Authenticated Data Structures for Dynamic Workloads

Ziheng (Tom) Shangguan[1], Aviv Yaish[2,3,4], Dahlia Malkhi[1]

[1]UCSB, [2]Yale University, [3]IC3, [4]Complexity Science Hub Vienna

Abstract

We introduce the *Huffman-Merkle Tree (HMT)*, an authenticated data structure (ADS) optimized for dynamic workloads where some items may be more frequently accessed than others, and access frequencies change over time. An ADS allows proving item membership against a short commitment to a large mutable state, with applications including verifiable storage, Internet transparency services, and blockchains. Optimizing ADS performance under continuously changing access frequencies has not been fully addressed before, neither in theory nor in practice.

HMT tackles access skew via tiering: "hot" items are stored according to a Huffman coding layout where frequently accessed items are closer to the root and thus contribute less to overall costs, while a binary Merkle Tree (MT) is used for cold and new elements to lower their update overhead. To efficiently handle dynamic workloads, we incrementally apply and batch layout changes, track access frequencies via count-min sketch, use a tier promotion cache, and consider various tier migration policies.

We implement HMT and compare it on real-world data against Ethereum's Merkle Patricia Trie (MPT) ADS and its proposed replacement, the Unified Binary Tree (UBT). Our evaluation considers two metrics: the amount of hashing per ADS update and access-weighted membership-proof size. The latter metric captures both the cost of accessing each item and its access frequency. We find that the best HMT policy uses about 2.4× and 0.34× less average hash operations than MPT and UBT respectively, while featuring proofs shorter by 0.18× than MPT and 0.55× than UBT.

Keywords: Merkle trees, state commitments, blockchains.

1 Introduction

Authenticated Data Structures (ADSs) allow committing to a large state using a short digest and proving against the digest that specific elements are included. They are useful for memory checking [40], remote storage [33, 27], internet certificates [7], and blockchains [25].

The Merkle Tree (MT) is a widely used ADS where elements appear as leaves, each internal node stores a hash of the concatenation of its children, and the root hash serves as a compact commitment to the entire structure. Item inclusion is proven by showing the sibling hashes along the path from the element to the root. So, inclusion proofs in a balanced binary MT with N leaves contain $\log N$ hash values, incurring logarithmic cost regardless of element access frequencies. This shortcoming extends to other notable ADSs, e.g., Merkle-Patricia Tries (MPTs) [41], which also suffer costs that are independent of access frequencies.

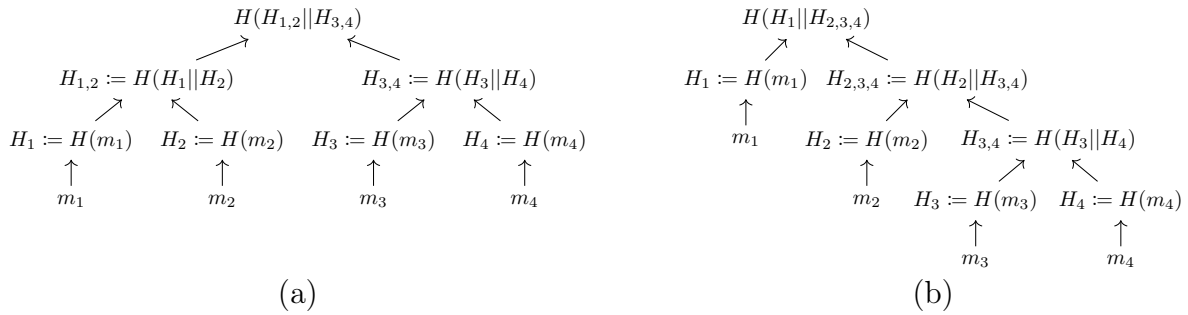


Figure 1: Panel (a) depicts a Merkle Tree (MT) and (b) shows a Huffman Merkle Hash Tree (HuffMHT), both with 4 items with monotonically decreasing access frequencies, i.e., m_i is accessed more frequently than m_{i+1}.

Real-world workloads are typically skewed, with some elements accessed more frequently than others [17]. Our focus is therefore the *average-case* performance of ADSs under such workloads, where each element's proof and update cost is weighted by its access frequency.

When all access frequencies are known in advance, this optimization problem is well understood: the optimal static tree can be described by a Huffman code [29]. However, the dynamic setting is more challenging. Prior work has addressed changing access distributions by rebuilding the entire data structure whenever the distribution changes [28]. However, this incurs overhead which must be accounted for in the same concrete units as proof generation and verification: the number of hash-compression invocations. To our knowledge, the practical challenge of devising an ADS which efficiently adapts to evolving access frequencies while explicitly accounting for restructuring cost has not been addressed before.

Given these gaps, we ask:

> *Can we design efficient authenticated data structures for applications where elements may be updated and access frequencies can change?*

1.1 This Work

We introduce the Huffman-Merkle Tree (HMT), an efficient frequency-aware ADS for dynamic workloads. HMT separates state into tiers: frequently-accessed "hot" items reside in a small Huffman Merkle Hash Tree (HuffMHT) and a balanced Merkle Tree (MT) stores cold elements. A single global commitment authenticates all tiers. Our tiered approach is designed to obtain the "best of both worlds": HuffMHT obtains good average runtime complexity for operations involving hot items, while the cold tier is flatter and thus more suitable for efficient insertions of new elements given there are many rarely-accessed items. We furthermore optimize our ADS for dynamic workloads by batching and incrementally updating its layout when access frequencies change, rather than completely rebuilding it.

Our approach builds on two observations. First, authentication can be separated from layout optimization, that is, value updates are reflected in the next committed root of an item's tier, while layout changes are batched. Items admitted to the hot tier between rebuilds are authenticated through an overflow MT, so HMT never needs to restructure the hot

HuffMHT after every access. Second, given a monotone initial frequency distribution, a single access requires only swapping two nodes and updating hashes along the affected paths. The cost of a single swap is proportional to the distance between the relevant pair of nodes, and several swaps can be batched to avoid redundant hashing and thus lower update overhead.

HMT uses lightweight, deterministic metadata to decide which elements should be considered as hot. A Count-Min Sketch (CMS) efficiently estimates access frequencies without maintaining exact counters for every observed key [10]. A bucketed least-frequently-used (LFU) cache tracks candidate elements for promotion and demotion. On top of this, HMT supports several migration policies: (1) Ratio-Based uses lifetime access rate and is simple and stable. (2) Sliding-Window uses recent accesses and is more responsive to workload shifts. (3) Dynamic-Control adjusts admission thresholds and tier capacity using feedback from occupancy and rejected promotions. Policy decisions are batched; in our blockchain evaluation, this takes place at the end of the block-building process.

Using real-world data, we compare five ADSs (MPT, UBT, and three HMT configurations) using two metrics. First, commit-time hash input measures the number of hashed bytes, thus capturing the hashing work induced by updates while avoiding implementation- and hardware-specific differences. Second, *average* membership-proof length, weighted by the frequency with which each item is accessed.

Our evaluation shows that HMT improves on prominent ADSs, and that Sliding-Window is the best-performing policy: it uses about 2.4× less average hash operations than MPT and about 0.34× less than UBT. Its access-weighted proofs are also substantially smaller, reducing the plotted proof sizes by 0.18× than MPT and 0.55× than UBT. Ratio-Based HMT is slightly less responsive but remains stable, while Dynamic-Control HMT demonstrates an adaptive admission mechanism that remains competitive in proof size at the cost of additional hash-input work in this run. These results indicate that frequency-aware authenticated layouts can improve both witness size and update overhead on real workloads.

Contributions To summarize our contributions:

1. We propose HMT, a frequency-aware ADS which exploits skewed access distributions to tailor performance to real-world workloads while supporting updates, insertions, deletions, deterministic root computation, and bounded restructuring work. HMT combines a cold authenticated tier, a periodically rebuilt HuffMHT hot tier, and an overflow tree for newly hot elements admitted between rebuilds.

2. We develop efficient mechanisms for adapting HMT under dynamic workloads, including Count-Min Sketch (CMS) frequency estimation, a bucketed least-frequently-used (LFU) promotion cache, lifetime-rate promotion, sliding-window promotion with lazy demotion, and feedback-controlled admission.

3. We implement HMT and evaluate it on real-world Ethereum data, comparing its performance against Ethereum's current MPT design and its proposed UBT replacement. The evaluation shows HMT obtains improved performance with respect to both hashing overhead and access-weighted membership-proof size.

Paper Structure In Section 2, we cover preliminaries required for our work. Next, we describe different approaches for efficiently optimizing ADS layouts when item access frequencies can dynamically change in Section 3. We proceed by specifying the tiered architecture of our HMT in Section 4, and its dynamic tier adjustment policies in Section 5. Then, we evaluate several HMT configurations against prominent ADSs in Section 6. Finally, we go over related work in Section 7 and conclude in Section 8. In the interest of space, some algorithmic details are relegated to Section A and part of the evaluation to Appendix B.

2 Preliminaries

We now cover the foundations of ADSs, their application to blockchain state management, and the principles of coding theory utilized in our proposed architecture.

2.1 Authenticated Data Structures

An ADS allows a prover to convince a verifier that an element is a member of a dataset represented by a succinct commitment, without requiring the verifier to see the entire dataset. ADSs expose an API for reading data, generating membership proofs, updating data, committing to the current state, and verifying membership proofs against the committed state, as formalized in Definition 2.1.

Definition 2.1 (Authenticated Data Structure Interface)**.** *An Authenticated Data Structure (ADS) over a dataset D of key-value pairs (k, v) exposes the following methods:*

- $\mathsf{Put}(D, k, v) \to D'$*: inserts or updates initial state D with the key-value pair (k, v), producing an updated state D' such that $(k, v) \in D'$.*
- $\mathsf{Delete}(D, k) \to D'$*: removes a key-value pair (k, v) from initial state D, returning an updated state $D' \leftarrow D \setminus \{(k, v)\}$.*
- $\mathsf{Commit}(D) \to c$*: returns a commitment c of D.*
- $\mathsf{Get}(D, k) \to \{v, \bot\}$*: returns the value v associated with the key k in state D if it is indeed present. Otherwise, k is absent and $\bot$ is returned.*
- $\mathsf{GetProof}(D, k) \to \{\pi, \bot\}$*: returns a proof $\pi(k, v, c)$ against $c = \mathsf{Commit}(D)$ if key k is present in state D. Otherwise, returns $\bot$.*
- $\mathsf{Verify}(c, k, v, \pi) \to \{0, 1\}$*: checks whether a proof π authenticates key-value pair (k, v) against a commitment c, where for $\pi = \mathsf{GetProof}(D, k)$ and given $c = \mathsf{Commit}(D)$, then $Verify(c, k, v, \pi)$ returns 1 iff $(k, v) \in D$.*

2.2 Merkle Trees (MTs)

A Merkle Tree (MT) is a cryptographic accumulator where elements reside in leaf nodes [26]. Each internal node stores the cryptographic hash of the concatenation of its children's hashes,

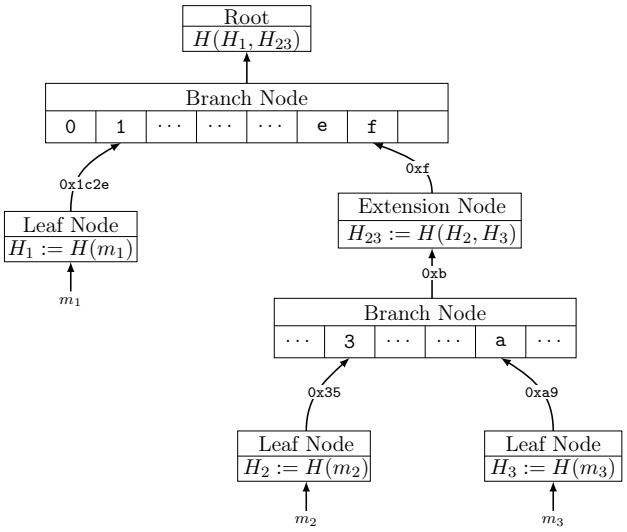


Figure 2: An example of a Merkle Patricia Trie (MPT).

and the tree's root serves as a commitment to the entire dataset. We describe MT inclusion-proof generation and verification in Algorithm 1. Briefly, an inclusion proof for a leaf x consists of the sibling hash and left/right orientation at each level on the path from x to the root. Thus, proof size is logarithmic in the number of leaves N: $O(\log_2 N)$. Given a proof, a verifier hashes x's value, iteratively combines the running hash with each sibling in the indicated order, and accepts if the final hash equals the trusted root commitment.

2.3 Blockchain State

Ethereum processes state-modifying transactions in batches called blocks, where each blocks contains a commitment to the blockchain's state up to that point [41]. The Merkle Patricia Trie (MPT) ADS is currently used to maintain this commitment, with the Unified Binary Tree (UBT) proposed as a future replacement [9]. We briefly describe both here for the sake of completeness, as we compare our solution with these structures in Section 6.

Merkle Patricia Trie (MPT) The MPT supports membership and absence proofs, but its witnesses can be large because proofs contain encoded trie nodes, rather than the standard MT's approach of having just one sibling hash per level. In Ethereum, a tree-of-tries layout is used for account and contract storage [41], with the choice of MPT specifically has been found to make Ethereum susceptible to Denial-of-Service (DoS) attacks [42]. At a high level, Ethereum's MPT proof interface converts the queried key into a hexary nibble path and returns the encoded trie nodes encountered along that path. Algorithm 2 illustrates this inclusion-proof procedure, and Fig. 2 shows an example MPT.

Unified Binary Tree (UBT) EIP-7864 is an Ethereum Improvement Proposal to replace the current MPT-based state ADS with one based on a Unified Binary Tree (UBT) [9]. It merges account data, storage, and code into one 32-byte key-value space, groups 256 related

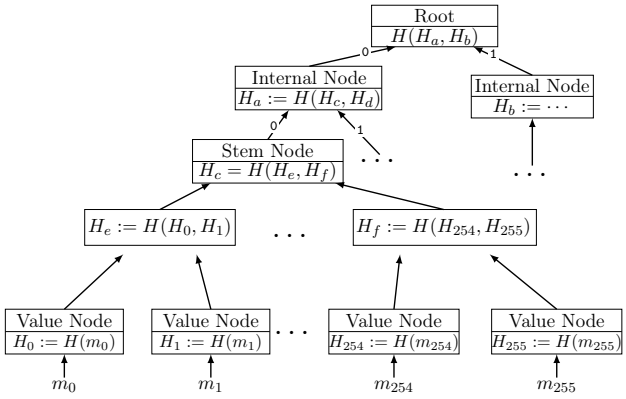


Figure 3: An example of Unified Binary Tree (UBT).

values by stem, and uses binary internal nodes to improve ordinary Merkle-witness size and proving-system compatibility [9]. A UBT proof follows the binary path determined by the first 248 bits of the key's 31-byte stem and then authenticates the selected value inside the 256-value stem subtree, rather than traversing Ethereum's current account/storage tree-of-tries. Figure 3 shows the corresponding stem-and-subtree structure used by UBT.

2.4 Frequency-Based Structures

Huffman Coding Huffman coding is a foundational entropy encoding algorithm used for lossless data compression [19, 29]. Given a distribution P over symbols, it constructs a variable-length prefix code represented by a binary tree in which higher-probability symbols receive shorter root-to-leaf paths. Huffman coding minimizes the expected prefix-code length $\sum_x p(x)L(x)$. Note that individual codeword lengths need not equal $\log_2 \frac{1}{p(x)}$.

Huffman Merkle Hash Tree (HuffMHT) The HuffMHT uses a frequency-aware layout to reduce access-weighted authentication path length for static access distributions [30]. It starts from a Huffman tree over element access weights, stores elements at the leaves, stores child-hash commitments at internal nodes, and uses the root hash as the commitment to the set of values. The authentication interface is the same as in a binary Merkle tree, where an inclusion proof for element x consists of the sibling hashes along the root-to-leaf path, and the verifier recomputes the root from the claimed value and the proof. The proof depth for x is therefore the Huffman code length $L(x)$, so the same expected-length objective $\sum_x p(x)L(x)$ becomes the access-weighted expected proof depth.

Figure 1 contrasts a balanced MT with a Huffman layout for four elements whose access frequencies decrease from m_1 to m_4. Real-world data often exhibits Zipfian skew, e.g., word frequencies in natural text and city population sizes [31]. Figure 4 reports the same effect across larger skewed workloads, with the i-th element accessed with probability proportional to $(i + b)^{-a}$, where $a = 4$ and $b = 1$. However, the basic HuffMHT layout is static, so the novelty of HMT is not the use of a Huffman-shaped authenticated tree itself, but the dynamic authenticated-map design that maintains such a layout under insertions, updates, and changing access frequencies.

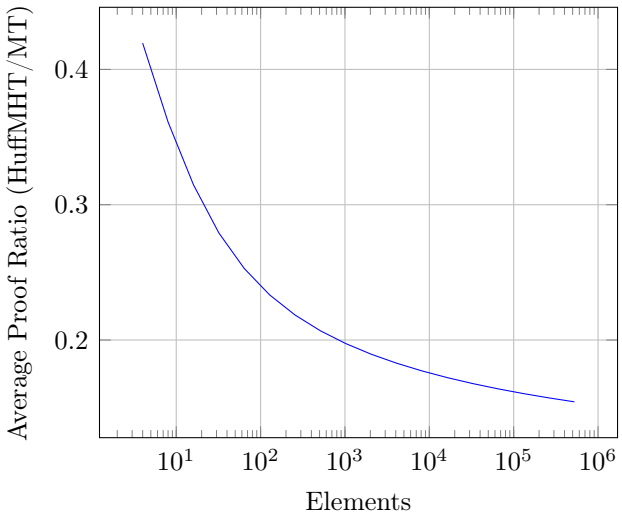


Figure 4: Average proof-size ratio of HuffMHTs to balanced MTs as the number of elements increases under Zipf-distributed accesses, with $p_i \propto (i+b)^{-a}$, $a = 4$, and $b = 1$.

3 Handling Dynamic Access Frequencies

We tailor our solution to efficiently handle dynamic workloads where new items may be continually added and access frequencies can change over time. This contrasts with the standard Huffman coding approach, which considers a fixed frequency distribution over a known element set. We now go over several different methods for the dynamic setting, starting with completely rebuilding the structure whenever access frequencies change a la [28], and continuing with two novel which we use in HMT.

Complete Rebuild Strategy One update approach is to recompute the entire tree each time, whether for single changes or batches of them. The cost each time is linear in the number of elements (and not in the delta that changed).

Incremental Update Strategy If an item subset S is accessed in some period, then only the ranking of items in S in the access-frequency sequence ordering changes. Denote frequencies before and after the update respectively by $f(\cdot)$ and $f'(\cdot)$. Let elements be sorted in descending access frequencies $e_1, e_2, \ldots, e_n$. For brevity and without loss of generality, suppose only e_2 changed its relative access frequency such that $e_{j-1} < e_2 < e_j$. That is:

$$f(e_1) < \mathbf{f(e_2)} < \cdots < f(e_{j-1}) < f(e_j) < \cdots < f(e_n);$$
$$f'(e_1) < \cdots < f'(e_{j-1}) < \mathbf{f'(e_2)} < f'(e_j) < \cdots < f'(e_n).$$

Swapping e_2 with e_j fixes the code. This entails moving e_3 to e_2's place, e_4 to e_3, and so on, up to e_2 in e_{j-1}'s place. In total, $|S|$ pair-swaps transform f to f'. For each $e_i \in S$ whose access frequency increased from $f(e_i)$ to $f'(e_i)$, find a pair-swap element e_j for e_i where:

$$e_j \triangleq \arg\min_e \{f(e) \mid f(e) > f(e_i), f'(e) < f'(e_i)\}$$

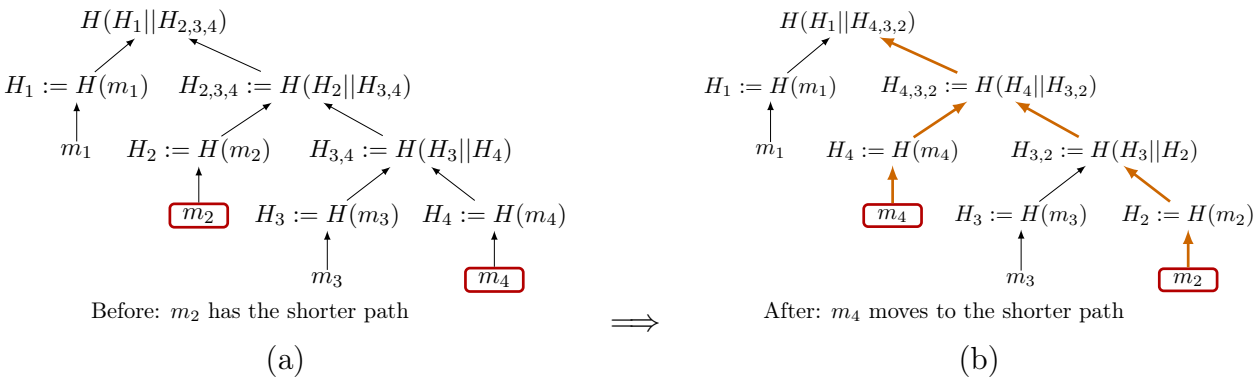


Figure 5: A local pair-swap in HuffMHT. If m_4 is accessed more frequently than m_2, the update swaps their positions. Only hashes on the highlighted ancestor paths are recomputed.

That is, e_j is the item whose access frequency is the minimal such that e_i and e_j swap places in access-frequency ranking order. The tree can be updated from f to f' by performing these (e_i, e_j) pair-swaps and updating hashes along the affected paths. The cost of a swap is proportional to the distance between the corresponding pair of nodes. Reflecting all these changes in the tree can be accomplished with $|S|$ pair-swaps. Figure 5 illustrates this local update for a single pair-swap; where an element's frequency crosses another's, the two elements exchange positions and only the highlighted ancestor hashes are recomputed.

Continuous Insertions and Periodic Rebuild The discussion above focuses on changes in the frequencies of existing elements. However, in settings such as blockchain and certificate transparency, new elements will appear continuously. Prior work proposed an adaptive variant of Huffman coding for this setting because it supports insertions and updates the code tree as accesses arrive [22, 39]. However, applying this variant to an ADS may perform suboptimally, as we show in Section 6.1.1. The intuitive reason for this is because each structural change to the tree also requires to perform the corresponding hash maintenance before the next root can be committed.

Our implementation instead keeps a base HuffMHT for elements that have already been absorbed into the optimized layout, and stores newly inserted or not-yet-absorbed elements in an overflow MT. We refer to this as the *periodic-rebuild* approach. The base and overflow components are kept key-disjoint. Elements stored in the overflow tree remain there until the next rebuild incorporates them into the base HuffMHT. When an existing element is updated, the periodic-rebuild HuffMHT updates the leaf value and marks the ancestor path dirty, so the next root computation reflects the new value. The layout changes are delayed until the rebuild boundaries. When the overflow tree is non-empty, the periodic-rebuild HuffMHT root is computed as $H(\mathsf{base\text{-}overflow}\|r_{\text{base}}\|r_{\text{overflow}})$, where r_{base} is the base HuffMHT root and r_{overflow} is the overflow Merkle root. When the overflow MT is empty, the root is computed as $H(\mathsf{base\text{-}only}\|r_{\text{base}})$. A proof against this root includes a component tag indicating whether the claimed key is authenticated by the base HuffMHT or by the overflow MT. At a rebuild boundary, the periodic-rebuild HuffMHT merges the live base elements with the overflow elements, rebuilds the base HuffMHT from the latest frequency

counts, and clears the overflow tree. Thus, it handles dynamic insertions without reshaping the Huffman tree after every operation. We evaluate this design choice in Section 6.1.1.

4 Tiered Tree Architecture

The intuition behind HMT is that element access patterns may be highly skewed, where a small and shifting subset of elements are more frequently accessed than others over a given period of time. To optimize update overhead and membership proof length for different access patterns, we store items in different tiers according to their popularity. To account for changes in access frequencies, tier migration is applied deterministically at period boundaries before the next global root is committed.

In our evaluation, the system operates with two tiers:

- **Cold Tier:** A balanced binary MT stores infrequent and newly inserted elements, providing logarithmic update paths in the cold tier size.
- **Hot Tier:** Select high-frequency elements are stored in a HuffMHT, reducing access-weighted proof size and path-rehashing work for those elements.

For such mutable workloads, a single frequency-optimized HuffMHT shortens paths for hot elements by placing colder elements deeper in the tree. As a result, updates to cold elements can become expensive. This motivates the two-tier instantiation above, where cold and newly inserted elements remain in the MT, while elements with sufficient observed access frequency migrate into the periodic-rebuild HuffMHT hot tier defined in Section 3.

Why the Cold Tier Uses a Merkle Tree Cold and newly inserted elements have not yet accumulated enough accesses to benefit from a frequency-optimized layout. Keeping them in a balanced MT avoids assigning them to the long tail of a Huffman-shaped tree. A single HuffMHT gives short paths to hot elements by assigning longer paths to colder ones. With N leaves, the deepest HuffMHT leaf can have depth $N-1$, compared with $\lceil \log_2 N \rceil$ in a balanced binary MT. Since inserts and value updates rehash the authenticated path to the root, this tail depth can make cold-element maintenance linear in the number of leaves. Figure 6 plots the corresponding worst-case depth ratio using $N/\log_2 N$. HMT keeps cold and new elements in an MT to avoid this tail cost, and migrates elements into the HuffMHT only after their observed frequency warrants shorter hot tier paths.

Since the benefit of the hot tier requires tracking the most frequently accessed items, HMT periodically reevaluates access frequencies and migrates items between tiers accordingly. This adaptive migration is based on the following promotion and demotion rules, whose candidate selection relies on Promotion Cache metadata that tracks tier membership and frequency order for hot tier elements and cold tier promotion candidates, as described in Section 4.3.

Promotion HMT monitors access frequencies. If the highest-scoring lower-tier candidate exceeds a threshold, it becomes eligible for promotion to the next hotter tier.

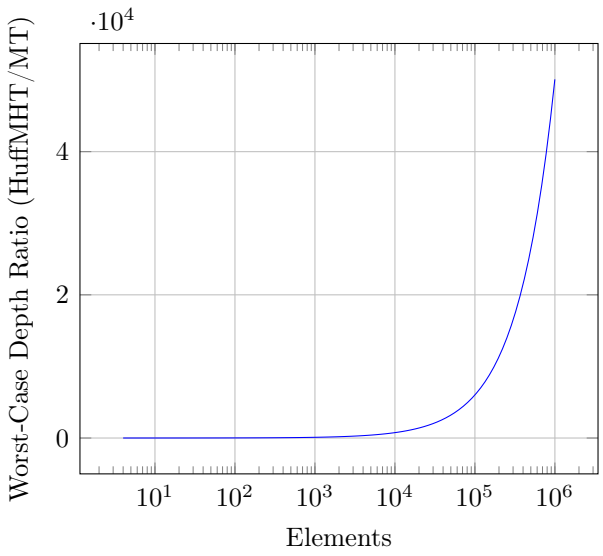


Figure 6: Worst-case depth ratio for a cold item in a HuffMHT vs. a balanced binary MT.

Demotion HMT supports movement from a hotter tier to a colder one. The active migration policy determines when an element should leave its current hot tier.

4.1 Membership-proof Soundness

Tiering is internal to HMT, while the interface described in Section 2.1 is exposed externally. We provide a high-level description of the interface in Algorithm 3. To allow one root to represent the entire state and maintain cryptographic integrity, the global root is calculated by concatenating the hashes of all individual root tiers and hashing the result. Given a hash-function H and T root tiers $\{\text{Root}_i\}_{i\in\{0,\dots,T-1\}}$, we define the global root $\text{Root}_{\text{global}}$:

$$\text{Root}_{\text{global}} = H\left(\text{Root}_0 \| \text{Root}_1 \| \cdots \| \text{Root}_{T-1}\right)$$

We now show membership-proof soundness for HMT. Relative to an authenticated root c, no probabilistic polynomial-time adversary can output (k, v, Π) such that $\mathsf{Verify}(c, k, v, \Pi) = 1$ but (k, v) is not represented by the committed tier roots, except with negligible probability.

We assume that H is collision resistant and that all hashed inputs are prefix-free and domain separated, including leaf encodings, internal-node encodings, empty roots, tier-root tuples, and global-root tuples. Throughout the algorithms, every displayed concatenation is shorthand for this canonical domain-separated encoding. We also assume that the committed state is key-disjoint across tiers, so at any committed root, each live key appears in exactly one tier. The adversary is given the public algorithms, tier policy, public metadata, and commitment c, and may choose any purported value and proof. The committed dataset is the state represented by the honest tier roots at the time c is produced.

Proposition 4.1 (Multi-tier Membership Soundness)**.** *Let $D_0, \dots, D_{T-1}$ be key-disjoint tier datasets, and let each tier i be represented by a membership-sound ADS $\mathcal{A}_i$ with root $r_i = \mathcal{A}_i.\mathsf{Commit}(D_i)$. Let the global commitment be the domain-separated hash of the ordered tuple*

$(r_0, \ldots, r_{T-1})$*; in Algorithm 3, the displayed concatenation denotes this fixed tuple encoding. So the resulting ADS is membership sound.*

Proof. Suppose an adversary produces a false accepting proof for $(k, v) \notin \bigcup_i D_i$. Let $(r_0^\star, \ldots, r_{T-1}^\star)$ be the honest tier-root tuple and $(\hat{r}_0, \ldots, \hat{r}_{T-1})$ be the proof's tuple. If they differ, acceptance gives two distinct domain-separated inputs with the same hash, contradicting collision resistance. Otherwise, the local proof verifies against the honest root $r_i^\star$, although $(k, v) \notin D_i$, contradicting membership tier i's soundness. □

Proposition 4.2 (HuffMHT Membership Soundness)**.** *A HuffMHT with key-value leaves and ordered-child internal hashes is membership sound when H is collision resistant and inputs use the encodings above.*

Proof. The proof is similar to the canonical MT argument. A false accepting proof reconstructs the honest root from a claimed leaf and an ordered sequence of siblings. Comparing the reconstructed path with the honest tree, the first point at which different encoded inputs yield the same digest gives a collision in H. The argument is independent of balance because Huffman layout changes only path lengths, not the bottom-up authentication recurrence. □

Proposition 4.3 (HMT Membership Soundness)**.** *The HMT construction is membership sound when its constituent tier ADSs are membership sound and H is collision resistant.*

Proof. By Proposition 4.2, HuffMHTs are membership sound; ordinary Merkle trees are membership sound by the same argument. A periodic-rebuild HuffMHT is membership sound when its exported root is a domain-separated commitment to its mode and constituent roots. If the overflow tree is empty, the exported root commits to the base HuffMHT root in the base-only domain. If the overflow tree is non-empty, the exported root commits to the ordered pair consisting of the base HuffMHT root and the overflow Merkle root, and a proof identifies which component contains the claimed key. In either case, a false accepting proof yields either a false proof for the base HuffMHT, a false proof for the overflow MT, or a collision in the exported-root hash. Applying Proposition 4.1 to the resulting tier roots gives membership soundness of the global HMT commitment. The CMS, promotion cache, and migration policy only determine which tier stores each key before the next root is committed; they do not change verification once the tier roots are fixed. □

4.2 Frequency Tracking via Count-Min Sketch (CMS)

To dynamically optimize layout, HMT maintains access-frequency estimates for when items should move between tiers. Instead of exact per-item counters, we use Count-Min Sketch (CMS) [10], a $d \times w$ counter array. With $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$, CMS uses $O(wd)$ counters. For a fixed queried key x, its estimate $\hat{f}(x)$ satisfies $f(x) \leq \hat{f}(x) \leq f(x) + \epsilon T$ with probability at least $1 - \delta$, where $f(x)$ is the true frequency of x and T is the total observed accesses. This guarantee is per queried key, rather than over all observed keys at once.

E.g., with $\epsilon = \delta = 10^{-6}$ and 32-bit (4-byte) counters, CMS uses $4\lceil e/\epsilon \rceil \lceil \ln(1/\delta) \rceil \approx 152$ MB, and the per-query overestimation bound is at most 10^3 accesses when $T = 10^9$. In contrast, exact counting for 10^7 distinct 32-byte keys needs at least 360 MB for keys and counters, excluding hash-table overheads. So, after (ϵ, δ) are chosen, the CMS memory

budget is fixed rather than proportional to the number of distinct elements. HMT derives CMS row indices from one BLAKE3 hash computation [32], and uses conservative update to reduce overestimation in practice [16, 5]. The standard CMS bound assumes suitable row hash functions; our deterministic hash instantiation is an engineering choice so replicas derive the same metadata from the same access stream. A CMS update or query touches one counter in each of the d rows, so CMS maintenance costs $O(d)$ time per operation.

4.3 Promotion Cache

To coordinate element migration, HMT uses an in-memory *Promotion Cache* as policy metadata, not as authenticated state. For each tracked key, the Promotion Cache records its current tier and maintains the frequency order used for promotion and demotion decisions.

For a hot tier with capacity C_i, the corresponding Promotion Cache also has capacity C_i; every key in the hot tier ADS has a cache entry. Cold tier cache entries, by contrast, are only promotion candidates and may be evicted from metadata; hot tier entries record exact tier membership and must remain until the key is demoted or deleted. The cached frequency order identifies candidates for promotion to a hotter tier and cold keys for demotion. The cold tier may be much larger, so its Promotion Cache does not mirror the full cold tier ADS; it tracks only possible promotion candidates. Thus, CMS estimates the frequency of a queried key, and the Promotion Cache determines which keys are currently eligible for migration. CMS does not enumerate hot keys by itself; promotion and demotion candidates are selected only from the bounded set of keys tracked by the Promotion Cache. This means cold keys absent from the cache may be missed until they are observed again, and CMS overestimation can affect layout quality but not proof soundness. Promotion and demotion update both the affected ADS tiers and the corresponding Promotion Cache entries.

We implement each tier's cache by batching keys into frequency ranges of width R, with the replacement policy following the least-recently-used (LRU) method [24]. This reduces the number of bucket relocations and avoids moving keys after every frequency change.

All Promotion Cache instances share one CMS for frequency estimation. LOCATE scans the L caches and performs expected $O(1)$ key lookup in each, so it costs $O(L)$. In our evaluated configurations, L is fixed and small, so this lookup component is constant.

When a key is accessed, HMT updates its CMS frequency estimate in $O(d)$. If the key remains in the same frequency bucket, the update is $O(1)$; otherwise, finding or creating the destination bucket costs $O(K_i)$ in the worst case, where K_i is the number of active buckets in tier i. Eviction is $O(1)$ by removing the least-recent key from the lowest-frequency bucket, given maintained pointers to the minimum-frequency bucket. Algorithm 4 summarizes the Promotion Cache operations used to locate, update, and migrate elements.

The total space is $O(wd + \sum_i (N_i + K_i))$, where $O(wd)$ is the shared CMS space, $N_i \leq C_i$ is the number of cached keys in tier i, and K_i is the number of active buckets.

5 Dynamic Customization

HMT dynamically adjusts tiering periodically per batch of accesses, where batch (period) length is a tunable parameter. For each batch, migration rules are applied given three policy

Table 1: Policy choices for tier migration.

Choice	What it controls	Options
Promotion score	Which elements are considered hot.	Based on cumulative frequency, lifetime rate, or recent-window rate.
Evaluation schedule	When promotion and demotion are checked.	Every I periods, for different I values.
Parameter control	Whether hot tier threshold and capacity change over time.	Fixed or feedback-controlled.

choices (summarized in Table 1): how to score promotion candidates, how often to evaluate migration, and whether admission parameters remain fixed. We number tiers from coldest to hottest as $0, \ldots, L-1$. For adjacent tiers i and $i+1$, promotion edge i denotes the tier boundary that an element crosses when it moves from tier i to the hotter tier $i+1$.

5.1 Absolute Threshold

For promotion edge i, Absolute Threshold scores each candidate x tracked by the Promotion Cache for tier i according to its cumulative CMS estimate $\hat{f}(x)$. An item is eligible for promotion when $\hat{f}(x) > \theta_i$, where θ_i is the configured minimum cumulative frequency required for promotion from tier i to $i+1$. If the destination hot tier has room, x is inserted there and removed from the colder tier. If the hot tier is full, HMT compares the candidate with the least-frequent element already admitted to that tier. The swap happens only when the candidate's cumulative estimate is strictly larger. A potential drawback with this design is that counts never decay. An element that was hot early in the run can keep a large score after its activity drops, while a newly hot element may need many periods to catch up, which can be undesirable when workloads change rapidly. We provide the pseudocode in Algorithm 5.

5.2 Ratio-Based

Ratio-Based uses a lifetime access-rate score. For promotion edge i and number of elapsed periods B, it scores each Promotion Cache candidate x by $s_B(x) = \frac{\hat{f}(x)}{B}$. A candidate is eligible for promotion when $s_B(x) > \theta_i$, where θ_i is the configured minimum lifetime access rate for promotion across edge i. Because candidates considered at a given moment share the same denominator B, Ratio-Based ranks candidates in the same order as Absolute Threshold; equivalently, the eligibility test is $\hat{f}(x) > B\theta_i$. This normalization makes the promotion threshold grow with run length, but the policy relies on lifetime counts, so stale hot keys may remain favored and newly hot keys may take time to become promotion candidates.

5.3 Sliding-Window

Sliding-Window HMT targets workloads where hot-set changes are expected over short time scales. Unlike the lifetime-frequency policies, it does not use cumulative CMS estimates for promotion, since CMS counters do not expire accesses that leave the window. Instead, HMT maintains a ring of recent period histograms and an aggregate window histogram $H_{\mathcal{W}}$; between periods, it removes the oldest period histogram and adds the newest one.

Let the number of periods included in the sliding window be W. For element x, let $H_{\mathcal{W}}(x)$ be its frequency in the current window $\mathcal{W}$, and define: $s_W(x) = \frac{H_{\mathcal{W}}(x)}{W}$. For promotion edge i, a candidate x is eligible for promotion when $s_W(x) \geq \theta_i$, where θ_i is the configured minimum window-local access rate for promotion across edge i. Because accesses outside the window are removed from $H_{\mathcal{W}}$, stale hot items lose influence once they stop appearing in recent periods. A hot-tier item is not demoted immediately when its window-local rate falls below the threshold. The implementation places the stale hot element on a per-edge demotion wheel and rechecks it after a delay; demotion occurs only if the item still $s_W(x) < \theta_i$ when its wheel slot expires. This lazy demotion policy reduces churn for items oscillating around the boundary while preserving responsiveness to bursts. Algorithm 6 gives the pseudocode.

5.4 Dynamic-Control

Dynamic-Control HMT removes the need to choose one static promotion threshold and one static hot tier capacity for all workload phases. For promotion edge i, we maintain a threshold θ_i, hot tier capacity C_i, and two feedback loops.

Threshold Control Every M periods, the controller checks occupancy $\rho_i = n_i/C_i$ (where n_i is the current hot tier size) and rejection ratio $q_i = R_i/A_i$ (where A_i and R_i are the promotion attempts and rejected attempts in the window). The terms "high" and "low" are policy choices: one can configure an occupancy band $[\rho_{\text{low}}, \rho_{\text{high}}]$ and a rejection threshold q_{high}. If $\rho_i > \rho_{\text{high}}$ and $q_i > q_{\text{high}}$, the controller raises θ_i to admit fewer candidates and reduce churn. If $\rho_i < \rho_{\text{low}}$ for U consecutive control windows, it lowers θ_i. The threshold is clamped to $[\theta_i^{\min}, \theta_i^{\max}]$, where $\theta_i^{\min}$ is configured and $\theta_i^{\max}$ is the edge-i upper bound derived from the least-frequent element currently admitted to the destination tier $i+1$. This upper bound prevents the controller from setting the promotion threshold above the access rate at the tail of tier $i+1$. Thus, a candidate in tier i that is hotter than the current tail of tier $i+1$ can still be considered for promotion, allowing stale elements in tier $i+1$ to be replaced.

Capacity Control A loop evaluates the hot tier capacity in every K control window, i.e., every KM periods. The controller increases C_i if the hot tier remains congested while θ_i is near $\theta_i^{\max}$, and shrinks C_i if the tier is under-utilized while θ_i is near $\theta_i^{\min}$, so capacity changes only after threshold adjustment is exhausted.

The two loops run at different cadences. Threshold control reacts to short-term admission pressure, whereas capacity control adjusts the hot tier capacity only after that pressure persists across multiple threshold-control windows.

We report the parameter values used in our experiments in Section 6, but the design is not tied to those numeric choices. For replicated deployments, the parameters are fixed for a

run, and controller updates occur only at period boundaries. Given the same access stream and deterministic tie-breaking, replicas therefore make the same promotion and demotion decisions. Algorithm 7 gives the pseudocode.

6 Evaluation

We evaluate HMT using two types of benchmarks. First, controlled microbenchmarks evaluate the layout-update strategies introduced in Section 3. They measure the maintenance cost of rebuilding and updating Huffman-authenticated layouts under skewed workloads. Second, we employ a deterministic replay of real Ethereum state accesses at account granularity, where each Ethereum account is treated as one authenticated element, and every compared ADS processes the same operation trace in the same order.

6.1 Layout-Update Microbenchmarks

Continuous Update Microbenchmark

We evaluate the incremental update strategy discussed in Section 3 by comparing two HuffMHT instances over the same key set: one performs a full rebuild at each milestone, while the other batches accesses between milestones and applies frequency-order pair swaps.

We first insert $N = 100{,}000$ keys into both instances, and then apply 5,000,000 get/update operations drawn from a Zipf distribution with $a = 4$, $b = 1$. This removes insertion cost from the comparison, so the measured milestone cost comes only from adapting the tree shape to changed access frequencies. At every 500,000-operation milestone, we compare rebuilding the Huffman tree from current frequencies against applying frequency-order pair swaps. Figure 7 reports the milestone restructuring time. Milestone 1 is used only to rebuild both instances from the frequencies observed during the first 500,000 operations. The full-rebuild and batched pair-swap strategies are compared from milestone 2 onward. Over milestones 2–10, swaps average 47.1 ms, while full rebuilds average 100.3 ms, a 2.13× reduction. This shows that incremental restructuring can reduce rebuild overhead for stable key sets.

6.1.1 Continuous Insert Microbenchmark

We next evaluate update time under continuous insertions. The workload starts from an empty tree and inserts exactly $N = 100{,}000$ keys over 5,000,000 operations. The first operation is forced to be an insert, and the remaining insert positions are randomly distributed across the run. Non-insert operations (get/update) access already-inserted keys under a Zipf distribution with $a = 4$, $b = 1$. We compare periodic-rebuild HuffMHT, which inserts new keys into an overflow MT and rebuilds every 500,000 operations, against an Adaptive Huffman tree, which updates its tree shape after every insert, get, or update. Figure 8 reports the update time measured for every 500,000-operation milestone interval. For periodic-rebuild HuffMHT, this time includes the operations performed during the interval, plus the rebuild time at the milestone. For Adaptive Huffman, this time includes the adaptive tree update performed after each operation during the same interval. At the final milestone, periodic-rebuild HuffMHT spends about 180 ms, while Adaptive Huffman spends about 136 seconds.

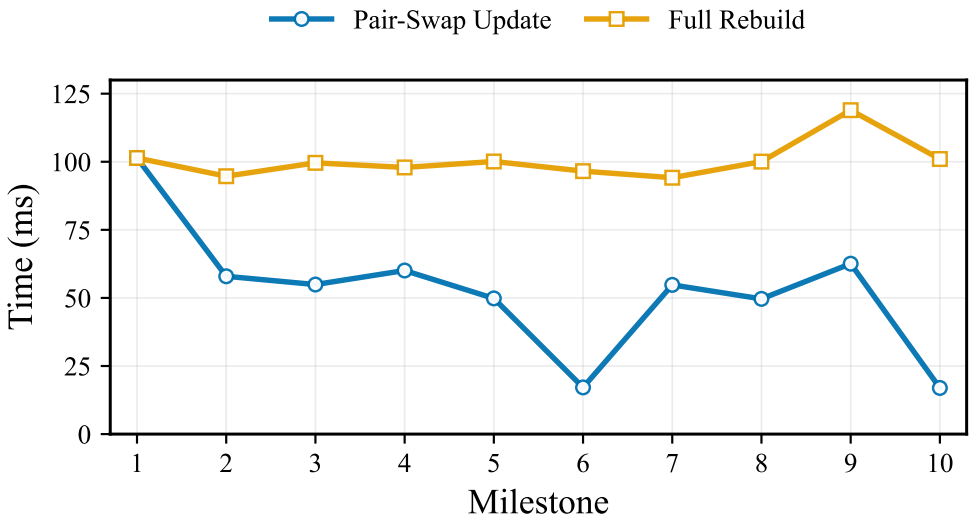


Figure 7: Pre-insert benchmark restructuring time.

This gap comes from the cost of maintaining an authenticated Huffman layout after every operation. Periodic rebuilds avoid this per-operation restructuring by putting new elements in the overflow MT and rebuilding the HuffMHT only at milestone boundaries.

6.2 Ethereum Replay Evaluation

6.2.1 Experimental Setup

Data The input is a trace of go-ethereum `StateDB` operations recorded during block execution. All compared ADSs process this trace in the same order. The replayer follows `snapshot`/`revertsnapshot` markers so reverted operations do not affect ADS state.

Every Ethereum account is mapped to one authenticated element. The trace records 20-byte Ethereum account addresses; before lookup or update, the replayer maps each address to the same 32-byte account key used by Ethereum's account trie. Storage operations are attributed to the owning contract account. A `getstate` contributes one read access to that account, and a `setstate` contributes one update access with the recorded storage value used as deterministic payload. At each `commit` marker, the replayer recomputes each ADS root and records the per-block total hash input and weighted average proof size for the block.

The replay starts at block 17,000,000 without reconstructing the full Ethereum pre-state. On first access to an account, every ADS materializes it with the same deterministic placeholder before applying the recorded operation. Thus, all ADS variants process the same account-level operations, in the same order, with deterministic payloads.

Compared Data Structures We compare the following:

- **MPT**: an Ethereum-style Merkle Patricia Trie baseline.

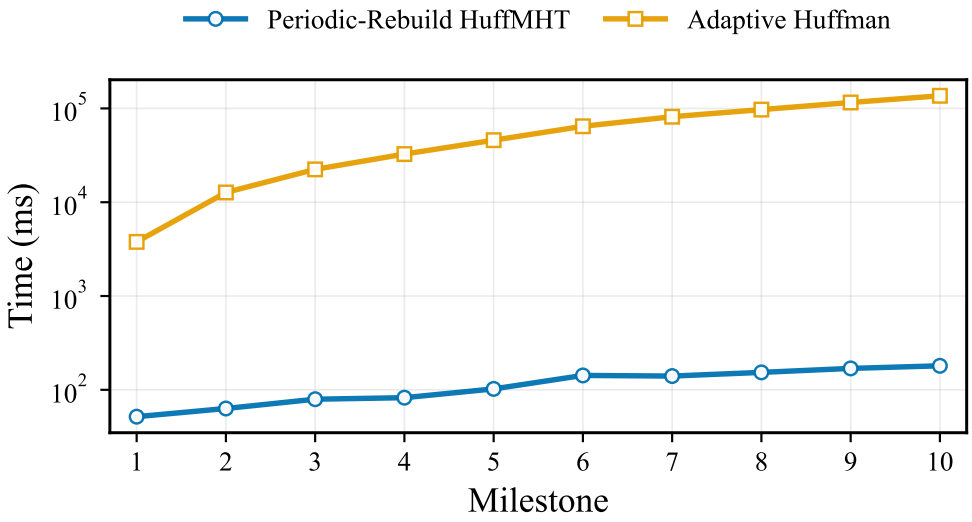


Figure 8: Update time under dynamic insertions.

- **UBT**: a unified binary authenticated tree baseline, representing the binary-tree direction considered for Ethereum state [9].
- **Ratio-Based HMT**: a two-tier configuration, where the hot and cold tiers are respectively instantiated using a HuffMHT and a MT. Promotion uses lifetime access rate, with threshold 0.05 and cold/hot promotion cache capacities of 8,000/16,000 entries.
- **Sliding-Window HMT**: the same two-tier configuration, but promotion uses only recent accesses in a 1,000-block rolling window. It uses a promotion-rate threshold of 0.05 and a 100-block lazy-demotion delay.
- **Dynamic-Control HMT**: an adaptive policy that starts from the Ratio-Based configuration and uses feedback to adjust hot tier capacity and promotion control. The controller uses $M = 2{,}500$-block threshold-control windows, evaluates capacity every $K = 3$ windows, targets occupancy band $[0.75, 0.95]$, uses rejection threshold $q_{\text{high}} = 0.3$, lowers thresholds after $U = 3$ low-occupancy windows, and clamps thresholds with $\theta_i^{\min} = 0.05$ and tail-derived $\theta_i^{\max}$.

Each HMT configuration rebuilds its HuffMHT every 500 blocks and uses LFU Promotion Cache buckets of range 10. Appendix B reports two additional HMT variants.

Metrics We report two per-block metrics. Different ADSs in our evaluation store different amounts of information per node. When they traverse their respective data structures to generate proofs, verify proof paths, or update the data structure, the computation cost is determined by the amount of data that is input to hashing. First, hashed input bytes measures the total number of bytes hashed during a block commit. We compute this quantity per authenticated node, using the byte string included in that node's hash input: MPT contributes trie-node bytes; Merkle and HMT leaves contribute a domain separator, key,

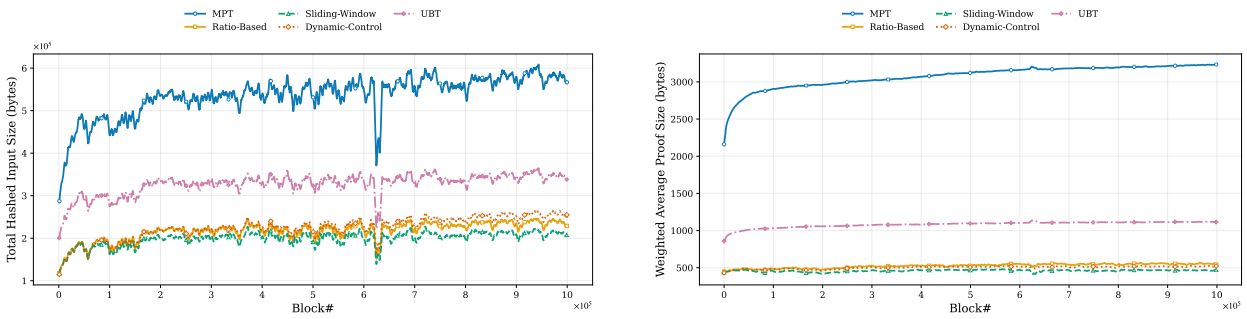


Figure 9: Performance comparisons of MPT, UBT, and HMT instantiations with various policies. The evaluation (fully described in Section 6.2) uses one million Ethereum blocks. The left panel shows the total bytes hashed during the processing of each block, and the bottom panel shows the weighted average inclusion proof size.

and value payload, while internal nodes contribute a domain separator and child hashes; UBT contributes the inputs to its stem and internal-node hashes. For HMT, we sum this quantity across tiers. Weighted average proof size measures the inclusion-proof size for keys accessed in a block, weighted by each key's access count in that block. Appendix C records this average per-block separately for read and write accesses.

Testbed We run all experiments on a Dell PowerEdge R7715 server with a 16-core AMD EPYC 9135 CPU and 32 GB of DRAM. Our Go implementation is compiled with Go 1.25.4.

6.2.2 Evaluation Results

Hash Input Bytes The left panel of Figure 9 shows the per-block hashed input bytes across the one-million-block replay. The x-axis reports the replay offset from block 17,000,000, and the y-axis reports the total number of bytes input to hash functions during the commit for that block. The visible dip around replay offset 6.2×10^5 is workload-driven rather than specific to one ADS. A 10,000-block summary of the trace shows that the average number of distinct updated account keys drops from about 68 per block to about 48 per block in this interval, even though total operation volume does not drop proportionally. Since repeated writes to the same account coalesce into the same dirty path at commit time, all ADSs perform less rehashing in this region. MPT has the largest byte-level hash workload throughout, while UBT remains the second most expensive method. The HMT variants consistently outperform both: Ratio-Based and Dynamic-Control stay close to one another, and Sliding-Window is the lowest-cost method for most blocks.

Proof Size Figure 9's right panel shows the weighted average proof size for the accounts accessed in each block. The x-axis reports the replay offset from block 17,000,000, and the y-axis reports the inclusion-proof size averaged over the keys accessed in that block, weighted by the number of times each key appears. MPT has the largest average proof size, around several kilobytes per accessed account. UBT reduces this cost by using a binary authenticated layout, but its proofs remain roughly twice as large as the HMT variants. The

HMT variants keep frequently accessed accounts in the hot HMT tier, so the access-weighted average is dominated by short hot tier paths rather than by cold tier paths. Sliding-Window is lowest or tied for lowest over most of the replay.

Policy Tradeoffs Across the hashed-input and proof-size figures, the three HMT policies expose different tradeoffs. Ratio-Based is simple and stable, but lifetime frequency can keep older hot accounts influential after the workload shifts. Sliding-Window reacts to recent locality and is the best policy in these figures for both hashed input bytes and weighted proof size. Dynamic-Control remains competitive in proof size but consumes more hashed input bytes than the static HMT policies in this run. We therefore treat Dynamic-Control as an exploratory adaptive policy rather than the primary winning configuration.

7 Additional Related Literature

We now go over additional relevant work.

Authenticated State Commitments Authenticated dictionaries, hash tables, and generic ADS frameworks provide the basic abstraction of committing to a mutable map and proving query answers [11, 33, 27]. Recent memory-checking work studies locality-optimized layouts for authenticated storage [40]. Blockchain state commitments impose additional constraints: roots are committed frequently, witnesses must be compact, and updates must be cheap. Ethereum's MPT supports membership and non-membership proofs through authenticated trie paths, but its encoded witnesses are large and updates touch many trie nodes [41]. Proposed Ethereum state-tree changes reduce different bottlenecks, including semantic key co-location and binary witnesses in UBT [9] and Verkle-tree migration proposals [4]. Related commitment schemes target stateless validation and efficient proof updates using accumulators, key-value commitments, and vector commitments [6, 1, 38]. Storage-oriented authenticated-state systems such as mLSM, RainBlock/DSM-Tree, Jellyfish Merkle Tree, LVMT, QMDB, and AlDBaran instead focus on improving storage locality, reducing I/O, lowering update cost, or simplifying state management [35, 34, 15, 23, 43, 21]. Unlike Verkle and vector-commitment proposals, HMT does not change the commitment primitive; unlike storage-oriented designs, it targets the authenticated layout rather than primarily the backing store or I/O path. Its main feature is to adapt short authenticated paths to keys that are hot under the observed workload while preserving a hash-based state commitment.

Frequency-aware Trees Some tree-based ADSs opt for unbalanced skewed constructions [20], with such designs being applied to various settings, e.g., certificate revocation [12] and memory integrity checking [37]. As we show, of particular relevance to blockchains is a specific design where frequently accessed elements are positioned closer to the root, because this can minimize the overhead of updating hot keys and the overall expected proof length. A natural approach for constructing such trees is to use Huffman coding, which is optimal in some cases [29] and can obtain a substantial improvement over vanilla MTs for certain access distributions. However, this optimality is achieved only when element accesses follow a static distribution and are independent and identically distributed, while blockchains are

dynamic and may also feature intricate access patterns (e.g., it is reasonable to expect that some decentralized exchanges are more likely to be accessed in tandem than others). Indeed, devising performant ADSs for the case where access distributions are dynamic is highlighted as a promising future direction by previous work [37], thus motivating our study.

Evaluating Dynamic Data Structures The common approach to evaluating the performance of dynamic data structure algorithms (e.g., insertion of a new key, or modifying an existing key's value) considers an online setting [36, 3, 2]. A sequence of requests arrives over time, and the examined algorithm must complete them without having any reliable forecast of future requests [18, 13, 14]. One benchmark used by the theoretic literature is the *competitive ratio*, i.e., the performance of a given online algorithm facing adversarially-generated requests is compared to an optimal offline one which has full knowledge of the future sequence of requests [8]. While this type of analysis adopts a worst-case perspective, our focus is on optimizing real-world average-case performance.

8 Conclusion

This paper addresses the cost of authenticated state access in dynamic, skewed workloads by proposing HMT, a frequency-aware ADS. The key idea is to decouple authentication from layout optimization, where cold elements remain in a conventional authenticated tier, frequently accessed elements move into a periodically rebuilt HuffMHT, and newly promoted hot elements are handled through an overflow tree until the next rebuild. This tiered design exploits access skew by giving hot elements shorter authenticated paths, while batching layout changes so that the tree does not need to be reshaped after every access.

Experiments on real-world Ethereum data show that this design improves several metrics. The best-performing policy, Sliding-Window, uses about 2.4× less average hash input than MPT ,and 0.34× less than UBT. Its access-weighted proofs are also smaller by 0.18× than MPT and 0.55× than UBT. The microbenchmarks support the same design choice at a smaller scale: batched pair-swap maintenance reduces Huffman layout restructuring time when the key set is stable, and periodic rebuilds avoid the high cost of updating an Adaptive Huffman tree after every operation under dynamic insertions.

References

[1] Shashank Agrawal and Srinivasan Raghuraman. "KVaC: Key-Value Commitments for Blockchains and Beyond". In: *Advances in Cryptology – ASIACRYPT 2020*. Ed. by Shiho Moriai and Huaxiong Wang. Cham: Springer International Publishing, 2020, pp. 839–869. ISBN: 978-3-030-64840-4. DOI: 10.1007/978-3-030-64840-4_28.

[2] James Aspnes. "Competitive Analysis of Distributed Algorithms". In: *Online Algorithms: The State of the Art*. Ed. by Amos Fiat and Gerhard J. Woeginger. Berlin, Heidelberg: Springer, 1998, pp. 118–146. ISBN: 978-3-540-68311-7. DOI: 10.1007/BFb0029567.

[3] Baruch Awerbuch, Shay Kutten, and David Peleg. "Competitive Distributed Job Scheduling (Extended Abstract)". In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*. STOC '92. New York, NY, USA: Association for Computing Machinery, July 1992, pp. 571–580. ISBN: 978-0-89791-511-3. DOI: 10.1145/129712.129768.

[4] Guillaume Ballet, Ignacio Hagopian, Gajinder Singh, Ansgar Dietrichs, Gottfried Herold, Jamie Lokier, Tanishq Jasoria, Parithosh Jayanthi, Gabriel Rocheleau, and Karim Taam. *EIP-7748: State conversion to Verkle Tree*. en. July 2024. URL: https://eips.ethereum.org/EIPS/eip-7748.

[5] Younes Ben Mazziane, Sara Alouf, and Giovanni Neglia. "Analyzing Count Min Sketch with Conservative Updates". In: *Computer Networks* 217 (2022), p. 109315. DOI: 10.1016/j.comnet.2022.109315.

[6] Dan Boneh, Benedikt Bünz, and Ben Fisch. "Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains". In: *Advances in Cryptology – CRYPTO 2019*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Cham: Springer International Publishing, 2019, pp. 561–586. ISBN: 978-3-030-26948-7. DOI: 10.1007/978-3-030-26948-7_20.

[7] Joseph Bonneau, Jessica Chen, Miranda Christ, and Ioanna Karantaidou. "Merkle Mountain Ranges Are Optimal: On Witness Update Frequency for Cryptographic Accumulators". In: *Advances in Cryptology – CRYPTO 2025*. Ed. by Yael Tauman Kalai and Seny F. Kamara. Cham: Springer Nature Switzerland, 2025, pp. 170–202. ISBN: 978-3-032-01878-6. DOI: 10.1007/978-3-032-01878-6_6.

[8] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge, United Kingdom: Cambridge University Press, 2005.

[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*. en. Jan. 2025. URL: https://eips.ethereum.org/EIPS/eip-7864.

[10] Graham Cormode and S. Muthukrishnan. "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications". In: *Journal of Algorithms* 55.1 (2005), pp. 58–75. DOI: 10.1016/j.jalgor.2003.12.001.

[11] Scott A. Crosby and Dan S. Wallach. "Authenticated Dictionaries: Real-World Costs and Trade-Offs". In: *ACM Trans. Inf. Syst. Secur.* 14.2 (Sept. 2011), 17:1–17:30. ISSN: 1094-9224. DOI: 10.1145/2019599.2019602.

[12] Farid F. Elwailly, Craig Gentry, and Zulfikar Ramzan. "QuasiModo: Efficient Certificate Validation and Revocation". en. In: *Public Key Cryptography – PKC 2004*. Ed. by Feng Bao, Robert Deng, and Jianying Zhou. Berlin, Heidelberg: Springer, 2004, pp. 375–388. ISBN: 9783540246329. DOI: 10.1007/978-3-540-24632-9_27.

[13] Yotam Gafni and Aviv Yaish. *Greedy Transaction Fee Mechanisms for (Non-)Myopic Miners*. 2022. DOI: 10/g9z6cx. arXiv: 2210.07793.

[14] Yotam Gafni and Aviv Yaish. *Scheduling with Time Discounts*. Feb. 2024. DOI: 10/g9z6mt.

[15] Zhenhuan Gao, Yuxuan Hu, and Qinfan Wu. "Jellyfish Merkle Tree". In: URL: https://api.semanticscholar.org/CorpusID:275924142.

[16] Amit Goyal, Hal Daumé III, and Graham Cormode. "Sketch Algorithms for Estimating Point Queries in NLP". In: *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Jeju Island, Korea: Association for Computational Linguistics, 2012, pp. 1093–1103. URL: https://aclanthology.org/D12-1100/.

[17] Dongchao Guo, Jiaqing Dong, and Kai Wang. "Graph Structure and Statistical Properties of Ethereum Transaction Relationships". In: *Information Sciences* 492 (2019), pp. 58–71. DOI: 10.1016/j.ins.2019.04.013.

[18] Bruce Hajek. "On the Competitiveness of On-Line Scheduling of Unit-Length Packets with Hard Deadlines in Slotted Time". In: *Proceedings of the 2001 Conference on Information Sciences and Systems*. 2001.

[19] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[20] Marek Karpinski and Yakov Nekrich. *A note on traversing skew Merkle trees*. Dec. 2004. URL: https://eccc.weizmann.ac.il/report/2004/118/.

[21] Bernhard Kauer, Aleksandr Petrosyan, and Benjamin Livshits. *AlDBaran: Towards Blazingly Fast State Commitments for Blockchains*. arXiv:2508.10493 [cs]. Aug. 2025. DOI: 10.48550/arXiv.2508.10493. URL: http://arxiv.org/abs/2508.10493.

[22] Donald E Knuth. "Dynamic huffman coding". In: *Journal of Algorithms* 6.2 (June 1985), pp. 163–180. ISSN: 0196-6774. DOI: 10.1016/0196-6774(85)90036-7.

[23] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. "LVMT: An Efficient Authenticated Storage for Blockchain". In: *ACM Trans. Storage* 20.3 (June 2024), 17:1–17:34. ISSN: 1553-3077. DOI: 10.1145/3664818.

[24] Silvano Maffeis. "Cache Management Algorithms for Flexible Filesystems". In: *ACM SIGMETRICS Performance Evaluation Review* 21.2 (Dec. 1993), pp. 16–25. ISSN: 0163-5999. DOI: 10.1145/174215.174219.

[25] Anna Mendonca, Hudson Shi, Triet (PJ) Huynh, Ivan Pryvalov, and Amir Herzberg. *Efficient Merkle-Tree Consistent Accumulator*. 2026. URL: https://eprint.iacr.org/2026/673.

[26] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology – CRYPTO '87*. Springer, 1987, pp. 369–378. DOI: 10.1007/3-540-48184-2_32.

[27] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. "Authenticated Data Structures, Generically". In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 411–423. ISSN: 0362-1340. DOI: 10.1145/2578855.2535851.

[28] Avi Mizrahi, Noam Koren, Ori Rottenstreich, and Yuval Cassuto. "Traffic-Aware Merkle Trees for Shortening Blockchain Transaction Proofs". In: *IEEE/ACM Transactions on Networking* 32.6 (Dec. 2024), pp. 5326–5340. DOI: 10.1109/TNET.2024.3466245.

[29] Alistair Moffat. "Huffman Coding". In: *ACM Comput. Surv.* 52.4 (Aug. 2019), 85:1–85:35. ISSN: 0360-0300. DOI: 10.1145/3342555.

[30] Jose L. Muñoz, Jordi Forné, Oscar Esparza, and Manel Rey. "Efficient Certificate Revocation System Implementation: Huffman Merkle Hash Tree (HuffMHT)". In: *Trust, Privacy, and Security in Digital Business*. Ed. by Sokratis Katsikas, Javier López, and Günther Pernul. Vol. 3592. Lecture Notes in Computer Science. Springer, 2005, pp. 119–127. DOI: 10.1007/11537878_13.

[31] MEJ Newman. "Power laws, Pareto distributions and Zipf's law". In: *Contemporary Physics* 46.5 (Sept. 2005), pp. 323–351. ISSN: 0010-7514. DOI: 10.1080/00107510500052444.

[32] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. *BLAKE3: One Function, Fast Everywhere*. https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf. 2020.

[33] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. "Authenticated Hash Tables". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. Ccs '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 437–448. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455826.

[34] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. "RainBlock: Faster Transaction Processing in Public Blockchains". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 333–347. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/ponnapalli.

[35] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. "mLSM: Making Authenticated Storage Faster in Ethereum". In: *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, 2018. URL: https://www.usenix.org/conference/hotstorage18/presentation/raju.

[36] Daniel D. Sleator and Robert E. Tarjan. "Amortized Efficiency of List Update and Paging Rules". In: *Commun. ACM* 28.2 (Feb. 1985), pp. 202–208. ISSN: 0001-0782. DOI: 10.1145/2786.2793.

[37] Jakub Szefer and Sebastian Biedermann. "Towards fast hardware memory integrity checking with skewed Merkle trees". In: *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '14. New York, NY, USA: ACM, June 2014, pp. 1–8. ISBN: 9781450327770. DOI: 10.1145/2611765.2611774.

[38] Ertem Nusret Tas and Dan Boneh. "Vector Commitments with Efficient Updates". In: *5th Conference on Advances in Financial Technologies (AFT 2023)*. Ed. by Joseph Bonneau and S. Matthew Weinberg. Vol. 282. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 29:1–29:23. ISBN: 978-3-95977-303-4. DOI: 10.4230/LIPIcs.AFT.2023.29.

[39] Jeffrey Scott Vitter. "Design and analysis of dynamic Huffman codes". In: *J. ACM* 34.4 (Oct. 1987), pp. 825–845. ISSN: 0004-5411. DOI: 10.1145/31846.42227.

[40] Weijie Wang, Yujie Lu, Charalampos Papamanthou, and Fan Zhang. "The Locality of Memory Checking". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 1820–1834. ISBN: 9798400700507. DOI: 10/g9z6jq.

[41] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger (Shanghai Version)*. 2025. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[42] Aviv Yaish, Kaihua Qin, Liyi Zhou, Aviv Zohar, and Arthur Gervais. "Speculative Denial-of-Service Attacks in Ethereum". In: *33rd USENIX Security Symposium (USENIX Security 24)*. Usenixsec '24. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 3531–3548. ISBN: 978-1-939133-44-1.

[43] Isaac Zhang, Ryan Zarick, Daniel Wong, Thomas Kim, Bryan Pellegrino, Mignon Li, and Kelvin Wong. *QMDB: Quick Merkle Database*. 2025. DOI: 10.48550/arXiv.2501.05262.

A Algorithms

Algorithm 1: Merkle Tree Inclusion Proof

Input:
$\mathcal{T}$: Binary Merkle tree.
c: Trusted root commitment.
v: Claimed leaf value.
Output: A Merkle proof π, or an accept/reject bit.

Procedure *GetProof*$(\mathcal{T}, k)$**:**
- $\pi \leftarrow [\,]$;
- $u \leftarrow$ leaf storing (k, v);
- **if** $(k, v) \in u$ **then**
  - Append (k, v) to π;
- **else**
  - **return** $\perp$
- **while** u *is not the root* **do**
  - Append $(\mathsf{side}(u), \mathsf{hash}(\mathsf{sibling}(u)))$ to π;
  - $u \leftarrow$ parent of u;
- **return** π;

Procedure *Verify*(c, k, v, π)**:**
- $h \leftarrow H((k, v))$;
- **foreach** $(b, s) \in \pi$ *from leaf to root* **do**
  - **if** $b =$ *left* **then**
    - $h \leftarrow H(h \,\|\, s)$;
  - **else**
    - $h \leftarrow H(s \,\|\, h)$;
- **return** $h = c$;

Algorithm 2: MPT Inclusion Proof

Input:
$\mathcal{T}$: Merkle Patricia Trie.
c: Trusted root hash.
k: Queried key.
v: Claimed value.

Output: An MPT proof π, or an accept/reject bit.

Procedure *GetProof*$(\mathcal{T}, k)$**:**
- $p \leftarrow$ Nibbles(k);
- $u \leftarrow$ root node of $\mathcal{T}$; $\pi \leftarrow [\,]$;
- **while** *u is on the search path for p* **do**
  - Append Encode(u) to π;
  - $u \leftarrow$ next trie node selected by the remaining nibbles of p;
- **return** π;

Procedure *Verify*(c, k, v, π)**:**
- $p \leftarrow$ Nibbles(k);
- $r \leftarrow c$;
- **foreach** *encoded node $e \in \pi$ from root to leaf* **do**
  - **if** $H(e) \neq r$ **then**
    - **return** 0 ;
  - $u \leftarrow$ Decode(e);
  - **if** *u contains the value for the remaining path p* **then**
    - $\hat{v} \leftarrow$ value stored at u;
    - **break**;
  - **else**
    - $(r, p) \leftarrow$ next child commitment and remaining path selected by p;
- **return** $\hat{v} = v$;

Algorithm 3: HMT

State:

$\mathcal{A}_i$: ADS storing tier i.
P_i: tier-i Promotion Cache.
r_i: tier-i root.

Procedure *PUT*(k, v)**:**

$i \leftarrow$ LOCATE(k);
$\mathcal{A}_i$.PUT(k, v);
INSERTINTIER(i, k);

Procedure *DELETE*(k)**:**

$i \leftarrow$ LOCATE(k);
if $\mathcal{A}_i$.*GET*$(k) = \perp$ **then**
  return;
$\mathcal{A}_i$.DELETE(k);
REMOVE(k);

Procedure *COMMIT*$()$**:**

for $i \leftarrow 0$ **to** $T - 1$ **do**
  $r_i \leftarrow \mathcal{A}_i$.COMMIT();
$c \leftarrow H(r_0 \,\|\, \cdots \,\|\, r_{T-1})$;
return c;

Procedure *GET*(k)**:**

$i \leftarrow$ LOCATE(k);
$v \leftarrow \mathcal{A}_i$.GET$(k)$;
if $v = \perp$ **then**
  return $\perp$;
if k *is cached in* P_i **then**
  TOUCHINTIER(i, k);
else
  INSERTINTIER$(0, k)$;
return v;

Procedure *GETPROOF*(k)**:**

$i \leftarrow$ LOCATE(k);
$\pi_i \leftarrow \mathcal{A}_i$.GETPROOF$(k)$;
if $\pi_i = \perp$ **then**
  return $\perp$;
$\Pi \leftarrow (i, \pi_i, (r_0, \ldots, r_{T-1}))$;
return Π;

Procedure *VERIFY*(c, k, v, Π)**:**

Parse $\Pi = (i, \pi_i, (r_0, \ldots, r_{T-1}))$;
if $\mathcal{A}_i$.*VERIFY*$(r_i, k, v, \pi_i) = 0$ *or* $H(r_0 \,\|\, \cdots \,\|\, r_{T-1}) \neq c$ **then**
  return 0;
return 1;

Algorithm 4: Bucketed LFU Promotion Cache

Configuration:

T: Number of tiers.

R: Frequency-bucket span.

C_i: capacity of tier-i metadata cache.

State: $\mathcal{S}$: Shared CMS.

M_i: Key-to-node hash map for tier i.

B_i: Doubly-linked list of tier i's frequency buckets.

Procedure *Locate(k)*:

- **for** $i \leftarrow 0$ **to** $T-1$ **do**
  - **if** $k \in M_i$ **then**
    - **return** i;
- **return** 0 as base-tier fallback;

Procedure *Delete(k)*:

- $i \leftarrow$ Locate(k);
- **if** $i = 0 \wedge k \notin M_0$ **then**
  - **return** not cached;
- Delete $M_i[k]$ from B_i; delete $M_i[k]$;

Procedure *TouchInTier(i, k)*:

- $n \leftarrow M_i[k]$;
- update $\hat{f}$ of k in $\mathcal{S}$;
- RelocateIfNeeded$(i, n, \hat{f})$;

Procedure *RelocateIfNeeded(i, n, $\hat{f}$)*:

- **if** $\hat{f}$ *is inside* n*'s current bucket range* **then**
  - move n to the front of that bucket list;
- **else**
  - Remove n from current bucket;
  - $b \leftarrow$ find-or-create bucket in B_i for range of $\hat{f}$ with span R;
  - Insert n at front of b;

Procedure *InsertInTier(i, k)*:

- **if** $k \in M_i$ **then**
  - TouchInTier(i, k);
  - **return**;
- Update $\hat{f}$ of k in $\mathcal{S}$;
- $b \leftarrow$ find-or-create bucket in B_i for range of $\hat{f}$ with span R;
- Create node $n = (k, b)$, insert at front of b;
- $M_i[k] \leftarrow n$;
- **if** $|M_i| > C_i$ **then**
  - Evict tail of minimum-frequency bucket in B_i;

Procedure *MoveTier(k, i, j)*:

- Remove k from M_i and its bucket in B_i;
- InsertInTier(j, k);

Algorithm 5: Absolute Threshold Migration

Parameters:

T: Number of tiers.

C_i: Promotion Cache capacity for tier i.

θ_i: Absolute promotion threshold for edge i.

State:

$\mathcal{A}_i$: ADS storing tier i.

P_i: Promotion Cache for tier i.

```
// Process each promotion edge
```
for $i \leftarrow 0$ **to** $T - 2$ **do**

  $x \leftarrow \textsc{MostFrequent}(P_i)$;

  `// 1. Fill spare capacity in the hotter tier`

  while $x \neq \perp \wedge |P_{i+1}| < C_{i+1} \wedge \hat{f}(x) > \theta_i$ **do**

    Promote x from $\mathcal{A}_i$ to $\mathcal{A}_{i+1}$;

    $\textsc{MoveTier}(x, i, i+1)$;

    $x \leftarrow \textsc{MostFrequent}(P_i)$;

  `// 2. If full, admit only if hotter than tail`

  if $|P_{i+1}| = C_{i+1}$ **then**

    $z \leftarrow \textsc{LeastFrequent}(P_{i+1})$;

    if $x \neq \perp \wedge z \neq \perp \wedge \hat{f}(x) > \theta_i \wedge \hat{f}(x) > \hat{f}(z)$ **then**

      Promote x from $\mathcal{A}_i$ to $\mathcal{A}_{i+1}$;

      Demote z from $\mathcal{A}_{i+1}$ to $\mathcal{A}_i$;

      $\textsc{MoveTier}(x, i, i+1)$;

      $\textsc{MoveTier}(z, i+1, i)$;

Algorithm 6: Sliding-Window Migration

Input:

B: Access histogram for the completed period.

Parameters:

T: Number of tiers.
θ_i: Window-rate threshold for edge i.
W: Sliding-window length, in periods.
Δ_i: Delayed-demotion interval for edge i.

State:

$\mathcal{W}$: Ring of recent period histograms.
$H_{\mathcal{W}}$: Aggregate histogram over $\mathcal{W}$.
$\mathcal{Q}_i$: Demotion queue for edge i.
$\mathcal{A}_i$: ADS storing tier i.

```
// 1. Slide the window
```
$E \leftarrow$ oldest period histogram in $\mathcal{W}$;
$\mathcal{X}^- \leftarrow \{x \mid E[x] > 0\}$;
$\mathcal{X}^+ \leftarrow \{x \mid B[x] > 0\}$;
$H_{\mathcal{W}} \leftarrow H_{\mathcal{W}} - E + B$;
$\mathcal{W} \leftarrow \textsc{AppendNewest}(\mathcal{W} \setminus E, B)$;

```
// 2. Recheck delayed demotions
```
for $i \leftarrow T-2, T-3, \ldots, 0$ **do**
  foreach *expired candidate* $x \in \mathcal{Q}_i$ **do**
    Dequeue x from $\mathcal{Q}_i$;
    if $\textsc{Locate}(x) = i+1 \wedge s_W(x) < \theta_i$ **then**
      Demote x from $\mathcal{A}_{i+1}$ to $\mathcal{A}_i$;
      $\textsc{MoveTier}(x, i+1, i)$;

```
// 3. Schedule newly cold hot tier keys
```
foreach $x \in \mathcal{X}^-$ **do**
  $j \leftarrow \textsc{Locate}(x)$;
  if $j > 0$ **then**
    $i \leftarrow j - 1$;
    if $s_W(x) < \theta_i$ **then**
      if x *is not scheduled in* $\mathcal{Q}_i$ **then**
        Enqueue x in $\mathcal{Q}_i$ for time $+\Delta_i$;

```
// 4. Promote newly active qualifying keys
```
foreach $x \in \mathcal{X}^+$ **do**
  $i \leftarrow \textsc{Locate}(x)$;
  if $i < T-1$ **then**
    if $s_W(x) \geq \theta_i$ **then**
      Promote x from $\mathcal{A}_i$ to $\mathcal{A}_{i+1}$;
      $\textsc{MoveTier}(x, i, i+1)$;

B Evaluation of Additional Policies

Section 6.2 focuses on the three HMT policies used in the main comparison: Ratio-Based, Sliding-Window, and Dynamic-Control. Here we report two additional fixed-policy configurations from the same one-million-block Ethereum replay. Both use the two-tier HMT design with a balanced MT for the cold tier, a periodic-rebuild HuffMHT as hot tier, fixed cache capacities of 8,000/16,000 entries, a cache frequency range of 10, and 500-block batches. Absolute Threshold promotes an element when its cumulative CMS estimate exceeds 50,000. Periodic Evaluation uses the interval-based policy, evaluates promotion candidates every 500 blocks, and promotes an element upon meeting a rate threshold of 0.05. Figures 10 and 11 show that these additional fixed policies preserve the main evaluation trend. Both Absolute Threshold and Periodic Evaluation remain below MPT and UBT in hashed input bytes, while their access-weighted proof sizes stay clustered with the other HMT variants.

C Weighted Average Proof Size By Read/Write

The proof-size metric used in Section 6.2 weights each key by the total number of both write and read accesses in the block. We now consider refinements of this metric which account for only one of the two: either reads or writes. We evaluate each refinement and present the results in Figs. 12 and 13, noting that both show the same performance ordering as the aggregate metric. MPT has the largest proofs across metrics, rising from roughly 2 kilobytes at the beginning of the replay to above 3 kilobytes later in the run. UBT lowers the read- and write-weighted averages to about 1 kilobyte, but remains well above the HMT variants. All HMT policies remain in the hundreds-of-bytes range for both reads and writes, with Sliding-Window generally at or near the lowest curve. The similarity between the two metrics indicates that the proof-size reduction is not primarily a consequence of the trace's read/write composition. Rather, the reduction persists for both operations because frequently accessed accounts are placed on shorter paths in the hot tier.

Algorithm 7: Dynamic-Control Controller

Input:

b: Current period index.

Parameters:

T: Number of tiers.
q_{high}: Rejection threshold.
K: Capacity-control interval.
$\theta_i^{\min}$: Lower bound for edge i.
M: Threshold-control interval.
$\theta_i^{\max}$: Tail-derived upper bound.
U: Low-occupancy streak length.
$\rho_{\text{low}}, \rho_{\text{high}}$: Target occupancy band.

State:

n_i: Hot tier size.
C_i: Hot tier capacity.
θ_i: Current threshold.
A_i, R_i: Attempts and rejects.
u_i: Low-occupancy streak.

```
// Current control-window index
```
if $b \bmod M \neq 0$ **then**
    return;

$w_{\text{ctrl}} \leftarrow b/M$;

```
// 1.  Threshold control
```
for $i \leftarrow 0$ **to** $T-2$ **do**
    $\rho_i \leftarrow n_i/C_i$;
    $q_i \leftarrow R_i/\max(A_i, 1)$;
    if $\rho_i > \rho_{\text{high}} \wedge q_i > q_{\text{high}}$ **then**
        Raise θ_i, capped by $\theta_i^{\max}$;
    Update u_i: increment if $\rho_i < \rho_{\text{low}}$, otherwise reset;
    if $u_i \geq U$ **then**
        Lower θ_i, floored by $\theta_i^{\min}$;
    Reset A_i and R_i;

if $w_{\text{ctrl}} \bmod K \neq 0$ **then**
    return;

```
// 2.  Capacity control
```
for $i \leftarrow 0$ **to** $T-2$ **do**
    $\rho_i \leftarrow n_i/C_i$;
    if $\rho_i > \rho_{\text{high}} \wedge \theta_i$ *is near* $\theta_i^{\max}$ **then**
        Grow C_i;
    if $\rho_i < \rho_{\text{low}} \wedge \theta_i$ *is near* $\theta_i^{\min}$ **then**
        Shrink C_i;

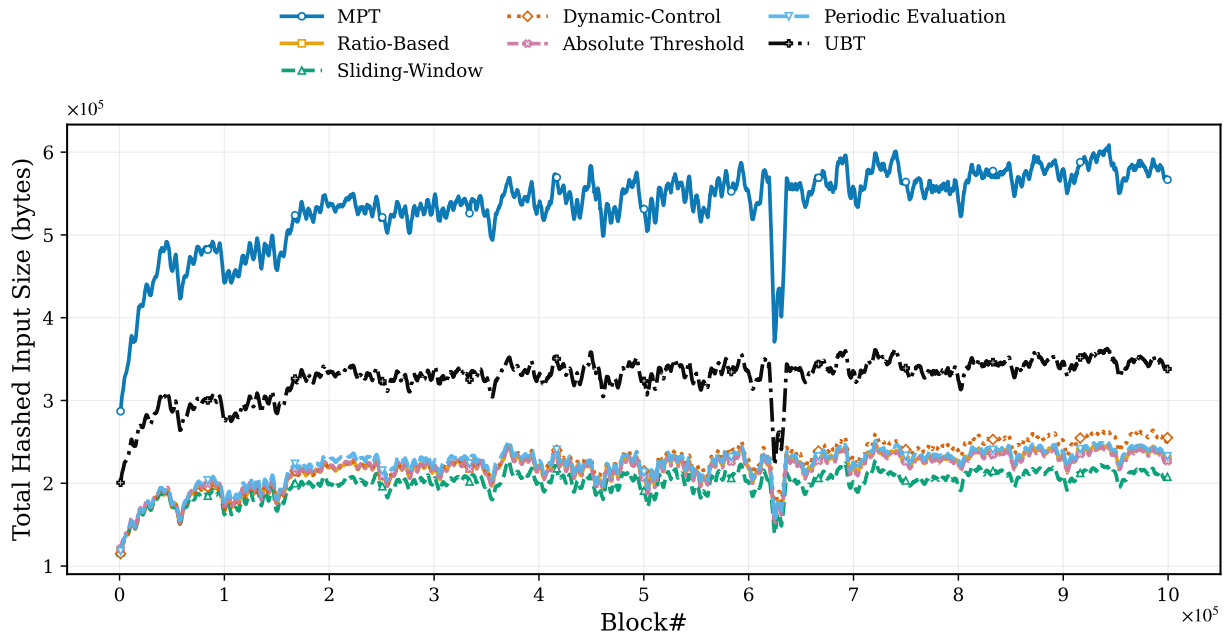


Figure 10: Per-block hashed input bytes for additional fixed-policy HMT configurations in the one-million-block Ethereum replay.

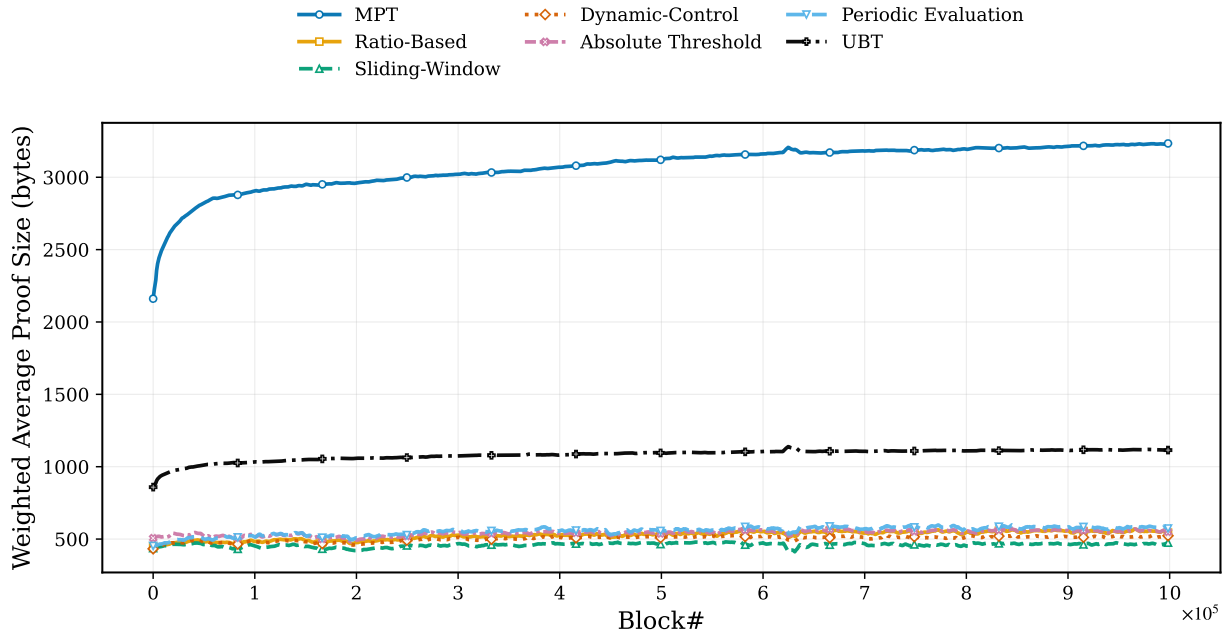


Figure 11: Access-weighted average proof size for additional fixed-policy HMT configurations in the one-million-block Ethereum replay.

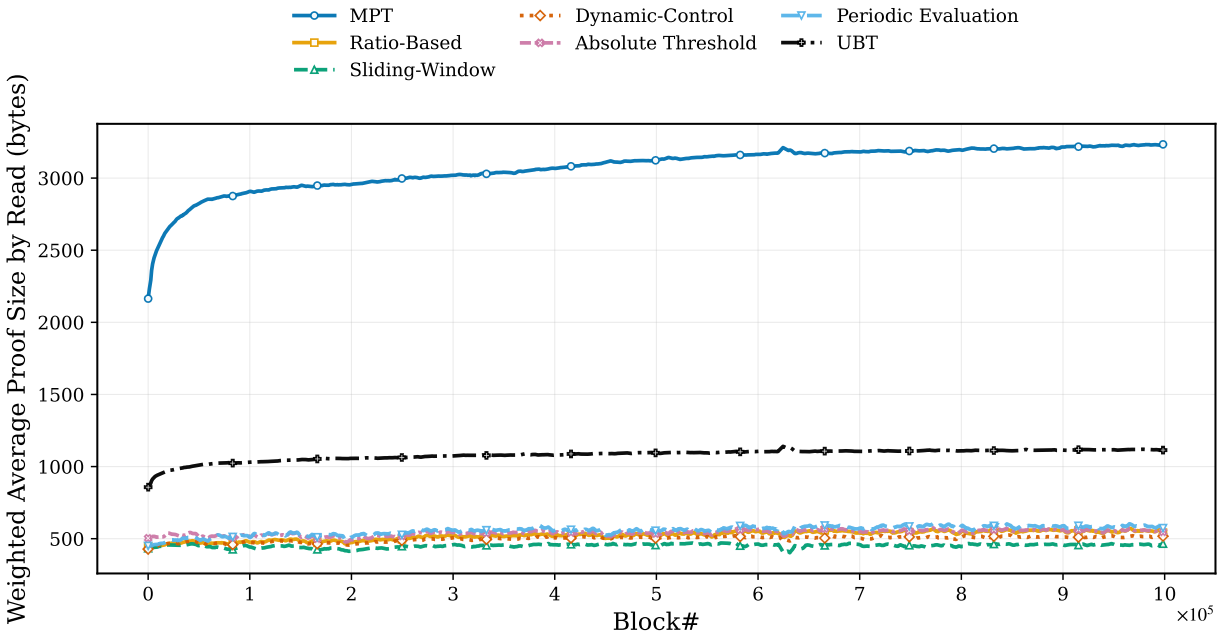


Figure 12: Read-weighted average proof size in the one-million-block Ethereum replay.

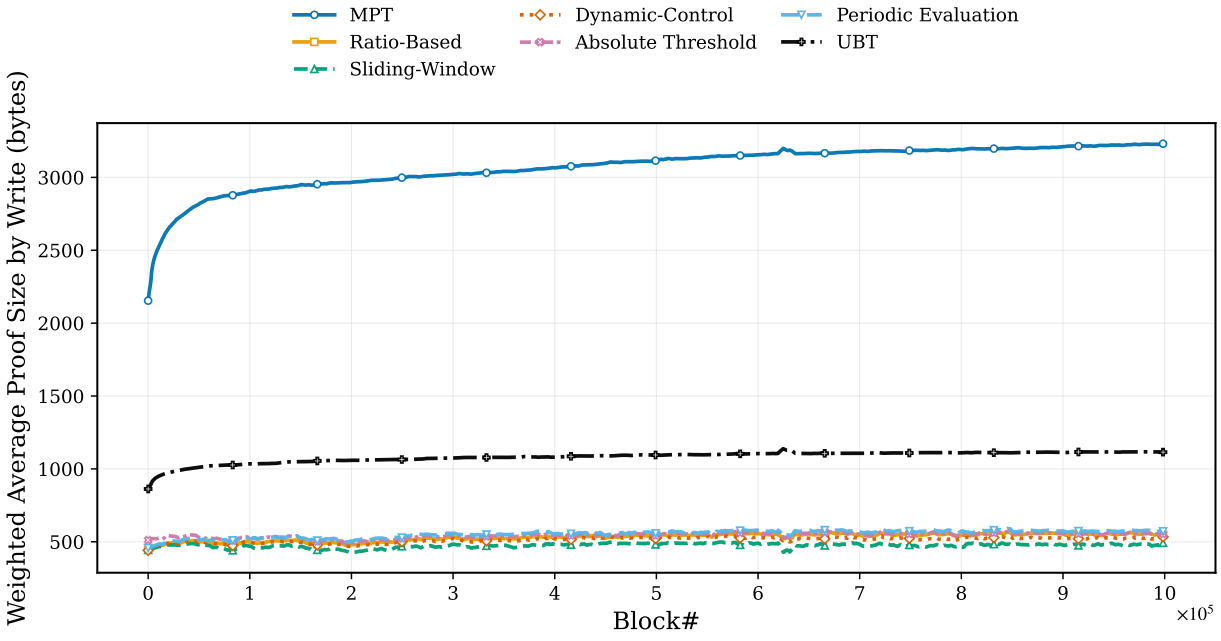


Figure 13: Write-weighted average proof size in the one-million-block Ethereum replay.