

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

MHOT: Height-Optimized Authenticated Data Structure for Blockchain State Commitment

Sipeng Xie, Qianhong Wu, Minghang Li, and Qiyuan Gao, *Beihang University;* Bo Qin, *Renmin University of China;* Qin Wang, *Independent*

https://www.usenix.org/conference/usenixsecurity26/presentation/xie

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

MHOT: Height-Optimized Authenticated Data Structure for Blockchain State Commitment

Sipeng Xie[*] Qianhong Wu[*] Minghang Li[*] Qiyuan Gao[*] Bo Qin[†] Qin Wang[‡]

Abstract

State root computation dominates (~78%) blockchain block processing time. Ethereum's canonical authenticated data structure, i.e., Merkle Patricia Trie (MPT), suffers from severe tree-height growth and is vulnerable to *Nurgle attacks* (S&P'24), where adversaries inflate path depth via hash collisions and degrade system performance at negligible cost. Existing defenses increase node fanout (span) to bound tree height, but higher span inflates proof size exponentially. Prior work mitigates this trade-off using vector commitments, at the cost of trusted setup or expensive verification.

We present MHOT, a height-optimal authenticated data structure for blockchain state commitment that preserves standard hash-based verification without trusted setup. Unlike MPT's fixed-prefix indexing, which couples span and fanout exponentially, MHOT indexes by *discriminative bits* that actually distinguish keys, achieving adaptive span with linear fanout coupling and provably minimal height. To prevent high fanout from inflating proofs, we introduce *hierarchical proofs*, a two-layer Merkle construction that reduces per-node proof overhead from $O(k)$ to $O(\log k)$.

On Ethereum mainnet workloads, MHOT achieves up to **9**× higher write throughput, **4**× lower write amplification, and **2**× smaller proofs than MPT. Under Nurgle attacks, even when the adversary consumes an entire block's gas budget, MHOT maintains a **0%** attack success rate (v.s., 99.97% for MPT). Our results, somewhat surprisingly, show that height optimality (not new crypto primitives!) is the key abstraction for scalable and attack-resilient blockchain state commitment.

1 Introduction

State root computation consumes 70–80% of total block processing time in modern blockchain systems [18, 20, 31, 40, 51]. Every block must commit to a cryptographic digest of the entire state so that light clients can verify query results without trusting full nodes. This authenticated commitment step, not transaction execution, has become the binding constraint on end-to-end block throughput and confirmation latency.

Execution-layer optimizations reduced the cost of everything *except* commitment. Transaction-level parallelism [27, 28, 41, 49], instruction-level acceleration [17, 32, 46, 59, 60], and decoupled state storage architectures [13, 18, 24, 47, 51, 54, 57] all improve execution throughput but cannot amortize the cost of updating the *authenticated data structure* (ADS), because commitment requires rehashing dependent node paths. Modern execution clients such as Erigon and Reth accordingly treat the ADS as a dedicated commitment engine [24, 47], making its write throughput the primary bottleneck.

Limitations of current approaches (Table 1). The Merkle Patricia Trie (MPT), Ethereum's canonical authenticated state structure [55], stores state in a Merkle-authenticated key-value trie. Lookups and updates follow the hexadecimal digits of a key from the root to a leaf, so each update rehashes every node on that path. As Ethereum's state has grown, average path depth has reached 8 to 11 levels [18]; the Nurgle attack [31] exploits the same prefix structure by choosing keys with long common prefixes and pushing selected paths toward the worst case. Partitioning, checkpointing, and in-memory upper-level caching, as in Chainspace [3], RainBlock [48], and LMPTs [20], improve scalability and average-case latency, but they preserve prefix-based traversal and leave worst-case height and Nurgle resistance unresolved.

The limitation is structural. In a prefix-based trie, each tree level consumes a fixed number of contiguous key bits. We call this per-level bit-width the *span* (s). A node that consumes s bits must provide a child slot for every possible s-bit pattern; we call this number of child slots the *fanout*. In a prefix-based trie, fanout $= 2^{\text{span}}$, so the two quantities are coupled **exponentially**.

The apparent defense against adversarially shared prefixes is to increase the span so that each level consumes more key bits. However, large span inflates both node size and membership proofs, which must include sibling hashes at every level. For example, bounding the worst-case depth to at most 10 for

[*]Beihang University
[†]Renmin University of China. Corresponding author.
[‡]Independent

Table 1: Comparison of ADS approaches for uniformly random 256-bit keys (e.g., Keccak-256 derived storage keys in Ethereum).

Approach	Crypto Primitive	Optimization	Height	Nurgle Resistance	Range Proof for Trie Key
MPT [55]	Standard Hash	Baseline	$\leq \max(64, O(\log_{16} N))$	✗ (low span)	✓ $[O(m+\log_{16} N)]$
RainBlock [48]	Standard Hash	Storage (DSM-Tree)	$\leq \max(64, O(\log_{16} N))$	✗ (low span)	✓ $[O(m+\log_{16} N)]$
LMPTs [20]	Standard Hash	Storage (Mem/Disk Layering)	$\leq \max(64, O(\log_{16} N))$	✗ (low span)	✓ $[O(m+\log_{16} N)]$
Prefix MPT [18]	Standard Hash	Storage (Sequential Locality)	$\leq \max(64, O(\log_{16} N))$	✗ (low span)	✓ $[O(m+\log_{16} N)]$
DMM-Trie [51]	Standard Hash	Storage (Delta-encoded)	$\leq \max(64, O(\log_{16} N))$	✗ (low span)	✓ $[O(m+\log_{16} N)]$
Unified Binary [9]	Standard Hash	SNARK-Friendly (Binary)	$\leq \max(256, O(\log_2 N))$	✗ (low span)	✓ $[O(m+\log_2 N)]$
LVMT [40]	Vector Comm.	VC-based (High-span + HMT)	$\leq \max(16, O(\log_{65536} N)) + O(\log_2(\text{Epoch} \times \Delta))$	✓ (high span)	Not Native
Verkle Trie [37]	Vector Comm.	VC-based (High-span + stem)	$\leq \max(32, O(\log_{256} N))$	✓ (high span)	Not Native
MHOT	**Standard Hash**	**Cross-layer (Adaptive Span + Bit-indexing + SIMD)**	$\mathbf{\leq \max(\lceil \frac{256}{k-1} \rceil, O(\log_k N))}$	✓ **(adaptive span)**	✓ $\mathbf{[O(m+\log_k N)]}$

256-bit keys requires $s \geq 26$, resulting in roughly 67 million child slots per node. This *span–proof trade-off* implies that no prefix-based scheme can simultaneously guarantee bounded worst-case height and compact membership proofs.

Vector-commitment alternatives (e.g., Verkle tries [37], LVMT [40]) avoid this trade-off by algebraically decoupling proof size from fanout. However, both rely on trusted setup ceremonies and pairing-based elliptic-curve commitments, introducing trust assumptions and higher arithmetic costs that hash-based schemes do not require. Moreover, neither provides native support for range proofs, which are essential for verified state synchronization.

Our approach: height optimality without cryptography. The key observation behind our approach is that prefix indexing causes the *span–proof trade-off*. This observation motivates our use of Height-Optimized Tries (HOT) [6] as the structural foundation.

HOT branches on *discriminative bits* rather than on fixed prefixes. A discriminative bit is a bit position where at least two keys in the current subtree differ. Prefix-based indexing consumes bits at fixed positions regardless of whether they distinguish any keys; discriminative-bit indexing consumes a bit only where keys actually diverge. This difference lets HOT achieve provably minimal height among radix tries for any given key set. Discriminative-bit indexing couples span and fanout *linearly*, since a compound node with fanout k consumes up to $k-1$ discriminative bits, compared to the $\log_2 k$ contiguous bits resolved by a prefix-based node of the same fanout. This linear coupling gives compound nodes enough span to absorb adversarial insertions without immediate depth growth. When restructuring is needed, HOT's structure-adapting insertion algorithm places the new key to preserve the height-optimized invariant. These properties prevent long shared prefixes from directly turning into long root-to-leaf paths.

MHOT inherits HOT's wide compound nodes, but this may create a new authentication cost. A direct hash-based proof for a child would include all $k-1$ sibling hashes inside the compound node, so proof size grows linearly with fanout and, in MHOT, with span. We address this with *hierarchical proof*, a two-layer Merkle architecture where each compound node maintains an internal Merkle tree over its children. Proving membership of a child then requires only $O(\log k)$ sibling hashes from this internal tree. Thus, proof size grows only logarithmically with MHOT's fanout and, due to HOT's linear span–fanout coupling, only logarithmically with its realized span. This is different from a prefix-indexed trie: because its fanout grows as 2^{span}, the same internal Merkle tree would still leave proof size linear in span.

Another challenge is that HOT was originally designed for in-memory indexing and does not support persistence or authentication [6]. The original work explicitly identifies disk-based deployment as an open challenge, since discriminative-bit tracking breaks the prefix locality relied upon by traditional persistent tries. Adapting HOT to MHOT therefore requires preserving its structural invariants while adding content-addressable persistence and cryptographically binding state roots. The search and insertion logic must remain intact, and membership, multipoint, and range proofs must be generated and verified against those roots.

We resolve these limitations in MHOT by making HOT-style nodes content-addressable, so node identifiers and state roots follow deterministically from the trie structure. This design makes MHOT persistent and cryptographically binding, while a parallel height-stratified commit pipeline and an LSM-tree-friendly layout control commitment and storage costs.

Our contribution. We present MHOT, an instantiation of HOT that supports both persistence and authenticated commitment, composing naturally with content-addressable, copy-on-write storage backends. Beyond throughput, MHOT's compound node design provides structural resistance to Nurgle attacks. Even when an adversary controls an entire block's gas budget, prefix collisions are absorbed through internal restructuring without propagating depth increases.

Evaluation under Ethereum mainnet workloads shows MHOT achieves up to 9× higher write throughput and 4× lower write amplification than MPT-based implementations, with 2× smaller proofs. Under Nurgle attack conditions, MHOT exhibits a 0% attack success rate, in contrast to the 99.97% success rate observed for MPT.

In short, we make the following contributions:

- We identify span-proof coupling as the structural reason prefix-based authenticated tries cannot easily combine low worst-case height with compact proofs. MHOT

avoids this coupling through HOT's linear span-fanout relation and therefore uses only standard hash commitments rather than relying on vector commitments.
- We adapt HOT to blockchain state commitment, presenting MHOT as a height-optimized ADS that resists adversarial key distributions through discriminative-bit tracking and high-fanout compound nodes.
- We design *hierarchical proofs*, a two-layer Merkle construction that reduces per-node proof overhead from $O(k)$ to $O(\log k)$, breaking the linear dependence of proof size on span.
- We resolve the open problem identified in the original HOT work [6] on disk-based deployment through content-addressable indexing, a parallel height-stratified commit pipeline, and an LSM-tree-friendly layout for persistent keys.
- We benchmark MHOT against MPT, RainBlock, and LVMT under Ethereum mainnet workloads, demonstrating up to 9× higher throughput and 2× smaller proofs.

2 Why Existing Solutions Fail

Each ADS existing approach (Table 1) sacrifices at least one of listed properties: acceptable worst-case height, Nurgle resistance, transparent setup, or native range proofs. The root cause is prefix-based indexing. We dig into it.

2.1 The MPT Limitation

Ethereum stores its state in a Merkle Patricia Trie (MPT). To locate a value, the trie reads the key one hex digit at a time and follows the matching child pointer downward. When two keys share a prefix, they ride the same path until their digits diverge, at which point the trie forks. Branches that only a single key traverses are not kept as individual levels. Instead, the whole stretch is compressed into one node that remembers the shared prefix. Values sit at the leaves. Formally, the forking nodes are branch nodes, the compressed stretches are extension nodes, and the value-bearing endpoints are leaf nodes. This design, a Patricia-compressed radix trie, strips out unary chains but preserves the fixed-prefix indexing rule. Every update walks a **root-to-leaf** path, so changing any entry recomputes hashes along that whole path.

Ethereum implements this trie with content-addressed, copy-on-write nodes. Each materialized node is identified by the hash of its serialized content. Modifying a key-value pair creates a new leaf representation and recomputes hashes along the path from that leaf to the root (Figure 1). For a realized path of depth d, one state modification requires $O(d)$ hash computations and $O(d)$ node writes or lookups. Measurements show that average MPT depth on Ethereum mainnet has grown from 8 to 11 levels as state size increases. A single block now triggers over 8,700 disk I/O operations [18].

This depth-dependent performance creates a denial-of-service (DoS) vector. The Nurgle attack [31] exploits the MPT's prefix-based indexing and Patricia compression to deepen realized trie paths. Attackers search for keys whose hashed trie indices share long prefixes with existing keys. Because the MPT routes keys by prefix, these keys traverse the same trie region before diverging. When such a key is inserted into a region where Patricia compression has collapsed a long shared prefix into one extension node, the insertion forces that node to split. A compressed path is materialized into additional branch and extension nodes, increasing the realized root-to-leaf depth by 1–2 levels per insertion. Later accesses to the affected keys require more node traversals, hash verifications, and disk lookups. Because Ethereum's gas pricing charges per opcode rather than per node traversed [31], the attacker pays almost nothing while validators absorb the I/O cost of the deepened paths. Current mitigations, including historical data pruning [34] and gas repricing [5], address symptoms rather than the structural root cause.

2.2 The Span–Proof Trade-off

Storage and system-level mitigations. Several approaches reduce authenticated-state cost without changing the local indexing rule. RainBlock [48] decouples storage from consensus through sharded in-memory state. LMPTs [20] keep recent-update tries in memory and store larger snapshot tries on disk. Chainspace [3] splits smart-contract state across shards, builds commitments for shard-local state and history, and certifies those commitments through shard-level quorum signatures. ChainKV [18] exploits sequential key locality through prefix-based storage. Letus [51] uses delta-encoded state representation.

These mitigations improve normal-case performance and can reduce the short-term cost of adversarial accesses, but they do not remove the local depth mechanism exploited by Nurgle [31]. If a shard, checkpoint, or cache-backed component still uses a fixed-prefix trie, an adversary can target keys whose distinguishing prefixes fall below the cached or partitioned boundary. The attack surface narrows or shifts, but the local tree still routes by fixed prefixes.

With a fixed 16-ary branching factor, each node consumes 4 prefix bits per level, so 256-bit keys have worst-case depth 64. Storage and system-level mitigations therefore complement, rather than replace, a local authenticated structure whose height resists adversarial prefix construction.

The exponential coupling. Prefix-based indexing couples span and fanout exponentially (§1), making high-span nodes intractable. The exponential blowup extends to proofs and commitment computation. A naive membership proof must include all sibling hashes at each tree level, yielding $O(d \cdot 2^s)$ hash values in the proof. Hierarchical hashing reduces this to $O(d \cdot s)$ hashes, but each node still requires $O(2^s)$ hash

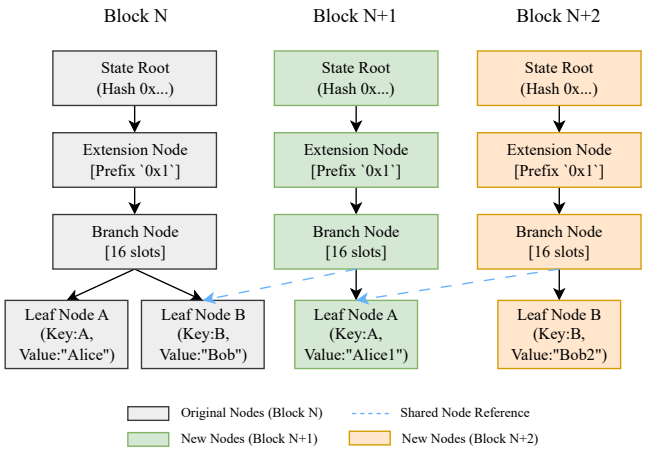


Figure 1: Cross-block node sharing in MPT under copy-on-write semantics. When accounts are modified, only affected paths change. Unchanged subtrees remain shared. Such append-only structure requires $O(d)$ writes per modification, where d is the path length from leaf to root.

computations during commitment. Prefix-based indexing thus forces a choice: low depth with impractically large fanout, or practical node sizes with deep trees vulnerable to attack.

Cryptographic approaches. Vector-commitment schemes sidestep this trade-off by decoupling proof size from fanout through algebraic techniques. Verkle trees [37] use inner-product arguments to achieve $O(1)$ proof size per node with 256-ary fanout. LVMT [40] extends this principle to 2^{16}-ary nodes via KZG polynomial commitments [35], achieving $O(1)$ updates through version-value separation.

Both Verkle trees and LVMT require trusted setup ceremonies whose compromise would enable proof forgery. Their pairing-based verification also imposes higher costs on resource-constrained light clients than standard hashing does. LVMT inherits the Authenticated Multipoint Evaluation Tree's in-place update model and thus cannot support historical state queries [40, 53]. Neither natively supports range proofs, proofs that all keys within a specified interval satisfy certain properties, a capability essential for state synchronization protocols.

Prefix-based indexing is the shared root cause of both camps' limitations. Storage-layer optimizations preserve transparency and range proofs but inherit Nurgle vulnerability because their low span cannot absorb adversarial insertions. Cryptographic solutions achieve Nurgle resistance through high span but sacrifice transparency and range proofs. Achieving attack-resistant depth without these trade-offs requires an indexing paradigm that decouples span from fanout.

3 Key Primitives

3.1 Authenticated Data Structures

An authenticated data structure (ADS) allows an untrusted prover to certify data-operation correctness without requiring the verifier to possess the entire dataset [43]. Formally, an ADS provides three core primitives.

- **Commit.** Given a dataset D, produce a short cryptographic digest C that binds to D. In blockchain terminology, C corresponds to the *state root* stored in block headers.
- **Prove.** Given a query q and its result a, generate a proof π demonstrating that a is the correct answer for q under the committed dataset.
- **Verify.** Given the commitment C, query q, claimed result a, and proof π, compute $\texttt{Verify}(C,q,a,\pi) \in \{0,1\}$ to determine whether a is authentic.

A secure ADS has two properties. *Computational binding* makes proof forgery computationally infeasible. *Succinctness* bounds proof size $|\pi|$ and verification time to $O(\log|D|)$, with some constructions achieving $O(1)$ for both [35, 37].

Consensus protocols agree only on block headers containing the state root, not the full state. Blockchains therefore rely on ADS to make these compact commitments verifiable, enabling light clients to verify query responses against the block header rather than trusting RPC providers.

Modern execution clients (e.g., Erigon [24], Reth [47]) decouple plain state storage from ADS. The underlying database handles execution-layer I/O directly, while the client propagates state updates to ADS for commitment maintenance. Under this architecture, the ADS functions as a dedicated *commitment engine*. Its primary performance metric is *write throughput*, measured as batch updates committed per second.

3.2 Height-Optimized Trie

HOT [6] breaks exponential coupling (§2.2) by branching on *discriminative bits* rather than contiguous prefixes. Given a set of keys K within a subtree, a bit position i is discriminative if at least two keys in K differ at that position. The discriminative bit positions of K are

$$D(K) = \{\, i \mid \exists k_a, k_b \in K,\ \mathrm{bit}_i(k_a) \neq \mathrm{bit}_i(k_b) \,\}.$$

Prefix-based indexing, in contrast, consumes a fixed block of contiguous bits at each level regardless of whether those bits distinguish any keys.

Discriminative-bit indexing couples span and fanout *linearly*. A compound node with fanout k consumes up to $k-1$ discriminative bits at arbitrary positions. A prefix-based node with the same fanout resolves only $\log_2 k$ contiguous bits per level, requiring far more levels to cover the full key. Concretely, a compound node with $k = 27$ entries consumes up to 26 discriminative bits, well within practical limits.

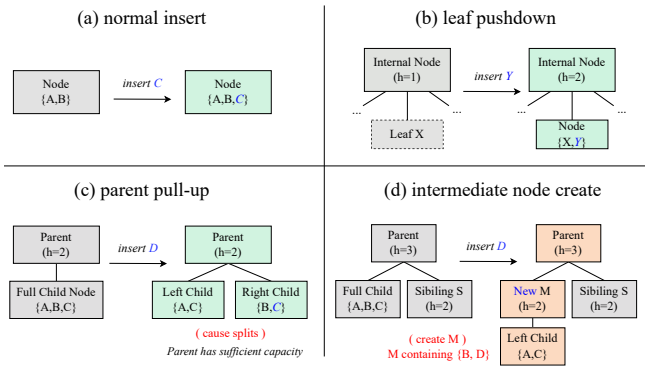


Figure 2: HOT insertion mechanisms with fanout $k = 3$. (a) *Normal Insert*: the new key is added when the node has capacity. (b) *Leaf pushdown*: a collision creates a new child node containing both entries. (c) *Parent pull-up*: overflow propagates upward when $h_{\text{child}} + 1 = h_{\text{parent}}$; this is the only path that increases global tree height. (d) *Intermediate node creation*: when $h_{\text{child}} + 1 < h_{\text{parent}}$, an intermediate node absorbs the split within the height gap.

Compound nodes aggregate multiple binary decisions into a single node containing up to k entries, where $k = 32$ is typical. Each compound node stores three components. The *extraction mask* identifies discriminative bit positions within this node's scope. The *sparse partial keys* record only the discriminative bits for each entry. The *entry data* holds either a child pointer or a leaf value.

The result is *adaptive span*. In dense key regions, a node covers few bit positions with many entries; in sparse regions, a node spans many positions with few entries. Tree height adapts to actual key distribution rather than following fixed prefix widths. For any key set and fanout parameter k, HOT achieves provably minimal height among radix tries [6]. For uniformly random 256-bit keys with $k = 32$, this yields a theoretical minimum depth of $\lceil 256/31 \rceil = 9$ levels, nearly $6\times$ fewer than the 52 levels required by a prefix-based trie with the same fanout ($\log_2 32 = 5$ bits per level).

Earlier work measures path length in *depth*, while HOT uses *height* following [6]; both count compound-node hops from root to leaf. HOT defines the height of an internal node n as $h(n) = \max_i h(c_i) + 1$, and $h(n) = 0$ for leaves. Unequal child heights create *height gaps*, which allow new nodes to be absorbed without increasing the global tree height.

HOT preserves height optimality via four insertion modes (Figure 2) that adapt the structure based on node height relations [6]. Only *Parent Pull-Up* increases global height, while *Normal Insert*, *Leaf Pushdown*, and *Intermediate Node Creation* perform local restructuring that may lengthen individual paths. §6.6 examines whether such local path increases can be exploited for Nurgle-style attacks.

Binna et al. [6] prove that HOT's dynamic construction produces the same structure as Static Minimum Height Partitioning (SMHP) [36] of the underlying binary Patricia trie. This equivalence guarantees that HOT minimizes compound nodes on any root-to-leaf path.

Structural determinism means that a given key set produces an identical trie structure regardless of insertion order. HOT inherits this property from its underlying Patricia trie representation, since both the Patricia trie and SMHP partitioning are uniquely determined by the key set [6].

These properties establish HOT as the structural basis for blockchain state commitment, but the original design operates in memory and provides no persistence or authentication.

4 Persistent Authentication for HOT

MHOT extends HOT with persistent authentication, enabling their use as a blockchain state commitment engine. We introduce a set of orthogonal extensions that build on top of HOT's core algorithms without modifying them.

We first outline requirements in §4.1. We then present key components: We introduce content-addressable persistence to provide deterministic node identifiers while inheriting cryptographic authentication (§4.2). We design a two-layer Merkle construction that reduces per-node proof overhead from $O(k)$ to $O(\log k)$ (§4.3). We develop a batched commit pipeline that defers and parallelizes hash computation, eliminating redundant work when insertions share ancestors (§4.4). We apply storage-layer optimizations that exploit HOT's structural properties to improve disk I/O efficiency (§4.5).

4.1 Design Requirements

A blockchain authenticated data structure must satisfy five requirements. *Short commit paths* reduce the computation per update, directly affecting block execution time. *Structural determinism* guarantees that identical key sets produce identical tree structures, a prerequisite for validators to agree on state roots. *Trust minimization* avoids trusted third parties, restricting security assumptions to cryptography alone. *Memory efficiency* reduces disk I/O by caching frequently accessed state in memory. *Attack resistance* prevents adversaries from exploiting structural flaws to extend storage paths and exhaust client I/O [4, 8, 11, 12, 31, 33, 55].

Short commit paths and structural determinism follow from HOT's height optimality and greedy partitioning, while memory efficiency derives from adaptive linearized node layout and SIMD-accelerated operations; we validate attack resistance empirically (§6). Two design challenges remain. First, the original HOT's page-based node identifiers are machine-dependent and carry no cryptographic meaning; content-addressable indexing (§4.2) resolves this by deriving each identifier from the hash of its content, satisfying trust minimization without a trusted setup. Second, embedding authentication in high-fanout compound nodes risks inflating proof size; our authentication mechanism targets logarithmic overhead to keep proofs practical (§4.3).

4.2 Content-Addressable Persistence

Indexing HOT nodes demands a custom strategy; approaches that work for conventional tries fail for HOT's compound-node structure. We consider and reject two naive alternatives before arriving at our solution.

Traditional radix tries index nodes by path prefix, but HOT's compound nodes track only discriminative bits without recording complete prefixes, so prefix-based indexing does not apply. One might instead use accumulated discriminative bits as identifiers; however, HOT's insertion algorithm dynamically adjusts which bits are discriminative, causing cascading identifier updates across affected subtrees.

We design MHOT around content-addressable indexing, where each node's database key equals the hash of its serialized content. This scheme exploits HOT's structural determinism (§3.2), which guarantees that identical key sets yield identical tree structures. A node's content depends uniquely on the keys it covers, since extraction masks, sparse partial keys, and child references are all determined by those keys; identical logical nodes therefore produce identical hashes regardless of creation time, so all validators derive the same identifiers for the same state. Hash-based indexing also requires no trusted setup, satisfying the trust minimization requirement.

Like MPT [55], content-addressable storage inherits copy-on-write semantics (see Figure 1), where modifying any node invalidates its hash and all ancestor references. This approach expands child references from the original 4–8-byte pointers to 40-byte identifiers (Figure 3), adding up to ∼1 KB per fully-populated node at $k = 32$. We accept this overhead because uniform, fixed-format references simplify serialization and eliminate branching in traversal code.

MHOT fixes the key length at 256 bits to match the output of Keccak-256 used for Ethereum's storage key derivation. As a result, extraction masks and sparse partial keys have fixed maximum sizes and are stored in fixed-length fields, while only the child reference array grows dynamically with the number of entries. This layout enables simple, single-pass serialization. Figure 3 illustrates the node structure.

4.3 Hierarchical Proof

Each compound node embeds k child hashes, so proof size grows linearly with fanout. A naive inclusion proof must contain all $k-1$ sibling hashes at each level so the verifier can reconstruct parent hashes up to the root, yielding $O(h \cdot k)$ hashes per proof. With 32-byte Keccak-256 hashes, this totals $h(k-1) \cdot 32 \approx 5$ KB for typical parameters ($k = 32$, $h = 5$).

We address this using a two-layer Merkle architecture that trades additional commitment-time computation for compact proofs. Each compound node computes a *Children Merkle Root* (CMR), a binary Merkle root over its k child hashes. Membership proofs then include only $O(\log k)$ binary Merkle siblings per node, instead of all $k-1$ child hashes, reducing

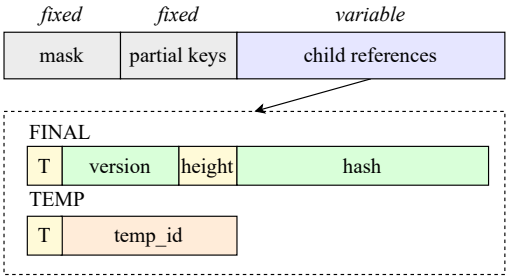


Figure 3: Node layout in MHOT. The top panel shows a compound node with fixed-size metadata and variable-length child array. The bottom panel shows two child reference formats. FINAL references store version, height, and hash; TEMP references store a temporary ID for deferred hashing.

per-node overhead from $O(k)$ to $O(\log k)$. For $k = 32$, this yields roughly a 6× reduction in hash overhead. In practice, single-point proofs are about 1.1–1.4 KB, compared to 2.3–2.9 KB for MPT (§6).

We adopt binary Merkle trees to preserve standard hash-based verification without trusted setup. Although each CMR incurs $O(k)$ hashing cost at commit time, this overhead is absorbed by the batched commit pipeline (§4.4). §5.1 formalizes the construction and extends it to compact multiproofs.

4.4 Batch Commit Pipeline

Each insertion rehashes the entire root-to-leaf path. When multiple insertions share ancestors, naive processing repeatedly rehashes the same *dirty nodes*, i.e., nodes modified since the last commit. We eliminate this redundancy by proposing three optimizations as below.

Deferred hashing. We defer hashing until block commit using *mixed child references*. Each reference stores a type tag distinguishing finalized hashes from temporary IDs. During insertion, dirty nodes receive temporary IDs and enter a pending map. Parents reference these temporary IDs rather than hashes, deferring computation until commit.

Copy-on-write occurs at most once per node per block. Once a node enters the pending map with a temporary ID, subsequent traversals update it in place rather than cloning. When an insertion first modifies a finalized node, we clone it into the pending map (Algorithm 1, lines 11–12); subsequent insertions that traverse the same node find a temporary reference and update it in place without cloning (line 9). The resulting memory overhead is $O(h)$ pending nodes per insertion, where h is tree height.

Height-stratified batch commit. At block boundaries, we finalize all pending nodes in one batched traversal (cf. Algorithm 2). Correct ordering requires processing children before parents. HOT derives this order from each node's height (recall §3.2); since children have strictly lower heights, ascending height order satisfies the dependency. We group

Algorithm 1 Insert with Deferred Hashing

Require: key K, value v, tree T, pending map P
Ensure: A temporary ID referencing the updated root

$P[\text{ALLOC}()] \leftarrow (\text{LEAF}(K,v),0)$; $cid \leftarrow$ allocated ID
$path \leftarrow \langle\rangle$; $r \leftarrow T.root$
while r is internal **do**
  $n \leftarrow \text{GET}(r)$; $s \leftarrow \text{SLOT}(n,K)$ ▷ extract discriminative bits, match partial keys
  $path.\text{PUSH}((n,s,r))$; $r \leftarrow n.ch[s]$
end while
for all $(n,s,r) \in path$ in reverse **do**
  if $r.tag = \text{TEMP}$ **then** ▷ in P: update in place
    $P[r.id].n.ch[s] \leftarrow cid$; $cid \leftarrow r.id$
  else ▷ finalized: clone once
    $n' \leftarrow \text{CLONE}(n)$; $n'.ch[s] \leftarrow cid$
    $P[\text{ALLOC}()] \leftarrow (n', n'.ht)$; $cid \leftarrow$ allocated ID
  end if
end for
return cid

Algorithm 2 Height-Stratified Batch Commit

Require: pending map P: ID $\rightarrow$ (node, height), version v
Ensure: Root reference $(v, hash)$

$F \leftarrow \emptyset$ ▷ temp ID $\rightarrow$ finalized ref
for $h \leftarrow 0$ **to** $\max(P.heights)$ **do**
  for all $(id,n) \in P$ at height h **do in parallel**
    for all c in $n.ch$ **do**
      if $c.tag = \text{TEMP}$ **then** $c \leftarrow F[c.id]$
      end if
    end for
    $F[id] \leftarrow (v, \text{HASH}(\text{SER}(n)))$
  end for
end for
return $F[root_id]$

nodes by height and process from leaves upward, replacing temporary references with finalized hashes at each level.

Parallel execution. Nodes at the same height share no data dependencies, allowing concurrent hash computation within each level. Processing advances to the next height only after synchronizing all hashes at the current height. Correctness follows because temporary references always point to children at strictly lower heights, so each level reads only from entries that prior levels finalized in F (cf. Algorithm 2).

Because HOT records node height as a built-in property, MHOT can use height directly to schedule commit. Nodes at the same height have no Merkle dependencies on one another, so they can be hashed in parallel with one barrier between levels, without fine-grained synchronization or uneven subtree splits. The same height grouping aligns with the storage layout in §4.5, allowing finalized nodes to be flushed in locality-preserving batches. MPT-style tries do not expose this height order directly, so recovering such a schedule requires extra bookkeeping [29].

4.5 Storage Optimizations

Embedding hashes enlarges node footprint; three techniques mitigate this overhead.

Metadata compression. We serialize only populated fields. The extraction mask stores positions of set bits rather than a full 256-bit vector; sparse partial keys use adaptive bit widths. All fields concatenate without internal pointers, so serialization completes in a single pass.

Write ordering. We batch writes across multiple blocks. Each block computes its state root immediately, but nodes accumulate in memory until a configurable flush threshold triggers a single batch write. Each database key consists of a 64-bit prefix followed by the 256-bit content hash, totaling 40 bytes. The prefix encodes a 1-bit type flag distinguishing leaves from internal nodes, a 55-bit version number recording the block of creation, and an 8-bit height value. Version-prefixed ordering groups nodes by creation block, while the height suffix clusters same-height nodes within each block. This layout aligns with MHOT's height-stratified commit and enables sequential batch reads during traversal. Sorting keys before writing yields locality that LSM-tree storage engines exploit for efficient ingestion [18, 45].

Asynchronous persistence. We decouple state root computation from disk I/O by delegating writes to a background thread, avoiding stalls during block execution. When multiple batches complete before a flush finishes, pending writes coalesce into a single disk operation, amortizing `fsync` overhead. The underlying LSM-tree engine's write-ahead log ensures durability; committed data remains recoverable even if a crash precedes the next flush.

The next section formalizes MHOT's proof mechanisms (§5), followed by performance evaluation (§6).

5 Proof Mechanisms

MHOT supports four proof types within a unified framework: *single-point membership* and *non-membership* proofs, *multi-point membership proofs*, *lower bound proofs*, and *range proofs*. This section formalizes each mechanism.

5.1 Two-Layer Merkle Architecture

The naive approach of including all children's hashes directly in each proof node yields proof sizes of $O(h \cdot k)$, where h is tree height and k is fanout. For typical parameters ($k = 32$, $n = 10^8$), this produces approximately 5KB per proof—unacceptable for bandwidth-constrained applications.

MHOT addresses this through a two-layer Merkle architecture. The *inter-node layer* maintains the tree structure connecting compound nodes. The *intra-node layer* introduces a Merkle tree [44] over children within each compound node, enabling logarithmic-size proofs for child membership.

Definition 1 (Children Merkle Root)**.** *For a compound node N with children $C = (c_0, \ldots, c_{|N|-1})$, the children Merkle root is defined as:*

$$\mathrm{CMR}(N) = \mathrm{MerkleRoot}(H(c_0), H(c_1), \ldots, H(c_{|N|-1})) \quad (1)$$

where the Merkle tree is padded to the next power of 2 using a canonical zero hash.

Definition 2 (Node Content Hash)**.** *The node content hash of a compound node N incorporates all structural information:*

$$H_{content}(N) = H(M \parallel S \parallel \mathrm{CMR}(N) \parallel L) \quad (2)$$

where M denotes extraction masks (32 bytes), S denotes sparse partial keys ($|N| \times 4$ bytes), and L denotes child leaf counts ($|N| \times 4$ bytes).

This architecture reduces per-node overhead from $O(k)$ to $O(\log k)$ hashes.

Definition 3 (Node Proof Entry)**.** *For a compound node N and child index set $J \subseteq [0, |N|)$ with $t = |J|$:*

$$\mathrm{NPE}(N, J) = \big(J,\ M,\ S,\ L,\ \eta,\ v,\ \underbrace{\Pi_J^{\mathrm{CMR}}}_{\textit{intra-node proof}}\big) \quad (3)$$

where Π_J^{CMR} is a compact Merkle multiproof for children $\{c_j : j \in J\}$ within $\mathrm{CMR}(N)$, requiring $O(t \cdot (\log k - \log t))$ sibling hashes [7]. When $t = 1$, this reduces to a standard Merkle proof with $O(\log k)$ siblings.

All proof types discussed in the following embed Π_J^{CMR} within each node entry, ensuring proof size scales with $O(\log k)$ rather than $O(k)$ per node.

5.2 Single-Point Proofs

MHOT's single-point proofs rely on the fact that HOT search constitutes an *optimistic search*. Unlike traditional Patricia tries where searching for a non-existent key may fail mid-traversal, HOT's sparse matching semantics guarantee that search always reaches some leaf—though this leaf may differ from the query key. The search only matches discriminative bits at each node; the reached leaf shares all discriminative bit values with the query but is not guaranteed to be lexicographically adjacent. A final comparison between the reached leaf and the query key determines membership.

Lemma 1 (HOT Optimistic Search Invariant)**.** *For any non-empty* HOT *trie T and any query key K, the optimistic search procedure terminates at exactly one leaf node K' that agrees with K on all discriminative bits encountered during traversal, regardless of whether K exists in T.*

Proof sketch (Lemma 1). HOT search at each internal node computes $\mathrm{dense}(K, M)$ and finds the last index j satisfying $(\mathrm{dense} \wedge \mathrm{sparse}[j]) = \mathrm{sparse}[j]$. By the HOT construction invariant, the first sparse partial key is always 0 (corresponding to the leftmost subtree). Since $(\mathrm{dense} \wedge 0) = 0$ holds for all dense values, at least one match exists at every non-empty node. The search thus proceeds deterministically to a unique leaf that matches the query on all discriminative bits. However, non-discriminative bits may differ, so the final leaf K' must be compared against K to determine membership. □

This invariant unifies membership and non-membership proofs as they both execute identical search algorithms, differing only in the final comparison.

Corollary 2 (Proof Unification)**.** *Membership and non-membership proofs share identical proof generation and path verification algorithms. The distinction lies only in the final predicate:*

$$\mathrm{ProofType}(K, K_0) = \begin{cases} \textit{Membership} & \textit{if } K_0 = K \\ \textit{Non-Membership} & \textit{if } K_0 \neq K \end{cases} \quad (4)$$

where K_0 denotes the key of the leaf reached by searching K.

Membership proof. For a key K with associated value V, the membership proof takes the form:

$$\pi_{\mathrm{mem}}(K) = (K, V, v_{\mathrm{leaf}}, \mathrm{Path}) \quad (5)$$

where $\mathrm{Path} = (\mathrm{NPE}(N_0, J_0), \ldots, \mathrm{NPE}(N_{h-1}, J_{h-1}))$ traces the search path from root to leaf, with $|J_i| = 1$ for single-point proofs.

Algorithm 3 presents the verification procedure.

Non-membership proof. For a key K not present in the trie, the non-membership proof includes the neighbor leaf (K', V') reached by executing the same search algorithm. Verification additionally checks that both K and K' route through identical children at every node, ensuring K' is indeed the leaf that HOT search would reach for K.

Theorem 3 (Single-Point Soundness)**.** *If verification succeeds, then with overwhelming probability $1 - \mathrm{negl}(\lambda)$:*

1. *For membership proofs: $(K, V) \in T$*
2. *For non-membership proofs: $K \notin T$*

Proof sketch (Theorem 3). By collision resistance of H, the bottom-up hash reconstruction produces a unique sequence of node hashes. The routing consistency check ensures K traverses exactly the claimed path. For the final hash to match the root commitment, either the path authentically exists or the adversary found a hash collision. The latter occurs with probability at most $\mathrm{negl}(\lambda)$. For non-membership, Lemma 1 guarantees that the optimistic search uniquely determines the leaf K' reached for query K, proving no other leaf could be found by the same search procedure. □

Algorithm 3 Membership Proof Verification

Require: proof π, expected root R
Ensure: TRUE if proof is valid

▷ *Phase 1: Verify routing consistency*
for all entry $\in \pi$.path **do**
  $d \leftarrow$ DENSEKEY(π.key, entry.masks)
  $j \leftarrow$ SEARCHSPARSE(d, entry.sparse_keys)
  if $j \neq$ entry.child_idx **then**
    return FALSE ▷ Routing mismatch
  end if
end for
▷ *Phase 2: Reconstruct hashes bottom-up*
$h_{\text{child}} \leftarrow H_{\text{leaf}}(\pi.\text{key}, \pi.\text{value})$
for all entry $\in \pi$.path.REVERSED() **do**
  cmr $\leftarrow$ RECONSTRUCTCMR(h_{child}, entry)
  $h_{\text{child}} \leftarrow H$(entry.masks $\|$ entry.sparse_keys
    $\|$ cmr $\|$ entry.leaf_counts)
end for
▷ *Phase 3: Compare with expected root*
return $h_{\text{child}} = R$.content_hash

5.3 Multi-Point Membership Proof

A naive approach to verifying m membership proofs requires $O(m \cdot h \cdot (k-1))$ sibling hashes. MHOT reduces this through two optimizations: *path sharing* (keys traversing the same node share that node's metadata) and *compact multiproofs* (proving t children within a node requires $O(t(\log k - \log t))$ sibling hashes rather than $O(t(k-1))$).

Definition 4 (Compact Merkle Multiproof)**.** *Given a Merkle tree T with leaves L and a subset $I \subseteq [0, |L|)$ of indices to prove, a compact multiproof consists of:*

$$\pi_{multi} = (I, \Sigma) \tag{6}$$

where Σ represents the minimal set of sibling hashes needed to reconstruct the root from proven leaves $\{L[i] : i \in I\}$.

Compact multiproofs exploit shared ancestors among the m proven leaves to eliminate redundant sibling hashes, reducing proof size from the naive $m \cdot h$ to as few as $h + m - 1$ when leaves are clustered [7].

The compact multiproof generation algorithm (Algorithm 5 traverses the tree bottom-up, collecting sibling hashes only for nodes where one child is known but the other is not.

Multi-point HOT proof. For a set of keys $K = \{K_1, \ldots, K_m\}$:

$$\pi_{\text{multi}}(K) = (\text{Entries}, \text{Levels}) \tag{7}$$

where Entries contains tuples (K_i, V_i, v_i) sorted by key, and Levels organizes NPEs by depth with compact multiproofs at each node.

Theorem 4 (Multi-Proof Soundness)**.** *If verification succeeds, then all key-value pairs (K_i, V_i) exist in the trie committed by the root, with overwhelming probability.*

Proof sketch (Theorem 4). Each key K_i must pass the same routing consistency and hash reconstruction checks as in single-point verification (Theorem 3). Path sharing and compact multiproofs reduce proof size but do not weaken security: shared nodes are verified once with the same rigor, and multiproofs authenticate the same child hashes as independent proofs would. Forging any (K_i, V_i) requires finding a hash collision. □

5.4 Lower Bound Proof

Lower bound queries—finding the smallest key $\geq Q$—are fundamental to range operations in authenticated storage systems. Unlike membership queries where the search target either exists or does not, lower bound queries must locate a key that may differ from the query itself.

In prefix-organized tries such as MPT, siblings represent lexicographically adjacent key ranges, making neighbor identification straightforward. HOT organizes nodes by discriminative bits instead, so siblings may span non-contiguous ranges. We therefore construct lower bound proofs by authenticating the search path that HOT's bit-comparison traversal follows to locate the smallest key $\geq Q$.

Definition 5 (Lower Bound Query)**.** *For a query key Q, the lower bound operation* $\text{lb}(Q)$ *returns the smallest key $K \geq Q$ present in the trie, or $\perp$ if no such key exists.*

A critical subtlety arises from the nature of optimistic search. The search finds a leaf K' that matches the query Q on all discriminative bits encountered during traversal, but K' need not be lexicographically close to Q—the two may differ arbitrarily on non-discriminative bits. When searching for Q terminates at leaf K' where $K' \neq Q$, let $d = \text{diffbit}(Q, K')$ denote the first differing bit position. Three cases arise:

- *Exact match* ($K' = Q$): The query key exists; K' is trivially the lower bound.
- *Overshot* ($Q[d] = 0, K'[d] = 1$): The search entered a right subtree, so $Q < K'$ lexicographically. However, K' may not be minimal within this subtree; the true lower bound is the leftmost leaf reachable from the fork point.
- *Undershot* ($Q[d] = 1, K'[d] = 0$): The search ended in a left subtree where all keys are less than Q. The algorithm must examine right siblings at the fork point to find a subtree containing keys $\geq Q$.

The lower bound proof must authenticate both the leaf K' reached by optimistic search and the path to the actual result.

Definition 6 (Lower Bound Proof)**.** *For query Q, the lower bound proof structure is:*

$$\pi_{lb}(Q) = (Q, Path, K', V', v', Adj, K_{result}, V_{result}) \tag{8}$$

comprising the query key, the authenticated search path to the leaf (K', V', v') reached by optimistic search, optional adjustment information Adj for non-exact matches, and the actual lower bound result (K_{result}, V_{result}).

When $K' \neq Q$, the adjustment information specifies how to navigate from the search path to the true lower bound:

$$\text{Adj} = (d, b, f, \text{AdjPath}) \tag{9}$$

where $d = \text{diffbit}(Q, K')$ is the first differing bit, $b = Q[d]$ indicates overshot ($b = 0$) or undershot ($b = 1$), f is the fork depth, and AdjPath authenticates the path from the fork point to the result leaf.

Definition 7 (Fork Depth)**.** *The fork depth f identifies where the query's hypothetical path would diverge from the path to the leaf reached by optimistic search:*

$$f = \max\{i : d \in \text{disc_bits}(Path[i])\} \tag{10}$$

where $\text{disc_bits}(\cdot)$ *extracts the set of discriminative bit positions encoded in a node's extraction masks.*

Lemma 5 (Lower Bound Correctness)**.** *The adjustment algorithm correctly computes* $\text{lb}(Q)$.

Proof sketch (Lemma 5). Let $d = \text{diffbit}(Q, K')$ denote the first bit position where the query and the reached leaf differ.

Overshot case ($Q[d] = 0$, $K'[d] = 1$): At fork depth f, the search entered a subtree rooted at a node whose discriminative bit at position d directed the search rightward. All keys K in this subtree share bit $d = 1$, hence satisfy $K > Q$ lexicographically. The minimum key in this subtree (found by repeatedly taking the leftmost child from the fork point) is therefore the smallest key exceeding Q. Note that this minimum need not be K' itself, as K' may reside anywhere within the subtree.

Undershot case ($Q[d] = 1$, $K'[d] = 0$): The search terminated in a left subtree where all keys share bit $d = 0$, hence are lexicographically smaller than Q. The algorithm must find the first right sibling at the fork point whose subtree contains keys $\geq Q$, then descend to that subtree's minimum. □

Verification. The verifier must ensure that the claimed result is indeed the minimum key $\geq Q$, not merely some key satisfying the bound. Four checks enforce correctness:

- *Path integrity.* The search path must authenticate K' against the committed root via bottom-up hash reconstruction.
- *Search consistency*: Both Q and K' must route identically through each path node—i.e., produce the same dense key and thus select the same child—confirming K' is the leaf that optimistic search reaches for Q.
- *Fork depth correctness.* The verifier independently recomputes f from the Merkle-committed extraction masks, rejecting any mismatch with the claimed value.
- *Structural minimality.* For overshot cases, every entry in AdjPath must have child index 0 (leftmost descent); for undershot cases, the first adjustment entry must be the immediate right sibling at the fork point. These constraints ensure the result is minimal.

Theorem 6 (Lower Bound Soundness)**.** *If verification succeeds, then* $K_{result} = \text{lb}(Q)$ *with overwhelming probability.*

Proof sketch (Theorem 6). The verifier independently recomputes the fork depth f from the Merkle-committed extraction masks using Definition 7, preventing adversarial manipulation of the branching point. Structural constraints enforce minimality: in the overshot case, each entry in AdjPath must have child index 0 (leftmost descent); in the undershot case, the first adjustment entry must be the immediate right sibling of the search path's child at the fork point. The adjustment path must reconstruct to the same node content hash as the search path at depth f, cryptographically binding it to the committed trie structure. Any attempt to return a non-minimal key will produce a root hash mismatch. □

5.5 Range Proof

Range queries are essential for authenticated storage applications such as blockchain state synchronization, where clients must verify that a returned dataset contains *exactly* the entries within specified bounds [1, 26]. The central challenge is *completeness*: an adversarial prover might return a subset of the true range, omitting entries to deceive the verifier.

Definition 8 (Range Query)**.** *For an interval* $[first, last)$*, the range query returns all key-value pairs* (K, V) *satisfying* $first \leq K < last$.

MHOT constructs range proofs by composing lower bound proofs with multi-point proofs. The lower bound proofs (Section 5.4) establish the range boundaries, while the multi-point proof (Section 5.3) authenticates all entries within.

Definition 9 (HOT Range Proof)**.** *For interval* $[first, last)$*:*

$$\pi_{range} = (first, last, \pi^L_{lb}, \pi^R_{lb}, \pi_{multi}) \tag{11}$$

where π^L_{lb} *authenticates* $\text{lb}(first)$*,* π^R_{lb} *authenticates* $\text{lb}(last)$*, and* π_{multi} *proves membership of all entries in the range.*

Rank-based completeness verification. Efficient completeness verification builds on the *rank function*, which counts the number of keys preceding a given key in the trie's total order. Since each node records the leaf counts of its child subtrees and commits them in the node content hash, the verifier can derive ranks directly from the boundary proofs, without enumerating all keys in the queried range.

Definition 10 (Rank)**.** *For a key K in trie T, the rank is the count of smaller keys:*

$$\text{rank}(K) = |\{K' \in T : K' < K\}| \tag{12}$$

Lemma 7 (Rank Computation from Path)**.** *Given the search path for key K, the rank can be computed as:*

$$\text{rank}(K) = \sum_{i=0}^{h-1} \sum_{j=0}^{child_idx_i - 1} \text{lc}(path[i].children[j]) \tag{13}$$

Algorithm 4 Range Proof Verification

Require: proof π, expected root R
Ensure: TRUE if proof is valid

▷ *Verify boundary proofs first (prevents omission attacks)*
if not VERIFYLB$(\pi.\pi^L_{\text{lb}}, R)$ **then return** FALSE
end if
if not VERIFYLB$(\pi.\pi^R_{\text{lb}}, R)$ **then return** FALSE
end if
if π.first $\geq \pi$.last **then**
    return π.entries $= \emptyset$
end if
if not VERIFYMULTI$(\pi.\pi_{\text{multi}}, R)$ **then return** FALSE
end if
▷ *Rank-based count verification*
$r_L \leftarrow$ COMPUTERANK$(\pi.\text{first}, \pi.\pi^L_{\text{lb}})$
$r_R \leftarrow$ COMPUTERANK$(\pi.\text{last}, \pi.\pi^R_{\text{lb}})$
if $r_R - r_L \neq |\pi.\text{entries}|$ **then**
    return FALSE ▷ Omission detected
end if
▷ *Verify ordering and boundaries*
for all $i \in [1, |\pi.\text{entries}|)$ **do**
    if $\pi.\text{entries}[i].\text{key} \leq \pi.\text{entries}[i-1].\text{key}$ **then**
        return FALSE
    end if
end for
return TRUE

Proof sketch (Lemma 7). At each level i of the search path, all children with indices $j <$ child_idx$_i$ contain keys lexicographically smaller than K, since sparse partial keys maintain sorted order within each compound node. The lc field, committed in the node content hash, records the total number of leaves in each child's subtree. Summing these counts across all path levels yields the total number of keys preceding K. □

This rank computation enables $O(1)$ completeness verification: the verifier simply checks whether $|\text{entries}| = \text{rank}(\text{last}) - \text{rank}(\text{first})$. If an adversary omits even a single entry, the count mismatch triggers rejection. Algorithm 4 presents the complete verification procedure.

The verification algorithm enforces a critical security invariant: boundary proofs must be verified *before* accepting any entries. This ordering prevents empty-proof attacks where an adversary provides valid but incomplete entry lists. The rank-based count check then ensures exactly the correct number of entries appears.

Theorem 8 (Range Proof Soundness)**.** *If verification succeeds, the entries contain exactly all keys K satisfying first $\leq K <$ last.*

Proof sketch (Theorem 8). The proof composes three security guarantees. First, by Theorem 6 (Lower Bound Soundness), the boundary proofs π^L_{lb} and π^R_{lb} correctly identify lb(first) and lb(last). Second, by Lemma 7 (Rank Computation), the verifier accurately computes rank(first) and rank(last) from these authenticated paths; since rank(last) − rank(first) equals exactly $|\{K : \text{first} \leq K < \text{last}\}|$, any count mismatch with |entries| reveals an omission attack. Third, by Theorem 4 (Multi-Proof Soundness), every entry in π_{multi} authentically exists in the committed trie. These guarantees ensure the returned entries are the keys in [first, last). □

6 Evaluation

We evaluate MHOT along four dimensions (formed by questions). **Q1.** Does MHOT improve write throughput compared to existing authenticated data structures? **Q2.** Does MHOT's batched persistence strategy reduce write amplification (WA)? **Q3.** How does tree height vary across workloads, and what are the implications for proof size? **Q4.** Does MHOT's compound node design mitigate Nurgle attacks?

6.1 Experimental Setup

Implementation. We implement all systems in Rust with release optimizations. RocksDB [23] serves as the underlying key-value store with a 2 GB LRU cache. We report the median of five independent runs; ranges in tables indicate variation across scales or configurations.

Hardware. We run experiments on an AWS EC2 instance with an 8-vCPU Intel Xeon Scalable processor (Sapphire Rapids, 3.2 GHz), 64 GB RAM, and EBS-optimized storage providing baseline 12,000 IOPS with burst capacity up to 40,000 IOPS.

Baselines. We compare MHOT against three representative systems under the benchmark setup used by LVMT [40]. (i) MPT is Ethereum's current authenticated state structure based on the Merkle Patricia Trie [55]. (ii) LVMT is a layered versioned multipoint trie that leverages KZG polynomial commitments to achieve $O(1)$ root updates [40]. (iii) RainBlock adopts DSM-TREE, a distributed sharded Merkle tree design optimized for in-memory storage [48]. The RainBlock-style configuration keeps the upper six levels in memory and pages deeper nodes from RocksDB, matching LVMT's layered-storage baseline [40]. For LVMT, we use the recommended configuration of 16 bits per level, resulting in a fanout of $2^{16} = 65{,}536$ per layer. We do not evaluate its History Merkle Tree functionality, as this component is not available in the open-source implementation.

Workloads. We evaluate two types of workloads. (i) The *synthetic* workload first populates the tree with 100k–1M entries, followed by 100 epochs, each consisting of 100,000 random updates. (ii) The *real-world trace* workload replays Ethereum mainnet blocks 13,500,000–13,510,000, grouped into 200 epochs of 50 blocks each, in accordance with LVMT's recommended configuration.

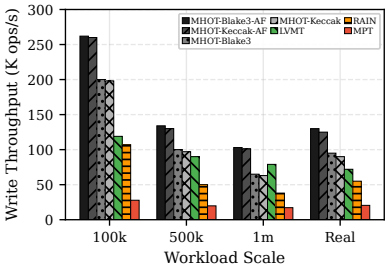


Figure 4: Write throughput. MHOT-AF denotes asynchronous flush; MHOT without suffix uses synchronous flush. MHOT outperforms MPT by 5–9× across all configurations.

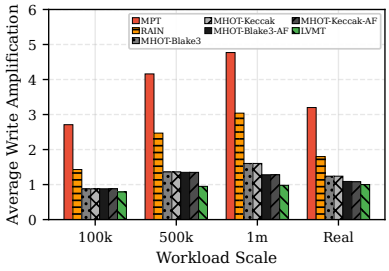


Figure 5: Average write amplification comparison. Lower is better. LVMT achieves the lowest WA via append-only storage. MHOT reduces MPT's WA by 3× via batched flushing.

6.2 Write Throughput (Figure 4)

Write throughput measures how fast the commitment engine processes batch updates (§3.1). We denote MHOT with asynchronous flush as MHOT-AF (asynchronous flush).

Synthetic workloads. At 100k keys, MHOT-AF reaches 260k ops/s, outperforming MPT (29k ops/s) by 9×. LVMT and RainBlock reach 120k and 108k ops/s respectively—roughly half of MHOT's throughput. Synchronous-flush MHOT variants hit 200k ops/s, still 7× faster than MPT.

As tree size grows, all systems show throughput degradation. At 500k keys, MHOT-AF maintains 135k ops/s while LVMT drops to 90k ops/s and RainBlock to 50k ops/s. At 1M keys, MHOT-AF delivers 104k ops/s versus MPT's 17k ops/s (6×), LVMT's 80k ops/s (1.3×), and RainBlock's 38k ops/s (2.7×). At larger scales, LVMT's $O(1)$ root update complexity narrows the gap, though MHOT retains an advantage through reduced tree traversal depth.

Real-world trace. Under Ethereum mainnet traces, MHOT-AF reaches 130k ops/s, outperforming LVMT (72k ops/s) by 1.8×, RainBlock (55k ops/s) by 2.4×, and MPT (20k ops/s) by 6.5×. The real-world trace exhibits higher key locality than synthetic workloads, benefiting MHOT's cache-friendly compound node layout. Blake3 variants outperform Keccak variants by 3–5% due to Blake3's lower computational overhead. For Ethereum compatibility, Keccak remains the default despite this modest penalty.

6.3 Write Amplification (Figure 5)

Synthetic workloads. At 100k keys, MHOT records average WA of 0.9, greatly lower than MPT's 2.7, a three times reduction. LVMT records the lowest WA (0.8) due to its LSM-tree-style append-only storage. RainBlock falls to 1.45.

As tree size grows, MPT's WA rises from 2.7 at 100k to 4.8 at 1M keys, reflecting the cost of maintaining deep Merkle paths with per-epoch commits. MHOT maintains WA of 0.9–1.6 across scales, a 3–3.7× reduction over MPT. LVMT consistently records the lowest WA (0.8–1.0).

Real-world trace. Under Ethereum mainnet traces, MHOT-AF records WA of 1.1, versus MPT's 3.2 (2.9× reduction), RainBlock's 1.8 (1.6× reduction), and LVMT's 1.0. Synchronous-flush MHOT variants show slightly higher WA (1.25) due to more frequent disk commits.

MHOT's batched flush strategy introduces per-epoch variance: most epochs complete with near-zero WA, while flush epochs reach 4–8 depending on accumulated changes. For aggregate storage efficiency (most relevant to long-running nodes), MHOT's lower WA reduces total I/O over time. LVMT records 10–15% lower WA than MHOT, but incurs 3–4 orders of magnitude higher verification latency (§6.5).

6.4 Tree Height Analysis (Figure 6)

Tree height directly impacts proof size and verification latency. Each additional level requires more node traversals and more sibling hashes in membership proofs.

Synthetic workloads. Under uniformly distributed keys, LVMT records the lowest tree height of 2 across all scales, reflecting its 2^{16} fanout per level. MHOT maintains height 5–6, while MPT and RainBlock reach 8–9. MHOT's compound node design yields 35–40% shallower trees than MPT.

Real-world trace. The Ethereum mainnet trace reveals a limitation of fixed-span architectures. LVMT maintains an average height of 2, but individual branches reach depth 9, approaching MPT and RainBlock's worst-case heights of 10. Real Ethereum addresses cluster within certain prefixes due to contract factories and sequential account creation, causing fixed 16-bit partitioning to produce unbalanced subtrees.

MHOT maintains stable height of 6 under real-world traces. Its variable-span compound nodes adapt to local key density by packing discriminative bits greedily, absorbing prefix collisions without proportional height increase. This consistency across workloads matters for blockchain deployments where key distributions vary unpredictably.

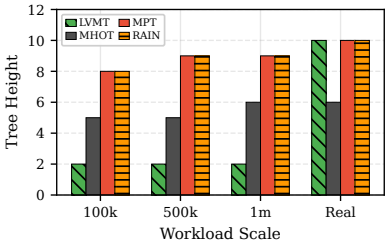


Figure 6: Tree height comparison. LVMT achieves optimal height under synthetic workloads but degrades to match MPT under real traces. MHOT maintains consistent height.

6.5 Proof Size and Latency

We evaluate single-point membership proofs across systems (Figure 7 and Table 2), then examine MHOT's scalability for multi-point and range proofs (Figure 8).

Single-point proofs. Figure 7 compares proof size and prove latency across five system configurations and four tree scales (100K–1M synthetic keys plus Ethereum mainnet trace with 1.6M keys). We evaluate two LVMT sampling strategies: *random* (best-case, sampling uniformly across keys) and *deepest* (worst-case, targeting keys at maximum tree depth).

MHOT-Blake3 achieves the most compact proofs across all scales, ranging from 1,139 bytes at 100K keys to 1,423 bytes under real-world traces. LVMT-random produces larger proofs (2,227–3,011 bytes) due to KZG commitment overhead, while LVMT-deepest reveals worst-case behavior with proofs reaching 4,411–19,123 bytes under real traces (13× larger than MHOT). MPT proofs range from 2,304 to 2,867 bytes, twice MHOT's size. MHOT's proof size advantage stems from its two-layer Merkle architecture: the intra-node Merkle tree requires only $O(\log k)$ sibling hashes per node rather than $O(k)$, where $k = 32$ is the maximum fanout.

For prove latency, LVMT-random achieves the lowest values (5.1–7.7 μs), while MHOT-Blake3 (8.9–11.9 μs) outperforms MHOT-Keccak (31.5–42.5 μs) by 3.5× due to Blake3's lower computational overhead. LVMT-deepest reaches 10.6–61.4 μs under real traces. MPT (6.2–8.9 μs) remains stable.

Table 2 presents verification latency. Hash-based schemes (MHOT, MPT) operate in microseconds, while LVMT's KZG polynomial commitment verification requires milliseconds due to pairing operations. MHOT-Blake3 verifies proofs in 5.3 μs, comparable to MPT (10–12 μs) while providing smaller proofs. LVMT's verification latency of 32–147 ms represents three to four orders of magnitude overhead compared to hash-based schemes, which is a critical trade-off for applications requiring fast verification such as light clients.

Multi-point and range proofs. Figure 8 shows proof size and latency scaling with batch and range size under Ethereum mainnet traces (1.6M keys). Multi-point proofs grow from 1.4 KB (single key) to 92.6 KB (1000 keys), while range proofs grow from 1.4 KB to 95.7 KB. For latency, multi-point

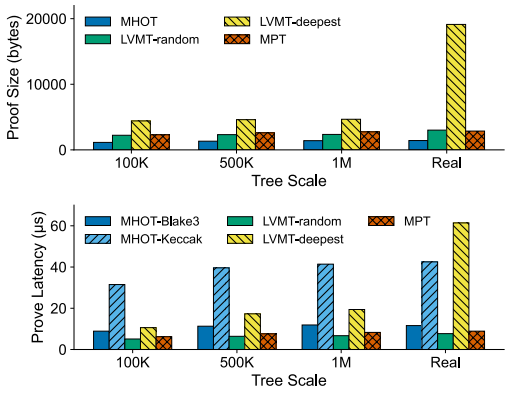


Figure 7: Single-point proof comparison across tree scales. Top panel shows proof size; bottom panel shows prove latency. LVMT-random represents best-case sampling; LVMT-deepest represents worst-case targeting of deep keys.

Table 2: Single-point verification latency. Hash-based schemes operate in microseconds; LVMT's KZG pairing requires milliseconds.

System	100K	500K	1M	Real
MHOT-Blake3	4.4 μs	5.1 μs	5.4 μs	5.3 μs
MHOT-Keccak	9.9 μs	11.2 μs	12.0 μs	11.9 μs
MPT	10.2 μs	11.5 μs	12.3 μs	12.3 μs
LVMT	33–47 ms	32–49 ms	32–51 ms	38–147 ms

prove time scales from 12 μs to 2.3 ms, while verify time scales from 5.5 μs to 4.9 ms. Range proofs exhibit similar scaling, with prove latency reaching 2.8 ms and verify latency reaching 5.1 ms at 1000 entries. The rank-based completeness verification (§5.5) adds minimal overhead, as rank computation requires only summation over pre-computed leaf counts.

6.6 Nurgle Attack Resistance

The Nurgle attack [31] exploits prefix collisions to inflate tree depth for targeted keys, increasing their proof costs. We evaluate resistance under the threat model where an adversary controls 52 prefix bits (matching the original Nurgle analysis) and commands an entire block's gas budget (300M gas, approximately 15,000 insertions).

Experimental setup. We sample 10,000 random keys as attack targets. For each target, the attacker generates collision keys matching the target's 52-bit prefix and inserts them until either the target's depth increases or the block's gas budget is exhausted. For MPT, we employ a round-robin strategy distributing insertions across all targets to maximize attack coverage. For LVMT, we track the depth distribution of both original and attack keys, since its fixed-depth-at-insertion property prevents existing keys from being pushed deeper.

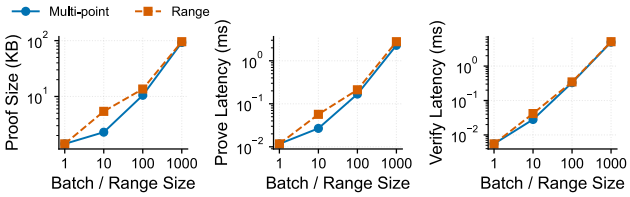


Figure 8: Multi-point and range proof scalability under Ethereum mainnet traces (Blake3). Proof size and latency scale linearly with batch size. Verify latency dominates at larger batch sizes due to hash recomputation.

Table 3: Nurgle attack on Ethereum mainnet.

System	Success Rate	Depth Δ (avg)	Depth Δ (max)
MPT	99.97%	+2.03	+3
MHOT	**0%**	**0**	**0**

Attack results (Table 3). MPT proves highly vulnerable: 99.97% of sampled keys experienced depth increases, with average depth rising from 6.88 to 8.91 (+2.03 levels) and maximum depth increasing from 9 to 12. In contrast, MHOT exhibits strong resistance: zero successful attacks across all 10,000 targets, even after exhausting an entire block's gas budget per target. MHOT's compound nodes absorb prefix collisions through internal restructuring (Leaf Pushdown, Parent Pull-Up) without propagating depth increases.

LVMT prefix pollution. LVMT presents a different security model: once inserted, a key's depth is fixed and cannot be increased by subsequent insertions. However, attackers can still pollute prefix regions by inserting keys that occupy slots at shallower levels, forcing future keys into deeper levels.

Figure 9 illustrates this effect. Before the attack, 82.3% of keys reside at level 0 and 17.4% at level 1 (average depth 0.18). After inserting 15,000 attack keys targeting specific prefixes, 99.9% of attack keys are placed at level 3, far deeper than legitimate keys. Although existing keys remain unaffected, the polluted prefix regions force any future keys sharing these prefixes to be placed at level 3 or deeper. This behavior constitutes a degradation-of-service attack against future users whose addresses collide with the targeted prefixes.

Key experimental findings. Our experiments yield four key findings. MHOT achieves 5–9× higher write throughput than MPT, reaching up to 260k ops/s at 100k keys and 100k ops/s at 1M keys (**Q1**). This gain is partly due to reduced write amplification: MHOT achieves average WA of 0.9–1.6, a 3–4× reduction compared to MPT (**Q2**). MHOT also maintains a stable tree height of 5–6 levels across all workloads, 35–40% shallower than MPT, resulting in 50% smaller membership proofs (1.1–1.4 KB vs. 2.3–2.9 KB) with comparable verification latency (**Q3**). Finally, under the Nurgle threat model with a 15,000-insertion budget, none of the 10,000 sampled keys experienced depth increases (**Q4**).

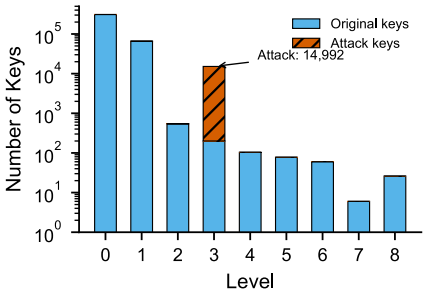


Figure 9: LVMT key distribution by level before and after Nurgle attack. Attack keys (hatched) concentrate at level 3, polluting targeted prefix regions for future insertions.

7 Conclusion

We presented MHOT, a height-optimal authenticated data structure for blockchain state commitment. By adapting height-optimized tries to persistent storage, MHOT achieves substantially higher throughput, lower write amplification, and smaller proofs than MPT, without relying on trusted setup or specialized cryptography. MHOT structurally mitigates Nurgle attacks, maintaining zero successful depth increases even under worst-case adversarial conditions. Our results show that careful data-structure design can fundamentally improve the scalability and robustness of blockchain state commitment.

Acknowledgement

This work is supported in part by the National Natural Science Foundation of China under grant 62572038, 62303037, 62272464 and U24B20144, the Key R&D Program of Zhejiang Province (No. 2025C01084), the Zhejiang Provincial Natural Science Foundation of China under Grant No. LZ26F030002, Beijing-Tianjin-Hebei natural science foundation cooperation project 25JJJC0033, the Fundamental Research Funds for the Central Universities; and the Research Funds of Renmin University of China(No. 202530187). Qin Wang is free of any funding supports.

Ethical Considerations

Stakeholder Identification and Impacts. Direct stakeholders are the research team, blockchain client maintainers (Geth, Reth, Erigon, Nethermind, Besu, etc.), and platform designers for new chains, appchains, L2s, and rollups. Indirect stakeholders include validators and full-node operators, end users and token holders, protocol governance participants (EIP authors, L2 councils), the broader research community, and adversaries targeting blockchain state commitment.

Research Team. No human subjects were involved, and we made no attempt to collect off-chain identity information, deanonymize addresses, or link on-chain addresses to

real-world identities; addresses and storage keys in traces are treated only as pseudonymous protocol-level identifiers. All experiments are mainnet-trace replay rather than live interaction. We used public RPC interfaces only to retrieve historical state-change records on Ethereum mainnet blocks 13,500,000 to 13,510,000 under normal rate limits, replayed in isolated local environments. We did not broadcast transactions, alter blockchain state, deploy contracts, or consume on-chain or public consensus resources.

Clients and Platform Designers. The open-source reference implementation lets these stakeholders evaluate MHOT against their deployment constraints; for new chains, appchains, L2s, and rollups it offers a concrete option with quantified trade-offs, while adoption by existing clients remains a decision through their own governance processes.

Validators, End Users, and Adversaries. Validators and full-node operators may benefit from reduced state-pollution exposure but bear migration and compatibility costs if their ecosystems adopt a new commitment structure; end users and token holders benefit indirectly from improved node sustainability but could be harmed by poorly coordinated migrations. Adversaries may glean cost-structure information about state-commitment defenses, though the underlying Nurgle attack is already public and our artifact provides no exploit automation.

Menlo Principles. *Beneficence.* MHOT is intentionally defensive, raising the structural cost of Nurgle-style state-pollution attacks for validators, full-node operators, and end users. The Nurgle attack was publicly disclosed by He et al. at IEEE S&P 2024, and we have not identified new attack vectors introduced by this paper. *Respect for Persons.* All on-chain data is publicly replayable; we made no attempt to deanonymize transactions, link addresses to identities, or interfere with any user's on-chain activity. *Justice.* No on-chain resources, consensus resources, or transaction fees were consumed; benefits distribute through open-source release while adoption and migration costs accrue only to parties who elect to adopt. *Respect for Law and Public Interest.* We used only publicly available blockchain data and open-source baselines, exploited no vulnerabilities, and violated no terms of service.

Potential Harms and Mitigations. An L1 hard fork carries consensus-divergence risk if mis-executed; we mitigate by publishing soundness theorems, a full reference implementation, and low-friction deployment paths for new chains, appchains, L2s, and rollups. Protocol-layer change burdens smaller operators and newer chains more than incumbents; open-source release and new-chain framing partially mitigate this, though the asymmetry cannot be fully neutralized for mainnet retrofits. MHOT's data-dependent bit extraction is less SNARK-friendly than Verkle-style designs, which may bias ZK-native platforms.

Decision to Conduct and Publish. *Conduct.* The Nurgle attack is publicly disclosed at IEEE S&P 2024, our experiments carry no on-chain footprint, and expected benefits to validators, full-node operators, and end users outweigh the low residual risk. *Publish.* Public dissemination enables independent audit, a stronger mitigation against mis-deployment than non-transparent withholding; the open-source artifact and soundness theorems let the community and protocol governance evaluate adoption on their own terms. We identified no ethical concerns sufficient to warrant withholding, though residual dual-use risk from unaudited deployments remains.

Open Science

To promote transparency and reproducibility, we make the MHOT artifact package publicly available[1]. The archive contains three software components: the main benchmark suite, the EVM I/O trace generator, and the mainnet replay-data exporter. Together, they provide the source code, baseline implementations, benchmark drivers, data-generation tools, and documentation needed to build the artifact and reproduce our benchmark workflows. Researchers can use these artifacts to understand the design, adapt the implementation to their own experiments, obtain the same dataset, and reproduce our evaluation results.

Usage of GenAI Tools

During the preparation of this manuscript, we only used generative AI tools to assist with the grammar checking and sentence polishing. All AI-assisted content was carefully reviewed, verified, and substantially revised by the authors to ensure technical accuracy and academic rigor. The experimental design, data analysis, result interpretation, and core technical contributions were conducted independently by the authors without AI assistance. We take full responsibility for all content in this paper.

References

[1] devp2p/caps/snap.md at master · ethereum/devp2p. URL: https://github.com/ethereum/devp2p/blob/master/caps/snap.md.

[2] Shashank Agrawal and Srinivasan Raghuraman. Kvac: Key-value commitments for blockchains and beyond. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, volume 12493 of *LNCS*, pages 839–869. Springer, 2020. doi:10.1007/978-3-030-64840-4_28.

[3] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *Network and Distributed System Security Symposium (NDSS)*, 2018. doi:10.14722/ndss.2018.23241.

[1] https://doi.org/10.5281/zenodo.20338741

[4] Guillaume Ballet and Vitalik Buterin. EIP-3102: Binary trie structure. Ethereum Improvement Proposals, September 2020. https://eips.ethereum.org/EIPS/eip-3102.

[5] Guillaume Ballet, Vitalik Buterin, Dankrad Feist, Ignacio Hagopian, Tanishq Jasoria, and Gajinder Singh. EIP-4762: Statelessness gas cost changes. Ethereum Improvement Proposals, February 2022. https://eips.ethereum.org/EIPS/eip-4762.

[6] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. Height optimized tries. *ACM Transactions on Database Systems (TODS)*, 47:1 – 46, 2022.

[7] Remco Bloemen. Merkle multi-proofs. Technical Note, 2025. Available at: https://xn--2-umb.com/25/merkle-multi-proof/.

[8] Vitalik Buterin. EIP-150: Gas cost changes for io-heavy operations. Ethereum Improvement Proposals, October 2016. https://eips.ethereum.org/EIPS/eip-150.

[9] Vitalik Buterin, Guillaume Ballet, Dankrad Feist, Ignacio Hagopian, Kevaundray Wedderburn, Tanishq Jasoria, Gajinder Singh, Danno Ferrin, Piper Merriam, and Gottfried Herold. EIP-7864: Ethereum state using a unified binary tree. Ethereum Improvement Proposals, January 2025. https://eips-wg.github.io/EIPs/7864/.

[10] Vitalik Buterin, Guillaume Ballet, Dankrad Feist, Ignacio Hagopian, Kevaundray Wedderburn, Tanishq Jasoria, Gajinder Singh, Danno Ferrin, Piper Merriam, and Gottfried Herold. EIP-7864: Ethereum state using a unified binary tree. Ethereum Improvement Proposals, January 2025. https://eips.ethereum.org/EIPS/eip-7864.

[11] Vitalik Buterin, Dankrad Feist, Kevaundray Wedderburn, Guillaume Ballet, Piper Merriam, Gottfried Herold, Ignacio Hagopian, Tanishq Jasoria, Gajinder Singh, and Danno Ferrin. EIP-6800: Ethereum state using a unified verkle tree. Ethereum Improvement Proposals, March 2023. https://eips.ethereum.org/EIPS/eip-6800.

[12] Vitalik Buterin and Martin Swende. EIP-2929: Gas cost increases for state access opcodes. Ethereum Improvement Proposals, September 2020. https://eips.ethereum.org/EIPS/eip-2929.

[13] Miao Cai, Xuzhen Jiang, Junru Shen, and Baoliu Ye. Splitdb: Closing the performance gap for lsm-tree-based key-value stores. *IEEE Transactions on Computers (TC)*, 73:206–220, 2024.

[14] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, volume 12492 of *LNCS*, pages 3–35. Springer, 2020. doi:10.1007/978-3-030-64834-3_1.

[15] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *International Conference on Practice and Theory in Public-Key Cryptography (PKC)*, volume 7778 of *LNCS*, pages 55–72. Springer, 2013. doi:10.1007/978-3-642-36362-7_5.

[16] Dario Catalano, Dario Fiore, Rosario Gennaro, and Emanuele Giunta. On the impossibility of algebraic vector commitments in pairing-free groups. In *Theory of Cryptography Conference (TCC)*, volume 13748 of *LNCS*, pages 274–299. Springer, 2022. doi:10.1007/978-3-031-22365-5_10.

[17] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 570–587, 2021.

[18] Zehao Chen, Bingzhe Li, Xiaojun Cai, Zhiping Jia, Lei Ju, Zili Shao, and Zhaoyan Shen. Chainkv: A semantics-aware key-value store for ethereum system. *Proceedings of the ACM on Management of Data (SIGMOD)*, 2023.

[19] Alexander Chepurnoy, Charalampos Papamanthou, Shravan Srinivasan, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968, 2018. https://eprint.iacr.org/2018/968.

[20] Jemin Andrew Choi, Sidi Mohamed Beillahi, Peilun Li, Andreas G. Veneris, and Fan Long. Lmpts: Eliminating storage bottlenecks for processing blockchain transactions. In *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2022.

[21] Leo de Castro and Chris Peikert. Functional commitments for all functions, with transparent setup. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2023.

[22] Yangshen Deng, Muxi Yan, and Bo Tang. Accelerating merkle patricia trie with gpu. *Proceedings of the VLDB Endowment (VLDB)*, 17:1856–1869, 2024.

[23] Siying Dong, Andrew Kryder, Yanqin Jin, Lin Peng, Kanchan Mehra, Jeremy Yakdus, Wei-Nee Chen, Abhishek Sharma, Youngjin Kwon, and Gary J. Katz.

RocksDB: Evolution of development, optimization and uses of lsm-based storage. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[24] Erigon Team. Erigon: Ethereum implementation on the efficiency frontier. GitHub Repository, 2024. Available at: https://github.com/erigontech/erigon.

[25] Libo Feng and Xian Deng. An efficient hot/cold data separation scheme for storage optimization in consortium blockchain full nodes. *Cluster Computing*, 28, 2025.

[26] Enrique Fynn, Ethan Buchman, Zarko Milosevic, Robert Soulé, and Fernando Pedone. Robust and fast blockchain state synchronization. In Eshcar Hillel, Roberto Palmieri, and Etienne Rivière, editors, *26th International Conference on Principles of Distributed Systems, OPODIS 2022, Brussels, Belgium, December 13-15, 2022*, volume 253 of *LIPIcs*, pages 8:1–8:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.OPODIS.2022.8, doi:10.4230/LIPICS.OPODIS.2022.8.

[27] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. In *International Conference on Software Engineering (ICSE)*, pages 2315–2326, 2022.

[28] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 232–244, 2023.

[29] Geth Team. Parallel intermediate node fetching (for a single trie). Go Ethereum Issue #28266, 2023. Available at: https://github.com/ethereum/go-ethereum/issues/28266. Accessed: 2026-05-18.

[30] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2007–2023. ACM, 2020. doi:10.1145/3372297.3417244.

[31] Zheyuan He, Zihao Li, Ao Qiao, Xiapu Luo, Xiaosong Zhang, Ting Chen, Shuwei Song, Dijun Liu, and Weina Niu. Nurgle: Exacerbating resource consumption in blockchain state storage via mpt manipulation. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2180–2197, 2024.

[32] Xiaowen Hu, David Zhao, and Bernhard Scholz. Synthesizing efficient super-instruction sets for ethereum virtual machine. In *ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2024.

[33] Simon Jentzsch and Christoph Jentzsch. EIP-1186: Rpc-method to get merkle proofs - eth_getproof. Ethereum Improvement Proposals, June 2018. https://eips.ethereum.org/EIPS/eip-1186.

[34] George Kadianakis, lightclient, and Alex Stokes. EIP-4444: Bound historical data in execution clients. Ethereum Improvement Proposals, November 2021. https://eips.ethereum.org/EIPS/eip-4444.

[35] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2010.

[36] András Kovács and Tamás Kis. Partitioning of trees for minimizing height and cardinality. *Information Processing Letters*, 89:181–185, 2004.

[37] John Kuszmaul. Verkle trees. 2019. Available at: https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf.

[38] Oleksandr Kuznetsov, Dzianis Kanonik, Alex Rusnak, Anton Yezhov, and Oleksandr Domin. Adaptive restructuring of merkle and verkle trees for enhanced blockchain scalability. *Internet of Things*, 27:101315, 2024.

[39] Derek Leung, Yossi Gilad, Sergey Gorbunov, Leonid Reyzin, and Nickolai Zeldovich. Aardvark: An asynchronous authenticated dictionary with applications to account-based cryptocurrencies. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4237–4254. USENIX Association, 2022.

[40] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. Lvmt: An efficient authenticated storage for blockchain. *ACM Transactions on Storage*, 20(3), June 2024. doi:10.1145/3664818.

[41] Haoran Lin, Hang Feng, Yajin Zhou, and Lei Wu. Parallelevm: Operation-level concurrent transaction execution for evm-compatible blockchains. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2025.

[42] Zhongtang Luo, Yanxue Jia, Alejandra Victoria Ospina Gracia, and Aniket Kate. Cauchyproofs: Batch-updatable vector commitment with easy aggregation

and application to stateless blockchains. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2025. doi:10.1109/SP61157.2025.00247.

[43] Charles U. Martel, Glen Nuckolls, Premkumar T. Devanbu, Michael Gertz, April Kwong, and Stuart G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39:21–41, 2004.

[44] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Annual International Cryptology Conference*, 1987. URL: https://api.semanticscholar.org/CorpusID:28484604.

[45] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[46] Rui Pan, Chubo Liu, Guoqing Xiao, Mingxing Duan, Keqin Li, and Kenli Li. An algorithm and architecture co-design for accelerating smart contracts in blockchain. In *Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[47] Paradigm. Reth: Modular, contributor-friendly and blazing-fast implementation of the ethereum protocol in rust. GitHub Repository, 2024. Available at: https://github.com/paradigmxyz/reth.

[48] Soujanya Ponnapalli, Aashaka Shah, Amy Tai, Souvik Banerjee, Vijay Chidambaram, Dahlia Malkhi, and Michael Yung Chung Wei. Rainblock: Faster transaction processing in public blockchains. In *USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[49] Xiaodong Qi, Zhihao Chen, Haizhen Zhuo, Quanqing Xu, Chengyu Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, Ying Yan, and Hui Zhang. Schain: Scalable concurrency over flexible permissioned blockchain. *IEEE International Conference on Data Engineering (ICDE)*, pages 1901–1913, 2023.

[50] Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments. In *USENIX Security Symposium*, pages 3001–3018. USENIX, 2022.

[51] Shikun Tian, Zhonghao Lu, Haizhen Zhuo, Xiaojing Tang, Peiyi Hong, Shenglong Chen, Dayi Yang, Ying Yan, Zhiyong Jiang, Hui Zhang, and Guofei Jiang. Letus: A log-structured efficient trusted universal blockchain storage. In *Companion of the 2024 International Conference on Management of Data (SIGMOD)*, 2024.

[52] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *Security and Cryptography for Networks (SCN)*, volume 12238 of *LNCS*, pages 45–64. Springer, 2020. doi:10.1007/978-3-030-57990-6_3.

[53] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan-Gueta, and Srinivas Devadas. Towards scalable threshold cryptosystems. In *IEEE Symposium on Security and Privacy (S&P)*, pages 877–893, 2020.

[54] Qian Wei, Zehao Chen, Xiaowei Chen, Yuhao Zhang, Xiaojun Cai, Zhiping Jia, Zhaoyan Shen, Yi Wang, Zili Shao, and Bingzhe Li. A semantic-integrated lsm-tree-based key-value storage engine for blockchain systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2024.

[55] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Yellow Paper, 2014. Available at: https://ethereum.github.io/yellowpaper/paper.pdf.

[56] Gavin Wood. EIP-161: State trie clearing (invariant-preserving alternative). Ethereum Improvement Proposals, October 2016. https://eips.ethereum.org/EIPS/eip-161.

[57] Cuihua Yang, Fan Yang, Quanqing Xu, Yongquan Zhang, and Junqing Liang. Solsdb: Solve the ethereum's bottleneck caused by storage engine. *Future Generation Computer Systems (FGCS)*, 160:295–304, 2024.

[58] Ce Zhang, Cheng Xu, Haibo Hu, and Jianliang Xu. Cole: A column-based learned storage for blockchain systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2023.

[59] Shijie Zhang, Ru Cheng, Xinpeng Liu, Jiang Xiao, Hai Jin, and Bo Li. Seer: Accelerating blockchain transaction execution by fine-grained branch prediction. *Proceedings of the VLDB Endowment (VLDB)*, 18(3):822–835, 2024.

[60] Wei Zhou, Changzheng Wei, Ying Yan, Wei Tang, et al. DTVM: Revolutionizing smart contract execution with determinism and compatibility. *arXiv preprint arXiv:2504.16552*, 2025. Available at: https://arxiv.org/abs/2504.16552.

A Extended Discussion

A.1 Scope and Complementary Mitigations

Our scope. Our evaluation targets the state commitment bottleneck in modern clients. Integrating MHOT into full execution pipelines would validate its end-to-end impact on block processing. MHOT operates at the ADS layer beneath Ethereum's account model [55], treating account fields as

opaque key-value pairs. It therefore remains compatible with state-clearing semantics [56], proof interfaces [33], and gas accounting standards [12]. Deployment can follow an incremental migration path where new commits use MHOT while historical proofs remain verifiable against archived MPT roots.

Reduced commitment latency may also benefit block validation throughput, though quantifying this effect requires end-to-end pipeline evaluation.

Complementary mitigation strategies. In-memory layering caches reused upper trie levels to reduce authenticated-state cost without changing the local tree. LMPTs [20] keep recent-update tries in memory and the snapshot trie on disk; our RainBlock-style baseline keeps the upper six levels in memory and loads deeper nodes from RocksDB. This helps the normal case where updates reuse upper ancestors, but it does not close the Nurgle path. Adversarial keys share long prefixes and grow paths below the cached levels (a single RTX3080 GPU can manipulate the first 15 MPT layers by colliding 13 nibbles [31]), and the tree still routes by fixed prefixes regardless of how deep the cache extends.

Systemic mitigations reorganize state above the local tree. Chainspace [3] splits state across shards and certifies shard-local commitments through quorum signatures, while RainBlock [48] distributes state in DSM-TREE shards and offloads disk access to storage nodes. These designs expose parallelism above the tree at the cost of data placement, network coordination, quorum certification, and cross-shard access, and compose with MHOT, since each shard or checkpoint still requires a local authenticated structure that remains exposed to depth inflation when prefix-based. MHOT fills that local structural role inside layered, partitioned, or checkpointed deployments.

A.2 Future Directions

Hardware acceleration. Increasing the span to $k = 64$ [6] exploits AVX-512 instructions on modern processors, further reducing tree height. For spans of $k \geq 256$, scalar implementations suffice because persistent storage I/O dominates latency. Alternatively, algebraic commitments such as vector commitments in AMT [40] trade setup transparency for reduced verification overhead. MHOT's deferred hashing exposes parallelism well-suited to GPU acceleration, where nodes at the same height level are mutually independent and map efficiently onto GPU SIMD units [22].

Storage optimization. Integrating HOT's leaf-optimized node layout would substantially reduce node sizes through dense-region collapsing and variable-length delta encoding [6], though deterministic Merkle hash computation must remain tractable.

State pruning. Content-addressable, copy-on-write storage creates new node versions on every modification. Version-prefixed database keys allow efficient range deletion of nodes older than a retention threshold, and the underlying LSM-tree engine's compaction naturally discards tombstoned entries. A production deployment would benefit from a configurable pruning policy that balances historical state availability against storage growth.

Table 4: Summary of notation.

Symbol	Description
s, n, k, h	Span, entry count, fanout (k=32), tree height
K, V, T	Key, value, trie
$H, \pi, \|$	Hash function, proof, concatenation
N	Compound node
M, S, L	Extraction masks, sparse keys, leaf counts
η, v, c_i	Node height, version, i-th child
$\mathrm{CMR}(N)$	Children Merkle root
J, t	Child index set, $t = \|J\|$
$\mathrm{NPE}(N, J)$	Node proof entry for children J
Π_J^{CMR}	Intra-node Merkle multiproof
$m, \mathrm{rank}(K)$	Batch size, count of keys $< K$
$\mathrm{lc}(c), \mathrm{lb}(Q)$	Leaf count of subtree c, lower bound of Q
$\lambda, \mathrm{negl}(\lambda)$	Security parameter, negligible function

Concurrency. HOT's copy-on-write semantics with wait-free readers [6] provide a foundation for concurrent access. MHOT's immutable nodes already support concurrent reads without synchronization. However, coordinating parallel writers remains an open problem.

Structural attack analysis. MHOT removes Nurgle's prefix-collision attack through discriminative-bit indexing, but its structure may open new attack vectors. An adversary could craft keys that concentrate discriminative-bit conflicts within certain subtrees, forcing repeated node splits such as leaf pushdown, parent pull-up, and intermediate node creation. Following Nurgle's analytical approach, a systematic study would estimate the effort to force such splits, measure the resulting path-length increase per malicious insertion, and evaluate whether these effects produce economic imbalance under the gas pricing model [12]. We leave tight bounds on adversarial path inflation to future work.

B Notations (Table 4)

C Deferred Related Work

Protocol-level updates. Ethereum's roadmap explores replacing MPT with alternative state commitment structures. Verkle Trees [37], proposed in EIP-6800 [11], adopt vector commitments to achieve constant-size proofs per tree level, enabling stateless client verification. This design relies on a trusted setup; while multi-party computation ceremonies distribute

trust, compromise remains possible. As an alternative, EIP-7864 [10] proposes a binary Merkle tree tailored for SNARK-based proof generation, partly motivated by post-quantum concerns of pairing-based schemes [21]. Both approaches require hard forks and incur nontrivial deployment costs, either through additional trust assumptions or substantial proof generation overhead.

In contrast, MHOT demonstrates that performance and robustness gains remain achievable within the existing hash-based commitment model. More importantly, MHOT preserves the simplicity of Merkle authentication without introducing algebraic verification overhead for deployment.

Vector commitments. VC literature addresses a different layer, asking how to commit to a value set and open positions with short proofs rather than how state is organized, indexed, or traversed; the two layers compose. When paired with a trie, as in Verkle and LVMT (§2.2), a VC decouples proof size from fanout cryptographically, while MHOT achieves the same decoupling structurally via discriminative-bit indexing and hierarchical proofs, a different design point, not a competing one. Pure VC constructions without a tree, such as Aardvark [39] (bucketed dictionary), KVaC [2] (flat key-value commitment), and EDRAX [19] (indexed authenticated array), target different objectives and do not map onto the axes of Table 1. Proof-aggregation techniques (Pointproofs [30], Hyperproofs [50], Cauchyproofs [42], aSVC [52]) are orthogonal to the base structure; foundational results [14–16] establish the VC primitive, incremental aggregation, and impossibility bounds. The lack of native range proofs in LVMT and Verkle (Table 1) reflects engineering status, not a fundamental VC limitation; interval certification needs extra construction that hash-based Merkle proofs avoid.

State storage optimization. Beyond modifications to the authenticated tree structure itself, prior work has explored workload-aware optimizations for blockchain state storage. Adaptive tree restructuring [38] dynamically promotes frequently accessed nodes toward the root, reducing average access latency for hot state. Hot–cold data separation schemes [25] migrate infrequently accessed state to lower-cost storage tiers while preserving fast access to active accounts. These approaches primarily optimize for access frequency and locality, rather than the structural properties of the key space.

In contrast, MHOT's height optimization targets worst-case structural depth induced by key relationships, independent of workload skew. The two strategies are orthogonal and can be naturally composed.

Storage architecture. Blockchain-aware storage engines mitigate I/O amplification by exploiting blockchain-specific access patterns, primarily through two complementary strategies. The first leverages key structure and layout. ChainKV [18] separates state from non-state data using Prefix-MPT to improve key locality, while Block-LSM [54] prefixes keys with block numbers to cluster same-block writes and reduce compaction overhead. The second strategy amortizes commitment cost over time. LETUS [51] employs log-structured delta encoding across blocks, and COLE [58] applies learned indexes to optimize read-heavy workloads.

Modern production clients such as Erigon [24] further adopt flat database designs that decouple state access from commitment computation, effectively treating the authenticated data structure as a dedicated commitment engine. MHOT follows this decoupling principle and focuses optimization squarely on the commitment bottleneck itself.

Acceleration techniques. Hardware acceleration complements algorithmic improvements. Deng et al. [22] parallelize MPT hash computation on GPUs via PhaseNU and LockNU algorithms, addressing node-splitting conflicts during concurrent updates. This achieves substantial throughput gains for commitment-intensive workloads. Combining GPU parallelism with height-optimized structures remains unexplored.

MHOT's deferred hashing exposes parallelism suited for such acceleration: nodes at the same height level share no data dependencies during hash computation.

Nurgle mitigation. The attack [31] exploits the predictable structure of MPT to inflate tree depth via adversarial key selection. Current mitigation efforts focus on economic disincentives or data pruning rather than structural defenses. EIP-4444 [34] enables historical data pruning but leaves the current state structure unchanged. EIP-4762 [5] proposes gas repricing for witness costs but remains in draft, facing determinism challenges. Verkle migration [11, 37] would alter the attack surface but requires ongoing protocol changes.

MHOT provides an immediate structural defense. Even when an adversary controls an entire block's gas budget, MHOT achieves zero successful depth increases, compared to 99.97% attack success rate against MPT. The compound node design absorbs prefix collisions through internal restructuring without propagating depth increases to existing keys.

Algorithm 5 Compact Multiproof Generation [7]

Require: leaves L, indices I to prove
Ensure: Compact multiproof π
depth $\leftarrow \lceil \log_2 |L| \rceil$
known $\leftarrow \{(\text{depth}, i) : i \in I\}$
proof_hashes $\leftarrow \langle\rangle$
for $\ell \leftarrow$ depth -1 **downto** 0 **do**
  for $i \leftarrow 0$ **to** $2^\ell - 1$ **do**
    left $\leftarrow (\ell+1, 2i) \in$ known
    right $\leftarrow (\ell+1, 2i+1) \in$ known
    if left **and** right **then**
      known.INSERT$((\ell, i))$
    else if left **or** right **then**
      $s \leftarrow 2i+1$ **if** left **else** $2i$
      proof_hashes.PUSH(HASH$(\ell+1, s)$)
      known.INSERT$((\ell, i))$
    end if
  end for
end for
return $(I, \text{proof_hashes}, \text{depth})$