models:

Baseline: Each block writes 5000 random state entries. This model is used to assess the benchmark performance of SWMT under long-term, stable, medium-intensity loads.

Spiky: Every 250 blocks, an extreme-load burst lasting 50 blocks (approximately 10 minutes) is injected. During each burst period, the per-block write volume increases to 100,000 operations (the

per-block capacity bound). These bursts emulate attacker-induced state expansion via protocol vulnerabilities (see Section 4.1.4) and test SWMT's resilience and stability under extreme load.

Growth: Starting from 1,000 states, the number of writes increases geometrically until reaching 100,000 states over 7,800 blocks. This model simulates the stepwise growth of user scale and state data, allowing us to observe SWMT's scalability under gradually accumulating state pressure.

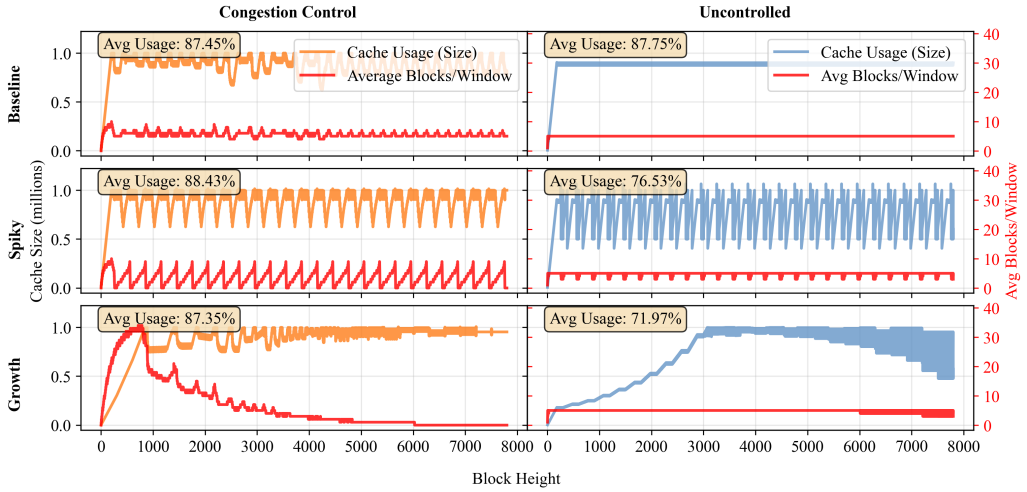


Fig. 12. Cache utilization dynamics across Baseline, Spiky, and Growth scenarios.

Figure 11 illustrates the per-block root computation time trends across different scenarios, reflecting the total latency introduced by state insertion, pruning, and root computation. As shown, under the Baseline scenario, Uncontrolled mode achieves better performance since the window configuration operates under optimal conditions. However, in the Spiky scenario, as the write load increases sharply, latency in the Uncontrolled mode grows significantly higher than that with congestion control enabled. In the Growth scenario, the state pruning time in the Uncontrolled mode continues to rise, whereas with congestion control enabled, the increase remains smoother, demonstrating effective control behavior.

Figure 12 shows the variation of cache utilization and the average block count per window under different test scenarios. We also present the average cache utilization.

Overall, in the Baseline scenario, the average utilization of both strategies remains nearly identical (around 87%), indicating that the congestion control mechanism can tune the window size close to the optimal level. However, in the Spiky and Growth scenarios, the curves with congestion control enabled exhibit smaller fluctuations, faster recovery, and higher long-term utilization levels. Specifically, under the Spiky load, congestion control leads to an average improvement of about 11 percentage points, while in the Growth scenario, the utilization remains around 88%, with the advantage further expanding to nearly 16 percentage points.

In contrast, without congestion control, cache utilization tends to drop sharply below 50% during Spiky loads and recovers slowly; Under Growth loads, the system shows lower utilization at small data volumes, and as the dataset grows, increasing pruning pressure leads to secondary degradation, resulting in noticeably lower long-term utilization. These results confirm that SWMT's dynamic window mechanism—inspired by TCP's congestion control principles—can adaptively expand the window under light loads and shrink it under heavy loads, thereby distributing data pressure and improving overall system efficiency.

6 Related Work

State access and commitment have emerged as core bottlenecks in blockchain systems, prompting extensive work at both the client and data structure levels. We review related efforts in two primary directions: client- and storage-level optimizations that enhance I/O and system efficiency, and structural refinements of state trees aimed at improving proof generation and update performance.

Client and Storage-Level Optimizations. Ethereum clients such as Geth [13], Erigon [28], and Reth [25] employ techniques like parallel I/O pipelines and flat state storage to mitigate MPT performance bottlenecks. These enhancements improve performance while maintaining cryptographic consistency with the standard MPT. On the storage side, MoltDB [20] separates current and ancient states in LSM-based databases [23] to accommodate Ethereum's archival requirements. QMDB [36] boosts throughput by writing in parallel to multiple SSDs, yielding near-linear gains—though at the cost of higher hardware demands. However, most client-side systems rely on MPT, leaving commitment latency unaddressed. By contrast, our approach introduces delayed writes to decouple commitment generation from transaction execution. This mechanism can be integrated into existing clients or storage engines without interfering with their low-level optimizations.

Refinements of State Tree Structures. Efforts to improve the scalability and verifiability of blockchain states have also focused on alternative tree structures. Sparse Merkle Trees (SMTs) [12] offer logarithmic-size inclusion and exclusion proofs based on fixed-depth binary trees, making them well-suited for sparse state representations. Merkle Mountain Range (MMR) [29] enables fast append-only operations but lacks support for updates and deletions, limiting its applicability in dynamic state scenarios. Adaptive Merkle Trees [18] reshape tree structures based on access frequencies to reduce both the average proof size and the computational overhead of commitment.

Other approaches introduce cryptographic commitments to enhance throughput and proof succinctness. Verkle trees [17] leverage KZG commitments [16] in a 256-ary structure and use inner product arguments to produce constant-size proofs. Binary Merkle Trees [5] are proposed as an alternative to Verkle trees to mitigate potential quantum computing threats [2]. They adjust the branching factor of the MPT and incorporate STARK-friendly hash algorithms [14] to enable zero-knowledge-friendly proof generation. Reckle Trees [24] incorporate recursive SNARKs [10] for batched updatable proofs with logarithmic update time. LVMT [19] uses append-only Merkle trees for state writing to reduce I/O amplification and generates a multi-layer authenticated multipoint evaluation tree (AMT) [30] as the commitment structure. It also leverages versioning to reduce the cost of commitment computation.

Generic ADSs such as Concerto [4] and FastVer [3] employ deferred memory verification to move integrity verification off the critical path and into batch processing, thereby alleviating the concurrency bottleneck caused by tree-node contention during single access operations. However, this approach does not directly apply to blockchains, which must generate a globally verifiable state root for every block. In contrast, LMPT [8] and mLSM [26] aim to reduce tree height by layering multiple MPTs, which, however, results in complex multi-proof verification. Building on these insights, SWMT captures recent state changes within a single-tree structure using a cyclic, transient cache. This not only defers expensive writes but also facilitates fast root generation across blocks. In addition, our TCP-inspired congestion control adaptively adjusts caching granularity based on workload intensity—a form of adaptive control rarely explored in prior state tree designs.

7 Conclusion

We propose SWMT, a cyclic authenticated caching layer that enables delayed commitment in blockchain state management. By decoupling state root generation from block execution, SWMT

alleviates the performance bottleneck associated with commitment computation and I/O overhead. Through its sliding window design and congestion-aware caching granularity, the system adaptively handles variable workloads while maintaining bounded memory usage and pruning costs. Our implementation on top of both MPT and Verkle trees demonstrates that SWMT significantly reduces commitment latency and block processing time. These results validate SWMT as a scalable and practical enhancement to existing blockchain infrastructures, particularly in high-throughput, state-intensive settings.

Acknowledgments

This work was supported by the National Key R&D Program of China (Grant No. 2021YFB2700800), Anhui Province Science and Technology Innovation Project (Grant No. 202423k09020016), and the Open Project of the Anhui Engineering Research Center for Agricultural Product Quality Safety Digital Intelligence (Grant No. FYKFKT24079).

References

[1] Ben Adams, Dankrad Feist, and Maria Inês Silva. 2024. EIP-7782: Reduce Block Latency. https://eips.ethereum.org/EIPS/eip-7782. Accessed: 2025-10-16.

[2] Marcos Allende, Diego López León, Sergio Cerón, Antonio Leal, Adrián Pareja, Marcelo Da Silva, Alejandro Pardo, Duncan Jones, David Worrall, Ben Merriman, Jonathan Gilmore, Nick Kitchener, and Salvador E. Venegas-Andraca. 2023. Quantum-resistance in blockchain networks. *Scientific Reports* 13 (2023), 5664. doi:10.1038/s41598-023-32701-6

[3] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath Setty, Nikhil Swamy, Alexander van Renen, and Min Xu. 2021. FastVer: Making Data Integrity a Commodity. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 89–101. doi:10.1145/3448016.3457312

[4] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 251–266. doi:10.1145/3035918.3064030

[5] Guillaume Ballet and Vitalik Buterin. 2020. EIP-3102: Binary trie structure. https://eips.ethereum.org/EIPS/eip-3102. Accessed: 2025-06-27.

[6] Nikita Belenkov, Valerian Callens, Alexandr Murashkin, Kacper Bak, Martin Derka, Jan Gorzny, and Sung-Shine Lee. 2025. SoK: A Review of Cross-Chain Bridge Hacks in 2023. https://arxiv.org/abs/2501.03423. arXiv preprint; Accessed: 2025-07-10.

[7] Vitalik Buterin. 2016. EIP-150: Gas cost changes for IO-heavy operations. https://eips.ethereum.org/EIPS/eip-150. Accessed: 2025-10-23.

[8] Jemin Andrew Choi, Sidi Mohamed Beillahi, Peilun Li, Andreas Veneris, and Fan Long. 2022. LMPTs: Eliminating Storage Bottlenecks for Processing Blockchain Transactions. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, Shanghai, China, 1–9. doi:10.1109/ICBC54727.2022.9805484

[9] CoinMarketCap. 2025. Ethereum (ETH) Market Capitalization. https://coinmarketcap.com/currencies/ethereum/. Accessed: 2025-06-27.

[10] Sai Deng and Bo Du. 2023. zkTree: A Zero-Knowledge Recursion Tree with ZKP Membership Proofs. Cryptology ePrint Archive, Paper 2023/208. https://eprint.iacr.org/2023/208

[11] Ansgar Dietrichs, Tim Beiko, and Marius van der Wijden. 2025. Protocol Update 001 – Scale L1. https://blog.ethereum.org/2025/08/05/protocol-update-001. Accessed: 2025-10-16.

[12] Kelvin Fichter. 2019. What's a Sparse Merkle Tree? https://medium.com/@kelvinfichter/whats-a-sparse-merkle-tree-acda70aeb837. Accessed: 2025-06-27.

[13] Ethereum Foundation. 2025. go-ethereum. https://github.com/ethereum/go-ethereum. Accessed: 2025-06-27.

[14] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Berkeley, CA, USA, 519–535. https://www.usenix.org/conference/usenixsecurity21/presentation/grassi

[15] Amber Group. 2025. Yield-Bearing Stablecoins: The Convergence of TradFi and DeFi. https://ambergroup.medium.com/yield-bearing-stablecoins-the-convergence-of-tradfi-and-defi-9f37d0cab327. Accessed: 2025-07-10.

[16] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. 2010. Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology - ASIACRYPT 2010*, Masayuki Abe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 177–194.

[17] John Kuszmaul. 2018. Verkle Trees. https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf. Accessed: 2025-06-27.

[18] Oleksandr Kuznetsov, Dzianis Kanonik, Alex Rusnak, Anton Yezhov, Oleksandr Domin, and Kateryna Kuznetsova. 2024. Adaptive Merkle trees for enhanced blockchain scalability. *Internet of Things* 27 (2024), 101315. doi:10.1016/j.iot.2024.101315

[19] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. 2023. LVMT: An Efficient Authenticated Storage for Blockchain. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 135–153. https://www.usenix.org/conference/osdi23/presentation/li-chenxing

[20] Junyuan Liang, Wuhui Chen, Zicong Hong, Haogang Zhu, Wangjie Qiu, and Zibin Zheng. 2024. MoltDB: Accelerating Blockchain via Ancient State Segregation. *IEEE Trans. Parallel Distrib. Syst.* 35, 12 (Dec. 2024), 2545–2558. doi:10.1109/TPDS.2024.3467927

[21] Conor McMenamin, Vanesa Daza, Matthias Fitzi, and Padraic O'Donoghue. 2022. FairTraDEX: A Decentralised Exchange Preventing Value Extraction. In *Proceedings of the 2022 ACM CCS Workshop on Decentralized Finance and Security* (Los Angeles, CA, USA) *(DeFi'22)*. Association for Computing Machinery, New York, NY, USA, 39–46. doi:10.1145/3560832.3563439

[22] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf. Accessed: 2025-06-27.

[23] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. doi:10.1007/s002360050048

[24] Charalampos Papamanthou, Shravan Srinivasan, Nicolas Gailly, Ismael Hishon-Rezaizadeh, Andrus Salumets, and Stjepan Golemac. 2024. Reckle Trees: Updatable Merkle Batch Proofs with Applications. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1538–1551. doi:10.1145/3658644.3670354

[25] Paradigm. 2022. Reth: Fast and Modular Ethereum Implementation in Rust. https://github.com/paradigmxyz/reth. Accessed: 2025-06-27.

[26] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. 2018. mLSM: making authenticated storage faster in ethereum. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage'18)*. USENIX Association, USA, 10.

[27] Roberto Tamassia. 2003. Authenticated Data Structures. In *Algorithms - ESA 2003*, Giuseppe Di Battista and Uri Zwick (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–5.

[28] Erigon Team. 2025. Erigon. https://erigon.tech/. Accessed: 2025-06-27.

[29] Kevin Todd. 2025. Merkle Mountain Ranges. https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md. Accessed: 2025-06-27.

[30] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. 2020. Towards Scalable Threshold Cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 877–893. doi:10.1109/SP40000.2020.00059

[31] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. 2021. Non-Fungible Token (NFT): Overview, Evaluation, Opportunities and Challenges. https://doi.org/10.48550/arXiv.2105.07447. arXiv preprint; Accessed: 2025-06-27.

[32] Wikipedia contributors. 2025. The DAO. https://en.wikipedia.org/wiki/The_DAO. Accessed: 2025-06-27.

[33] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf. Accessed: 2025-06-27.

[34] Liam Wright. 2024. Ethereum DeFi's TVL Reaches 2-Year High of $80 Billion, Reclaims USDT Dominance. https://cryptoslate.com/insights/ethereum-defis-tvl-reach-2-year-high-of-80-billion-reclaims-usdt-dominance/. Accessed: 2025-06-27.

[35] Ning Xia, Xiaolei Zhao, Yimin Yang, Yixuan Li, and Yucong Li. 2025. Exploration on Real World Assets and Tokenization. https://arxiv.org/abs/2503.01111. arXiv preprint; accessed: 2025-06-27.

[36] Isaac Zhang, Ryan Zarick, Daniel Wong, Thomas Kim, Bryan Pellegrino, Mignon Li, and Kelvin Wong. 2025. QMDB: Quick Merkle Database. https://arxiv.org/abs/2501.05262. arXiv preprint; Accessed: 2025-06-27.

Received July 2025; revised October 2025; accepted November 2025