Fast Integrity Checking for Secure Cloud Storage: A Quantitative Approach

QUINN BURKE, University of Wisconsin-Madison, USA
RYAN SHEATSLEY, University of Wisconsin-Madison, USA
RACHEL KING, University of Wisconsin-Madison, USA
OWEN HINES, University of Wisconsin-Madison, USA
MICHAEL SWIFT, University of Wisconsin-Madison, USA
PATRICK MCDANIEL, University of Wisconsin-Madison, USA

Merkle hash trees are the standard method to protect the integrity and freshness of stored data. However, hash trees introduce additional compute and I/O costs on the I/O critical path, and prior efforts have not fully characterized these costs. In this paper, we quantify performance overheads of storage-level hash trees in realistic settings. We identify that hashing (CPU) costs are the primary performance bottleneck, and develop an analytical model demonstrating why this occurs. We then design an optimized tree structure called *Dynamic Merkle Trees (DMTs)* that exploits patterns in workloads to reduce tree traversal costs. We implement DMTs in a block device driver and a file system and through extensive evaluation show that DMTs can exploit patterns in workloads to deliver up to a 2.2× throughput and latency improvement over the state of the art. Our novel approach provides a promising new direction to achieve integrity guarantees in storage efficiently and at scale.

CCS Concepts: • **Security and privacy → Systems security**.

Additional Key Words and Phrases: Storage Security, Trusted Computing, Data Integrity

ACM Reference Format:
Quinn Burke, Ryan Sheatsley, Rachel King, Owen Hines, Michael Swift, and Patrick McDaniel. 2025. Fast Integrity Checking for Secure Cloud Storage: A Quantitative Approach. 1, 1 (May 2025), 36 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

An increasing number of attacks against cloud services has fueled significant investment and research into trusted cloud storage systems: systems that provide high assurance of the confidentiality and integrity of data stored in-memory and on-disk through hardware-based access-controls and cryptographic proof systems [3, 17, 51]. To this end, using a Merkle hash tree [44] has become the state-of-the-art method to protect the integrity and *freshness* of both volatile [28, 57] and

Authors' Contact Information: Quinn Burke, qkb@cs.wisc.edu, Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin, USA; Ryan Sheatsley, sheatsley@wisc.edu, Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin, USA; Rachel King, rachelking@cs.wisc.edu, Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin, USA; Owen Hines, mohines@wisc.edu, Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin, USA; Michael Swift, swift@cs.wisc.edu, Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin, USA; Patrick McDaniel, mcdaniel@cs.wisc.edu, Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin, USA.

ACM XXXX-XXXX/2025/5-ART
https://doi.org/XXXXXXX.XXXXXXX

persistent [3, 16] storage. An exemplar use case is protecting disks attached to confidential virtual machines or file systems mounted to confidential containers [10, 20, 53].

However, hash trees introduce additional compute (hashing) and I/O (metadata fetching) costs on the I/O critical path, which can severely degrade performance. For example, consider that a 1 TB disk contains ≈268 M 4 KB blocks. A typical balanced, binary hash tree over the disk blocks would have a height of 28, requiring (at least) 28 hashes to be computed on every read or write. The total cost of fetching metadata and verifying/updating hashes can exceed several hundred μs, dwarfing the baseline latency of performing a data access on a high-performance storage device (which can be $< 60\ \mu s$).

Prior works have studied this phenomenon, primarily in the context of secure volatile memory [28, 31, 57, 62]. However, their performance implications in the context of storage systems at large remain largely unknown. The key difference is that storage systems are subject to vastly different workload characteristics, capacities, and cache behaviors than memory devices.

In this paper[1], we take a first-principles approach to analyzing hash tree performance in the context of (cloud) storage systems[2]. First, we demonstrate that state-of-the-art hash tree designs incur significant overheads and fail to reliably scale to large disks or files. We then demonstrate that CPU costs are the primary performance bottleneck. Next, we address the challenge of minimizing CPU costs by posing the fundamental question: *How can we model an optimal hash tree that achieves minimum hashing costs for storage systems?* We show that the problem of finding an optimal hash tree can be reduced to the problem of finding an optimal prefix tree in the context of lossless data compression [34]. More specifically, by constructing a hash tree as an optimal prefix code, we can produce a hash tree that achieves optimal throughput under a known workload profile.

Building on our observations of optimal trees, we then develop an *online* solution that can approximate an optimal tree without a priori knowledge by learning and adapting to workload patterns on-the-fly. It is known that real-world workloads are characterized by skewed access patterns (i.e., where a small number of disk/file blocks are accessed much more frequently than others) across all layers of the memory hierarchy [3, 21, 40, 61, 63]. In an offline setting, this manifests as optimal hash trees often being far from balanced—where frequently accessed blocks have shorter verification/update paths in the tree than infrequently accessed blocks. Towards this, we introduce a novel dynamic, unbalanced hash tree design called *Dynamic Merkle Trees (DMTs)*. DMTs are based on the splay trees commonly used in garbage collection systems [56]. They self-adjust at runtime to exploit reference locality and reduce hashing costs.

We implemented DMTs in a block device driver via Linux device mapper framework and a file system via FUSE. In the block device driver implementation, the DMT protects the entire set of disk blocks, and in the file system implementation we use a per-file DMT. We performed an evaluation in a real cloud setting with AWS EC2 instances and NVMe devices. We evaluated across a range of system and workload settings, parameterized by disk capacity, read/write ratio, I/O size, etc. Using a set of Zipfian workloads, an Alibaba dataset, and a Filebench OLTP workload, we show that the static nature of state-of-the-art approaches becomes prohibitive: they deliver less than 50% of optimal throughput on average across all experiments. In contrast, DMTs can exploit reference locality to deliver >85% of optimal throughput and up to a 2.2× throughput and latency improvement over the state of the art.

The adaptability of DMTs enables them to exploit patterns in workloads and adapt to changes in workload patterns over time, in a principled and provably efficient way. This improves throughput,

[1]This article is an extension of our USENIX FAST 2025 paper [14]; it includes substantial new theory development, implementation, and experiments.
[2]Note that we focus primarily on block and file storage, but our insights extend broadly to other storage types.

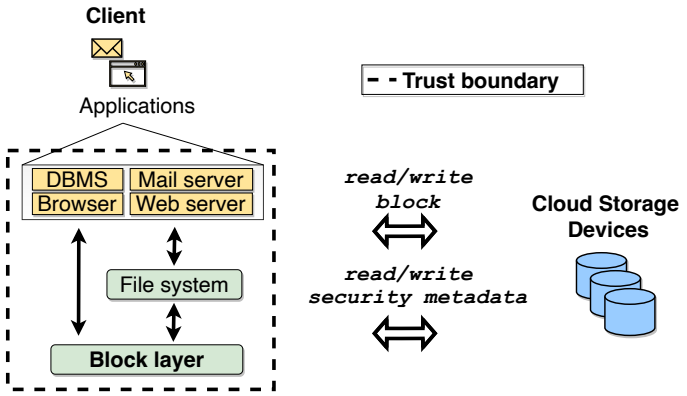


Fig. 1. We assume that code and data inside the VM (dashed line) are trusted and cloud storage interfaces are untrusted; VM memory can be protected with trusted execution primitives [53].

latency (mean and tail), and offers better scalability guarantees than balanced hash trees. We conclude that for cloud storage, balanced trees are ill-suited as the base construction of a hash tree, and DMTs are a preferable alternative. DMTs provide a new foundation in the search for integrity mechanisms that operate efficiently at scale. Our code is plug-and-play into Linux and open-sourced at https://github.com/MadSP-McDaniel/dmt.

2 Background

Cloud Storage Services. Cloud storage services are a backbone of modern public cloud infrastructure [7, 9, 19]. They have evolved from simple capacity provisioning to sophisticated, high-performance systems that must meet demanding requirements of modern applications. Cloud providers offer multiple storage tiers and interfaces, from high-IOPS NVMe-based block volumes capable of millions of operations per second to distributed file systems and object stores optimized for different access patterns and cost requirements. The architecture of cloud storage typically involves multiple layers of abstraction and virtualization. Physical storage devices (SSDs, NVMe drives) are aggregated into storage pools, which are then exposed through various interfaces: block storage volumes attached to VMs, distributed file systems accessible via network protocols, and object storage APIs for web-scale applications. This virtualization enables rich features like live migration, snapshotting, and replication, but also introduces additional attack surfaces that must be secured.

We consider a standard Infrastructure-as-a-Service (IaaS) deployment where an application runs inside of a guest VM and accesses storage through either fast, local block interfaces (e.g., NVMe) or file system interfaces (Figure 1). We will denote either type of storage as a *storage interface*. This model represents the most common and performance-critical deployment scenarios in modern cloud environments, where applications require both high throughput and low latency storage across both block and file access patterns. The application may be end-user facing (e.g., a web server) or the last hop in a networked storage system (e.g., a file server for other cloud-hosted applications).

Merkle Hash Trees. Merkle hash trees are the state-of-the-art method to protect the integrity and freshness of arbitrary datasets—largely due to their proven theoretical efficiency [17, 31, 42, 44, 50]. Originally proposed by Ralph Merkle in 1979 for digital signatures, Merkle trees have found

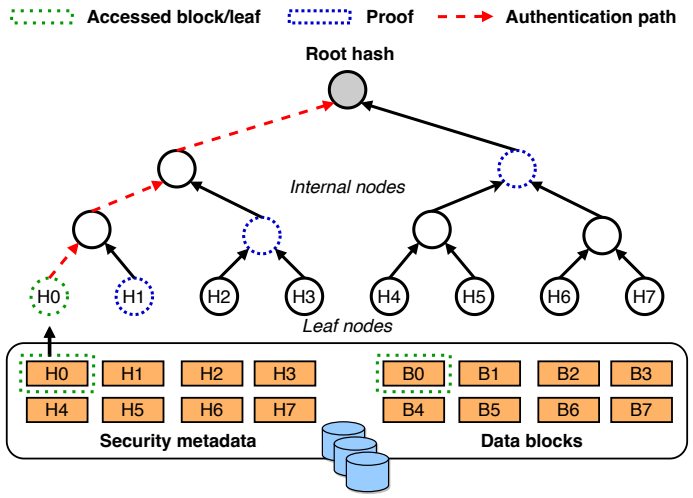


Fig. 2. A Merkle hash tree protects the integrity and freshness of data read from/written to a storage interface.

widespread application in distributed systems, blockchain technologies, and secure storage systems. Their appeal lies in their ability to provide efficient integrity checking with logarithmic verification time and the property that any change to the protected data results in a different root hash, making tampering immediately detectable.

As shown in Figure 2, a Merkle hash tree (or more simply, a hash tree) is typically a balanced binary tree, with each node in the tree containing a hash value. A leaf node contains the hash (MAC) of a disk/file block (and a cipher IV when encrypting data), and an internal node contains the hash of the concatenation of the hashes of its two children. Internal node hashes are iteratively computed from leaf to root. The root hash is used to *authenticate*, or verify the legitimacy of, the current contents of the storage interface. It is typically stored in a secure location (e.g., a persistent on-chip register or a TPM [49, 57]). All other nodes in the tree are stored on disk alongside the data. The number of leaf nodes in the tree n is equal to the number of blocks on the storage interface, and the total number of tree nodes is $2n - 1$.

There are two primitive operations on a hash tree: *verification* and *update*. When a block is read, it is verified against the root hash. The client's block layer first fetches the (encrypted) block data, MAC, and cipher IV from disk. It checks that the retrieved MAC is consistent with the retrieved block data by rehashing the data and comparing. It then fetches the proof of authenticity, a set of sibling hashes along the path from the accessed leaf to the root (see nodes highlighted in blue). The retrieved MAC is inserted into the tree at the appropriate leaf position, and parent hashes are iteratively computed along the *authentication path* using the sibling hashes (see red arrows). The computed root hash is then compared against the known root hash. If the two hashes match, verification succeeds. Updates are handled similar to verification, with a new root hash computed and saved to the secure location.

Caching hashes in secure memory (i.e., in a protected memory region) is also a standard hash tree optimization [3, 31]. Caching reduces I/O costs associated with verifying hashes. It also reduces CPU costs associated with verifying hashes: verification can "early exit" once it encounters a cached hash. If the computed hash matches the cached entry, the validity of the subtree is confirmed without needing to traverse the remaining path to the root; if it mismatches, corruption is detected immediately. Note that updates require traversing the full path to the root to commit a change.

Unlike simple checksums or MACs (which only prevent data corruption), Merkle trees provide a freshness guarantee (which prevents an attacker from replaying legitimate uncorrupted data). Moreover, compared to other authenticated data structures like RSA accumulators, polynomial commitments, or vector commitments, Merkle trees offer better performance characteristics for large datasets, with verification times typically in the microsecond range rather than milliseconds or seconds required by cryptographically heavier approaches [23]. In the context of storage, they have played a pivotal role in ensuring boot disk integrity with Linux *dm-verity* [1]—where they are implemented as a custom device driver that intercepts I/Os and implements the hash tree logic.

3 Security Model

As organizations increasingly migrate sensitive workloads to public cloud infrastructure, the assumption that the cloud provider's infrastructure is trusted has become untenable [5, 17]. Persistent data often sits far away from the compute (CPU) across several layers of abstraction and is prone to attackers trying to tamper with the data before it arrives at applications. These *data-only* attacks—attacks based on maliciously crafted data rather than control flow hijacking—have been recently shown to present a significant threat to modern applications [12, 30, 33, 35, 37]. Unlike traditional memory corruption attacks that aim to hijack control flow, data-only attacks manipulate application behavior by corrupting or replacing non-control data. Any attack that can be launched from (untrusted) storage would fundamentally upend the security guarantees provided by the rest of the system. Below we describe these attacks and outline security requirements to mitigate them.

Trust Model. We assume all VM contents (code and data) are trusted and the storage interfaces are untrusted. VM memory contents can be protected with hardware-based isolation primitives such as AMD SEV-SNP [53]. We trust only the CPU package and VM memory when protected by hardware isolation; all other components (storage interfaces, controllers, network infrastructure, hypervisor) are untrusted. The trusted and untrusted components have a simple block read/write interface (Figure 1). This models untrusted disks attached to confidential VMs as raw block devices or as formatted file systems (which similarly exit the VM through the block interface) [10, 20, 53].

Threat Model. We consider a privileged attacker who has access to the hypervisor or storage backbone in a public cloud datacenter [5, 17]. This could be a malicious co-tenant with escalated privilege, or a malicious cloud administrator. The attacker has comprehensive control over the storage subsystem with the ability to read, modify, drop, or replay storage operations, but cannot break cryptographic primitives or compromise the hardware TEE directly.

Example attacks. Armed with the capability to inject arbitrary data into the storage interface, the attacker could replay old data to the VM [35, 37]. Data would bubble up the call stack and either cause the VM to deliver old data to applications, or cause an outdated version of a binary to be read from disk and executed [12, 30, 35]. Similarly, consider an ext4 file system formatted on top of the disk. An attacker could arbitrarily replay inode table blocks and cause the VM guest OS to recognize an invalid set of permissions on a file, enabling unauthorized access to the file. More sophisticated attacks might involve replaying database transaction logs or manipulating configuration files to alter application behavior. Checksums or keyed hashes alone cannot prevent these data-only attacks: the received data would still pass verification.

Security Requirements. Ensuring the safety of user data and correct execution of applications requires four security properties for storage: *Confidentiality* ensures that stored data cannot be read by unauthorized parties. *Authenticity* (or integrity) ensures that any unauthorized modifications to stored data can be detected. *Uniqueness* (or spatial integrity) ensures that data cannot be copied

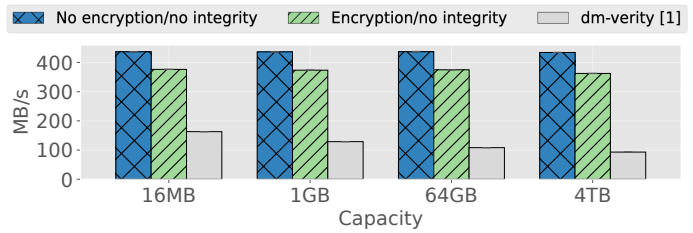


Fig. 3. This graph shows how throughput decreases w.r.t. capacity under an exemplar setup and workload. Experiment parameters: Workload: Zipf(2.5), Read ratio: 1%, I/O size: 32 KB, Cache size: 10%.

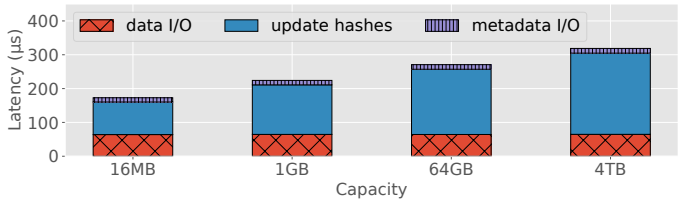


Fig. 4. CPU vs. I/O time during the driver `write` routine. Same experiment parameters as above.

from one location to another without detection. *Freshness* (or temporal integrity) ensures that old versions of data cannot be replayed.

Keyed encryption and MACs can ensure confidentiality, authenticity, and uniqueness. Merkle hash trees ensure data freshness [2, 3, 31, 41]: the root hash reflects the current *version* of the storage, so a replay attack would require changing the root hash, which is stored in a secure location and is out of control of the attacker[3].

4 Motivation

Though hash trees have played a pivotal role in ensuring boot disk integrity with Linux *dm-verity* [1], their performance implications in the context of cloud storage at large, and low-latency storage in particular, are largely unknown. In fact, prior works have identified that hash tree overheads can severely degrade performance for secure memory systems, and optimizations abound [57]. This raises the natural question of whether storage-level hash trees observe similar costs. Our goal in this paper is to quantify this effect and design optimizations to reduce overheads if so.

Scalability Problem. We begin with a motivating experiment in Figure 3, which demonstrates the performance of the state-of-the-art hash tree design used by dm-verity—a balanced, binary tree. The graph shows how throughput changes as disk capacity increases. We defer implementation and experiment setup details to Section 8.1, but note that in this example the hash tree is implemented in a block device driver that wraps a lower-level driver, and is exposed as a regular device to file systems or other applications as `/dev/XXX`. Moreover, we focus specifically on fast media such as NVMe SSDs. HDDs have a different performance profile that we are not optimizing for.

The graph shows that throughput decreases w.r.t. capacity. This is due to the tree size (height) increasing logarithmically with capacity, which is reflected in logarithmically increasing slowdowns. At 16 MB capacity, the hash tree incurs nearly a 60% throughput loss over the Encryption/no

[3]To improve performance, some prior works have loosened security requirements by permitting lazy verification [3]. However, this violates freshness guarantees; we therefore do not consider lazy verification in our analysis.

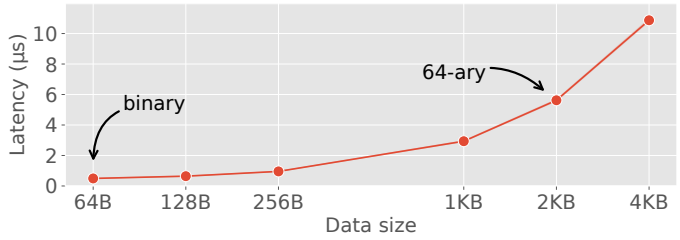


Fig. 5. This graph shows the latency of computing SHA256 hashes on a modern processor with hardware acceleration for cryptographic functions. The annotations highlight the input data size to the hash function at different tree arities.

integrity baseline. At 4 TB capacity, throughput loss increases to 75%. Note that the workload shape is immaterial here; the same overheads are always observed because of the tree structure (i.e., all accesses will require computing the same number of hashes).

Read-heavy workloads do not pose significant challenges, because the (small) hash cache is very efficient (hit rate >99%), and verifies benefit from early exits when they hit a cached hash. The problem is how to efficiently handle writes: under write-heavy workloads, hash tree overheads are prohibitive at small and large capacities, undermining the performance capability of the fast NVMe device[4].

Root Cause Analysis. Figure 4 shows the latency breakdown during the device driver `write` routine. As expected, for a 32 KB I/O the time spent pushing data out to disk (data I/O) is approximately 60 μs. The remaining time in the write routine is spent fetching/writing hashes to disk (metadata I/O) and performing hash updates (computing the new block hash and executing the hash tree update). Metadata I/O is negligible because the hash cache is very efficient. The majority of time is therefore attributed to managing the hash tree.

To understand why this occurs, Figure 5 shows the latency to compute a SHA256 hash (the standard hash function used in Merkle hash trees) vs. data size on a 2.9GHz Intel Xeon Platinum 8375C, a 3rd Generation Intel Xeon Scalable processor supporting AES and SHA instruction-set extensions to accelerate cryptographic operations. We observe that it takes approximately 490 ns to compute the hash of 64 B of data. We also measure the latency to encrypt and generate the MAC for a 4 KB block with AES GCM to be approximately 2 μs.

Figure 4 shows that at 1 GB capacity, approximately 150 μs is spent managing the hash tree. Consider that a 1 GB disk has 262,144 4 KB blocks and thus a height of 18, requiring one SHA256 computation per level. Further, with 4 KB disk blocks, executing a 32 KB write I/O would require $32768/4096 = 8$ hash tree updates executed sequentially—best-known methods still rely on a global tree lock to serialize tree updates. This amounts to $150\ \mu s/8 = 18.75\ \mu s$ spent encrypting data, generating the MAC, and updating the hash tree. Thus, we have $18.75 - 2 = 16.75\ \mu s$ time spent doing the actual hash tree update, and $16.75/18 = 0.93\ \mu s$ total time spent doing work at each level in the tree. Most of this time is spent computing the node hash, with the remaining work being cache lookups and buffer copying.

[4]Note that in some cases the page cache can hide disk-level performance impacts, but the performance of large-scale applications will depend heavily on the underlying disk performance (due to writeback contention and throttling under heavy memory pressure [22]). Our focus is thus on analyzing and optimizing the storage layer to ensure that the performance capabilities of fast NVMe devices are not undermined by hash tree operations.

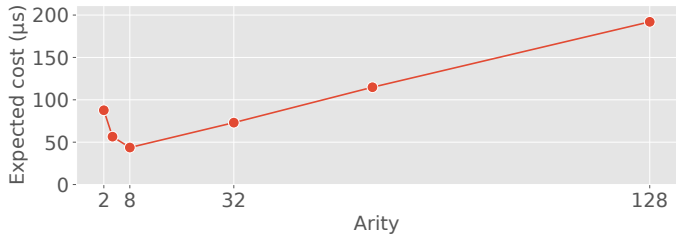


Fig. 6. We calculate the expected hashing costs for a 32 KB write I/O based on the tree height under different tree arities, given the measured SHA256 latencies for each arity in Figure 5. The graph shows that low-degree trees should have lower hashing costs than high-degree trees.

Note that with even faster storage interfaces in the future (with single-digit microsecond access latencies), the proportion of time spent hashing vs. doing data I/O will grow substantially.

Optimized Tree Structures. Fundamentally, this means that time spent hashing (*CPU costs*) is the bottleneck. Section 8 will show that caching and parallelization only help to an extent. What is needed is a structurally more efficient tree. Prior works optimizing hash trees for memory have largely converged on the idea that high-degree (e.g., 64-ary) trees are the solution to eliminate overheads [57]. The intuition is that by increasing tree fanout, one can decrease tree height and thus the number of hashes that must be computed per read or write. However, Figure 6 shows that high-degree trees are actually a suboptimal design choice. We compute the expected hashing costs based on the hashing latency and the height of the tree observed under a given arity for 1 GB capacity (e.g., 64-ary trees have height 3). While tree height is decreased, the graph shows that increased fanout results in high-degree trees incurring the highest expected hashing costs due to the non-linear growth of SHA256 latency with input size, ultimately delivering lower performance.

This counterintuitive result stems from the non-linear scaling behavior of SHA256 hardware implementations. As shown in Figure 5, hash latency does not scale linearly with input size—instead, it exhibits super-linear growth due to: (1) *cache effects*—larger hash inputs exceed CPU cache lines and cause cache misses; (2) *memory bandwidth saturation*—processing large inputs saturates memory bandwidth; (3) *reduced instruction-level parallelism*—cryptographic units become less efficient with larger inputs; (4) *context switching overhead*—larger operations increase scheduling and interrupt latency.

To quantify this trade-off, consider the total hashing overhead per tree update. For a tree with n data blocks and degree d, the tree height is $h = \lceil \log_d(n) \rceil$, and each update requires computing h hash operations with input sizes of $d \times 32$ bytes each (d child nodes multiplied by the 32-byte SHA256 hash). The total computational latency per update is:

$$L(d) = \lceil \log_d(n) \rceil \times \text{SHA256\_latency}(d \times 32) = \frac{\log n}{\log d} \times \text{SHA256\_latency}(d \times 32)$$

The key insight is that (for any hardware configuration) there exists a unique critical degree d^* that represents a fundamental saddle point in this optimization space. For degrees below d^*, performance is limited by tree height (requiring too many hash computations per tree traversal). For degrees above d^*, performance is limited by hash efficiency degradation (each individual hash operation becomes too expensive due to non-linear scaling). The optimal degree d^* balances these competing factors.

Formally, we can prove that any degree $d > d^*$ is suboptimal. Assuming there exists a critical degree d^* where hash efficiency begins to degrade significantly, we can show that any higher degree must perform strictly worse. If hash efficiency degrades for $d > d^*$ such that $\frac{\text{SHA256\_latency}(d\times 32)}{d\times 32} > \frac{\text{SHA256\_latency}(d^*\times 32)}{d^*\times 32}$, then for sufficiently large n:

$$L(d) = \frac{\log n}{\log d} \times \text{SHA256\_latency}(d \times 32) > \frac{\log n}{\log d^*} \times \text{SHA256\_latency}(d^* \times 32) = L(d^*)$$

This inequality holds because increasing degree from d^* to d reduces tree height by only a logarithmic factor of $\frac{\log d^*}{\log d}$, while the hash latency increases by a factor greater than $\frac{d}{d^*}$ due to the non-linear efficiency degradation shown in Figure 5. Since logarithmic improvements are always dominated by super-linear penalties for large datasets, the latency penalty inevitably outweighs the height benefit. This provides a clear stopping criterion: once we empirically identify the saddle point where efficiency degrades, any higher degree must be strictly suboptimal.

Figure 6 validates this theoretical prediction empirically. For modern processors with SHA extensions, the critical degree occurs at $d^* = 8$ when hash inputs exceed roughly 256 bytes. At this point, 8-ary trees achieve 45 μs (the saddle point), while 64-ary trees require 110 μs, and higher degrees perform worse. This analysis reveals that the conventional wisdom of using high-degree trees overlooks the performance characteristics of modern cryptographic hardware.

Exploiting reference locality in storage workloads. Given that low-degree trees deliver better performance, the key challenge then becomes how to best optimize and maneuver low-degree tree structures. Examining real-world storage workload characteristics offers a significant opportunity for optimization. Storage access patterns in cloud environments exhibit several important properties that traditional balanced hash trees fail to exploit: (1) *Temporal locality*—recently accessed data is likely to be accessed again soon; (2) *Spatial locality*—data located near recently accessed data is likely to be accessed soon; (3) *Skewed distributions*—a small fraction of data accounts for a large fraction of accesses, often following power-law distributions such as Zipfian distributions; (4) *Write dominance*—many cloud storage workloads are write-heavy due to logging, caching, and data processing pipelines.

These characteristics suggest a fundamental insight: *there is an opportunity to reduce hashing costs by exploiting reference locality*. Instead of treating all data uniformly as balanced trees do, we can break away from the balanced tree structure, allowing the tree to become unbalanced while keeping frequently accessed (hotter) data closer to the root. The result should be lower hashing costs for hot data and overall better performance on average. However, to capture this insight effectively, we need both (1) a principled way to determine what constitutes an optimal unbalanced tree structure, and (2) a data structure that can learn and adapt to access patterns on-the-fly, without sacrificing security guarantees or introducing excessive overhead in tree management.

The following sections address these two challenges: Section 5 establishes a theoretical foundation for optimal hash trees, while Section 6 develops a practical adaptive approach that addresses some limitations of the optimal definition. We note that in the following, we focus primarily on binary trees rather than 8-ary trees, which enables simpler theoretical analysis and design. Moreover, while indeed 8-ary trees perform best among balanced trees, we will show in Section 8 that they are outperformed by our optimal binary tree construction. For these reasons, we leave future work to extend our optimal definition and adaptive tree design to other (low-degree) k-ary trees.

5 Optimal Hash Trees

We approach this problem by asking the fundamental question: *Is there an optimal tree structure?*. Having a definition of an optimal tree serves two purposes: (1) under a specified set of assumptions,

it establishes an upper bound on performance, and (2) it discloses what characteristics of the tree structure are correlated with optimal performance.

5.1 Optimal Definition

We previously showed that CPU costs are the bottleneck that affect hash tree performance. Intuitively, an optimal hash tree must therefore be a tree that reduces the number of hashes that must be computed per update or verification, reducing hashing costs and therein improving performance.

We observe that the problem of finding an optimal hash tree can be reduced to finding an optimal prefix tree (or *prefix code*) in the context of lossless data compression [46]. Prefix codes map a set of symbols onto a set of codewords, with the goal of compression being that codewords are as short as possible to produce a maximally compressed representation of the original data. An example is shown in Figure 7. Formally:

Theorem 1. *A hash tree constructed as an optimal prefix code is optimal for an i.i.d. access probability distribution.*

Proof. Let $A = \{a_1, a_2, \ldots, a_n\}$ be a symbol alphabet and $W = \{w_1, w_2, \ldots, w_n\}$ be a set of associated symbol weights. Let $C = \{c_1, c_2, \ldots, c_n\}$ be a prefix code that represents the set of codewords for symbols in A. A prefix code C is said to be optimal if it minimizes the *expected* codeword length: $\arg\min_C \sum_{i=1}^{n} w_i|c_i|, c_i \in C$. The length of a codeword is the number of bits in a codeword, i.e., the *number of edges* in the path from the root to the symbol leaf in the prefix tree representation of C. Huffman coding is a widely-used algorithm to produce optimal prefix codes [34].

Now let $B = \{b_1, b_2, \ldots, b_n\}$ be a set of disk blocks and $F = \{f_1, f_2, \ldots, f_n\}$ be a set of access frequencies to blocks determined by some known workload profile. Suppose we map each block b_i to a symbol a_i and each access frequency f_i to a symbol weight w_i. Running Huffman's algorithm on A and W produces a prefix code with expected codeword length $\sum_{i=1}^{n} w_i|c_i| = \sum_{i=1}^{n} f_i|b_i|$.

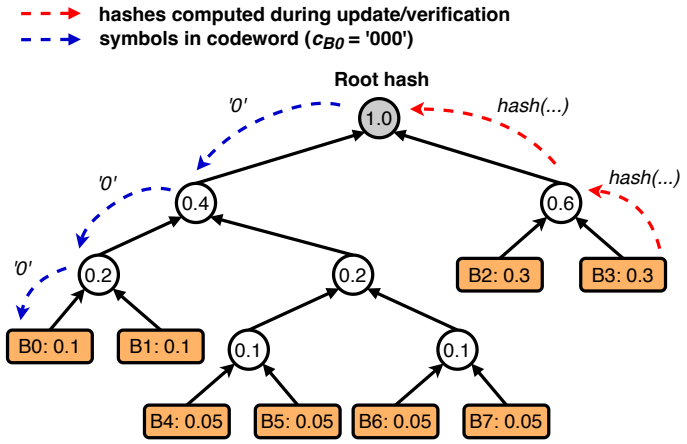
In the compression domain, the number of edges represents the number of bits needed to parse a symbol's codeword, while in the hash tree domain it represents the *number of hashes* that must be computed from leaf to root for a block. A hash tree constructed as a Huffman code minimizes the expected number of hashes computed during an update or verification and is therefore an optimal hash tree. □

5.2 Extended Optimal Definition

Now we extend our optimal definition to consider the effects of cache performance. Note that both data blocks and hashes can be cached in memory. In the compression domain, codeword paths in an (optimal) prefix tree can be parsed in constant work per edge, giving a total amount of work: $\sum_{i=1}^{n} w_i(|c_i| \cdot O(1)) = O(1) \cdot \sum_{i=1}^{n} w_i|c_i|$. However computing a hash requires (at least) two hash fetches in a binary hash tree: the node's two children. If both hashes are present in memory, fetch costs are negligible and the amount of work is similarly optimal: $\sum_{i=1}^{n} f_i(|b_i| \cdot O(1)) = O(1) \cdot \sum_{i=1}^{n} f_i|b_i|$. Otherwise if they must be fetched from disk, I/O costs are non-negligible: $\sum_{i=1}^{n} f_i(|b_i| \cdot t(b_i)) = \sum_{i=1}^{n} f_i|b_i| \cdot t(b_i)$, for some function $t(b_i)$.

We can model the incurred I/O costs using the average memory access time formula:

$$\begin{aligned} &\text{AMAT} = \text{hit time} + \text{miss rate} \times \text{miss penalty} \\ \implies\ &t(b_i) = \text{mem latency} + \text{miss rate} \times \text{reauth latency} \\ \implies\ &t(b_i) = H + mD = O(1) + mD \end{aligned} \tag{1}$$



Access probabilities

Symbol/Block	*B0*	*B1*	*B2*	*B3*	*B4*	*B5*	*B6*	*B7*
Probability	*0.1*	*0.1*	*0.3*	*0.3*	*0.05*	*0.05*	*0.05*	*0.05*

Fig. 7. A Huffman tree is an optimal prefix tree. A hash tree constructed as a Huffman tree with a given access probability distribution is an optimal hash tree.

where H is a fixed memory access cost, m is the miss rate of a node fetch in memory, and D is a fixed fetch/reauthentication cost. Substituting this in, the total amount of work is:

$$\begin{aligned}\sum_{i=1}^{n} f_i|b_i| \cdot t(b_i) &= \sum_{i=1}^{n} f_i|b_i| \cdot (O(1) + mD) \\ &= \underbrace{O(1) \cdot \sum_{i=1}^{n} f_i|b_i|}_{\text{base work}} + \underbrace{mD \cdot \sum_{i=1}^{n} f_i|b_i|}_{\text{I/O costs}}.\end{aligned} \tag{2}$$

Remark. From our model, we see that higher miss rates for block hashes incur more work per edge, proportional to the expected number of hashes that must be computed per update or verification. Specifically, at a given miss rate, the incurred I/O costs follow the same distribution as the underlying access probability distribution: hotter data has a lower expected amount of base work and incurs lower I/O costs, while colder data has a higher expected amount and incurs higher I/O costs.

We also see that with an optimal cache ($m = 0.0$), the expected total amount of work is exactly optimal. However, it has been empirically observed that as cache size decreases, miss rates increase with a power law [18], and thus as the cache size decreases, expected I/O costs increase with a power law. This implies that the performance of hash trees is very sensitive to cache size. In particular, as cache memory can be financially costly on cloud servers, being able to synergize well with relatively smaller caches (w.r.t. larger disks) is critical to a practical hash tree deployment.

Moreover, a Huffman tree is optimal under a *known* and *fixed* set of weights (cf. access probability distribution) for an *i.i.d.* source. If the symbol sequence (cf. block access sequence) observed while compressing a message (cf. during a program trace) exactly matches the one used to construct the tree, then the tree will be exactly optimal (i.e., provide optimal throughput). However, if the sequence deviates from the one used to construct the tree, the tree will not be exactly optimal.

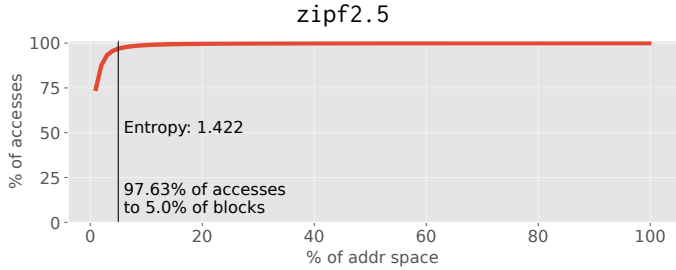


Fig. 8. Zipfian distributions are commonly used to model the shape of real-world storage workloads, which are often skewed [40, 63]. This graph shows that a small number of blocks is accessed most of the time and suggests that operations on the hash tree should also be skewed.

Similarly, if the source is not *i.i.d.*, then the tree will not be exactly optimal—temporal patterns in the workload may cause the tree to underestimate an upper bound.

5.3 Optimal Tree Oracle

Our optimal definition provides that, if we have knowledge of a concrete block access sequence (i.e., workload trace), we can instantiate an optimal hash tree from the trace and measure a concrete upper bound on performance (i.e., maximum possible throughput under the given workload). In an *offline* setting, where we have access to workload traces (e.g., recorded with tools like *blktrace* or *fio*), we can feasibly do so. We refer to this methodology as the optimal tree oracle.

The primary purpose is to measure whether overheads observed by a hash tree design (like dm-verity) are better attributed to the structure of the tree or to a fundamental scaling limit. For example, a hash tree may be performing optimally, but have high overheads, which would suggest that complimentary optimizations (e.g., dividing the tree into one or more independent security domains) may be the only way to break the performance ceiling. In contrast, a hash tree that does not perform optimally under a given workload may require a fundamental redesign.

We liken this approach to Belady's optimal page replacement algorithm [11], a clairvoyant algorithm that has a priori knowledge of future memory accesses and can make optimal page replacement decisions. This gives us the ability to make rigorously grounded conclusions about what hash tree designs perform well and when. We defer analysis with the optimal tree oracle (denoted by H-OPT) to Section 8.

6 Dynamic Merkle Trees

A condition of instantiating an optimal hash tree is that we must have a priori knowledge of the workload. This is rarely feasible in practice. Building on our insight that exploiting reference locality can reduce hashing costs, this section develops a novel hash tree design that can approximate an optimal tree by learning and adapting to workload patterns on-the-fly, without sacrificing security guarantees or introducing excessive tree management overhead.

6.1 Challenges

Finding a Suitable Tree Structure. State-of-the-art hash tree designs rely on static, balanced tree structures [1, 57]. Balanced trees are optimal under uniform access patterns. However, real-world storage workloads most often exhibit skewed (i.e., non-uniform) access patterns [3, 21, 40, 61, 63]

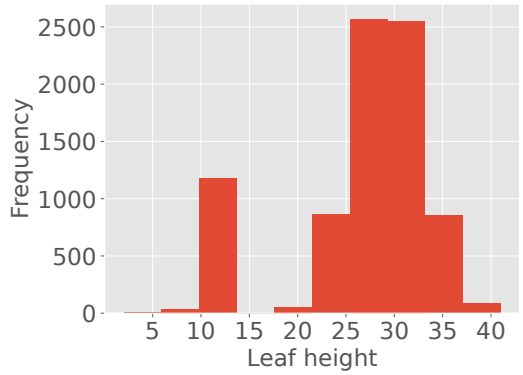


Fig. 9. Skewed patterns manifest in optimal hash trees being far from balanced: under a Zipfian probability distribution, there are distinct regions of relatively hotter and colder data.

where a small number of blocks is accessed most of the time. The result is that the optimal hash trees produced by Huffman codes are often far from balanced.

For example, Figure 8 shows the access distribution for a Zipfian workload; note that most real-world workloads obey Zipf's law and are modeled with Zipfian workloads [63]. The data accesses are highly skewed, which suggests that the operations on the hash tree may also be highly skewed. Figure 9 shows this to be true: in a balanced tree over 8192 blocks (a 32 MB disk/file), leaf node heights are constant at 13, but in the optimal tree we see two distinct regions representing hotter (height $\approx$ 10) and colder data (height $\approx$ 30).

Optimal trees tend to accumulate hot data high up in the tree and place cold data at nearly a 3$\times$ height difference—significantly reducing the verify/update latency for hot data. This indicates that an optimal tree is one that aggressively optimizes for hot data (the working set). Unfortunately, the *static* nature of standard hash tree designs precludes exploiting this skew when present in a workload.

Handling Changing Access Patterns. Yet, workload characteristics can also vary over time: access patterns may still be skewed but regions of interest may change, or some periods of time may be characterized by more uniform access patterns. This is particularly true for storage that is shared by multiple cooperating applications or users. Thus, a tree that is optimal at one point in time may not be optimal for another (i.e., dynamically optimal). An online solution, one that does not assume a priori knowledge of workload characteristics, therefore must not only be able to *capture* hot data by placing more frequently accessed nodes higher in the tree, but also be able to dynamically *adapt* to changes in what particular data is deemed hot or cold over time.

Adaptive tree structures have been widely studied, particularly for search. Most algorithms focus on keeping trees balanced to reduce worst-case running time. We explicitly aim to remove this constraint; commonly used self-balancing trees (e.g., AVL trees) are therefore ill-fit for our use case. We aim for a more aggressive optimization: allow the tree to become unbalanced as necessary, but be driven by the workload.

Managing Restructuring Costs. While intuitively it makes sense that unbalanced trees should be able to exploit skewed patterns by placing more frequently accessed leaf hashes closer to the root, realizing this is in a real system is a non-trivial problem. The mechanics of adapting trees involve a series of rotations. While rotations are cheap for search trees, consisting of a series of pointer

updates, they are expensive for hash trees, as we have to recompute hashes for all nodes from the rotation point up to the root. This applies when nodes are rotated during either verifications or updates.

The costs of rotating nodes in the tree may therefore quickly outweigh any expected benefits of moving frequent nodes closer to the root. The cost of a rotation itself is also not constant, but proportional to the current height of the nodes involved in the rotation. Further, search trees permit all nodes to be searchable, but as mentioned, only leaf nodes are searchable in a hash tree. We therefore must maintain the invariant that during a rotation, a leaf remains a leaf and an internal node remains an internal node. Otherwise, a rotation will result in an invalid tree structure.

6.2 Randomized Splaying

We draw a connection to the splay tree data structure, which is widely used in garbage collection and IP routing systems [56]. We develop a variant of splay trees called *Dynamic Merkle Trees (DMTs)*. Splay trees are a type of binary search tree that brings an accessed leaf to the root through a series of rotations. Importantly, splay trees capture temporal locality by keeping frequently accessed nodes closer to the root (Figure 10). Splay trees provide several key theoretical guarantees that make them attractive for hash tree adaptation: (1) *Static optimality*, which guarantees performance within a constant factor of any fixed tree structure; (2) *Working set property*, which guarantees efficient performance for workloads with temporal locality; and (3) *Amortized efficiency*, which guarantees that while individual operations may be expensive, average performance remains bounded. These properties ensure that our Dynamic Merkle Trees (DMTs) can maintain predictable performance while adapting to workload patterns.

However, naively used, splaying costs can be expensive, and splaying too frequently or opportunistically may keep the tree more balanced than desired. We adapt the conventional splay tree design to meet the constraints of a hash tree.

Heuristic Parameters. We define three parameters: a splay window flag w, splay probability p, and splay distance d. The splay window flag can be toggled on or off to indicate whether or not the splay window is active (i.e., whether or not we should consider a tree node to be splayed). This notion is useful because there may be certain periods at runtime where splaying should necessarily not occur. This may be the case, for example, if the system administrator has knowledge of current application access patterns or profiles them periodically, or if other background storage tasks may be in progress that require stability of data (e.g., health checks).

If the splay window is active, the splay probability denotes the probability that an accessed node should be splayed. The key intuition is that splaying is an expensive operation, but we can amortize costs by only splaying on a small percentage of accesses (e.g., 1% of the time). Finally, if a node is decided to be splayed, the splay distance defines the maximum number of levels that the node should be splayed (i.e., *promoted*) up the tree. This entire process occurs at the end of each verification or update call and before anything is returned to the caller.

6.3 Technical Approach

Analyzing Data Hotness. The splay distance is the central parameter that determines the effectiveness of a splay operation. Determining a suitable splay distance is challenging, as there is an inherent risk vs. reward trade-off when splaying a node. Splaying any node all the way to the root may be very beneficial if a node is relatively hot, as future accesses to the data can quickly benefit from the promotion. However, doing so would be severely wasteful if the node is cold, as the tree will then require several *additional* rotations to eventually promote hotter data and demote cold

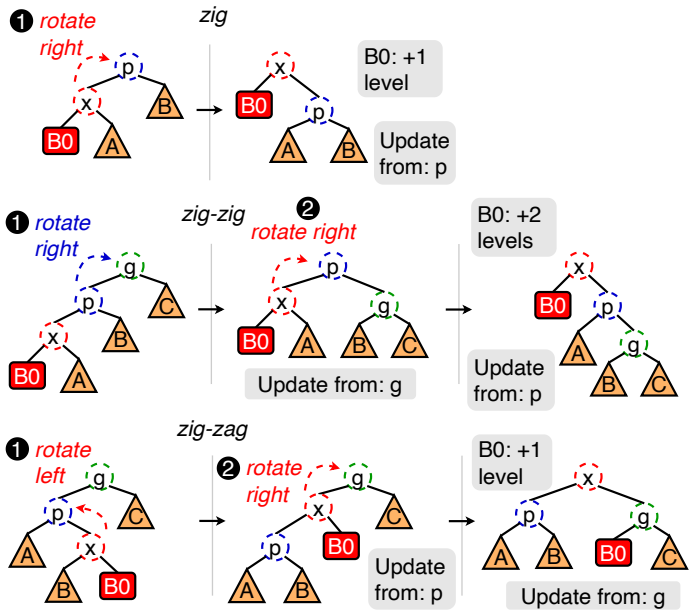


Fig. 10. Splay trees are a type of binary search tree that capture temporal locality by bringing an accessed leaf closer to the root. We use a splay-based hash tree design to similarly capture temporal locality in cloud block storage workloads.

data from a higher position in the tree. Finding an accurate and practical hotness metric is critical to balancing this trade-off.

Our approach to hotness tracking is motivated by the working set theorem for splay trees, which shows that frequently accessed elements naturally migrate toward the root over time. By tracking access frequencies through hotness counters, we can make informed decisions about splay distances that align with this theoretical behavior.

We attach an integer hotness counter to each tree node that is incremented whenever a node is promoted in the tree and decremented whenever a node is demoted in the tree. This applies to both leaves (i.e., blocks) and internal nodes (indicating the hotness of particular subtrees/blocks). The purpose of the hotness counter is to track the relative access frequencies of nodes as they are rotated, and use this information to determine how far to splay the node up the tree.

The counter is initialized to zero after the node is authenticated and cached; the hotness of nodes that are not currently cached in memory is therefore not tracked. The purpose of this is to localize our analysis of data hotness to the working set. Note that this approach can negatively affect performance for small caches, as it will be difficult to draw a contrast between relatively hotter, warmer, and colder data when counters are reset frequently. Nonetheless, the splay distance is a function of the hotness. The splay distance is computed in a straightforward manner: at a distance proportional to the hotness. For simplicity, we set the splay distance to be h levels, where h is the current hotness counter value of the accessed leaf.

Intuitively, nodes that are deeper in the tree (colder) will climb the tree slowly, while nodes that are higher in the tree (hotter) will climb quickly. Note that our initial exploration into this space could be expanded with sketching algorithms, machine learning, or other sophisticated techniques [32].

Promotion & Demotion. After computing the splay distance, the final step is to execute the splay operation. Splaying a DMT is done in nearly the same way as it is in a search tree. There are three cases to consider when splaying a node: *zig*, *zig-zig*, and *zig-zag* (Figure 10). In a zig case, the node's parent is the root, and we rotate the node up to the root. In a zig-zig case, the node's parent is not the root, and the node and the node's parent are either both left or right children. In this case, we perform two rotations along the same direction to rotate the node up two levels. In the zig-zag case, the node's parent is not the root, and the node and the node's parent are opposite-side children. Thus, we perform two rotations along opposite directions to rotate the node up two levels.

A consequence of splaying is that an accessed node will either be promoted two levels (or to the root). Neighboring nodes will similarly be promoted opportunistically as a side-effect of the splay. This provides two benefits. Nodes that are accessed frequently will therefore have an increasingly shorter path to the root, making verifications and updates quicker. More subtly, nodes that are accessed in close temporal proximity will slowly accumulate in nearby regions of the tree, allowing to exploit spatial locality within the tree.

Maintaining Hash Tree Invariants. We make three key changes to the standard splay operation to maintain three tree invariants. First, during a splay, we must ensure that a leaf node remains a leaf node and an internal node remains an internal node. Otherwise, a rotation will result in an invalid tree structure. For example, if a leaf node is splayed to the root, the root will become a leaf node, which is invalid. Whenever a block is read or written, we therefore execute a splay on the accessed leaf's parent rather than the leaf.

Next, we propagate the child status (left/right) to the splay operation, and swap the children of the parent node and the accessed node where necessary. This preserves the structural constraint for a valid hash tree while still ensuring the maximum degree of promotion for the accessed node.

Finally, splaying naturally introduces inconsistency into the tree, as it alters parent-child relationships. This will cause any subsequent hash fetches on a cache miss to fail due to an inconsistent root hash. We must therefore ensure that the tree remains in a consistent state by preemptively fetching (and authenticating) all sibling hashes before performing a rotation, then using them to commit the change immediately after. That is, parent hashes up to the root are recomputed per rotation (see "Update from" in Figure 10). While updates can be costly, splaying can reduce these costs over time.

Empirical Validation Strategy. The theoretical guarantees of splay trees provide confidence in DMT performance, but we can also validate this empirically. By comparing DMT performance against the optimal tree oracle (H-OPT) from Section 5 under known workload traces, we can concretely observe how closely DMTs approach optimal performance. If DMTs perform well relative to the oracle on fixed traces, this provides strong evidence that they will also adapt effectively to unknown or changing workload patterns, since the splay tree properties that enable good performance on known patterns are the same ones that enable adaptation to new patterns.

7 Implementation

We present two implementations of DMTs to demonstrate their applicability across different layers of the storage stack: a block-level implementation using custom block device drivers, and a file system-level implementation using FUSE. Both implementations validate the core DMT principles while addressing the unique challenges and opportunities present at each layer.

7.1 Block Layer Implementation

Our block layer implementation focuses on Linux platforms, where there is rich infrastructure in place to implement our custom integrity logic.

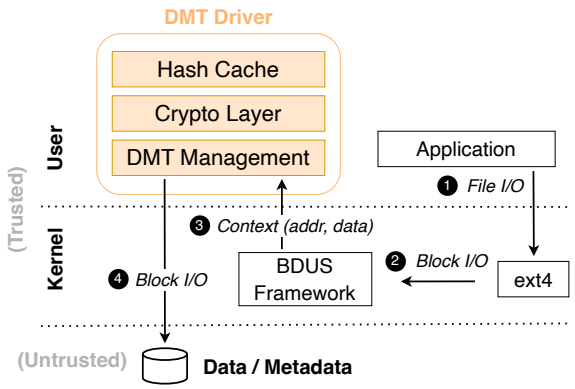


Fig. 11. DMT block layer implementation. Our implementation leverages the BDUS framework to intercept block I/O and invoke Merkle tree operations on the critical path.

Components and Interfaces. We implement the DMT in a block device driver in 5 K lines of C++. An overview is shown in Fig. 11. We use the BDUS (Block Device in Userspace) framework [27], which allows creating custom block device drivers that intercept storage I/O operations (reading and writing blocks). BDUS provides a kernel module that exposes block layer hooks to userspace through efficient shared memory channels. This enables us to process all I/O operations without kernel modifications while maintaining good performance.

The block device driver interface operates through two primary entry points: `read()` and `write()` functions invoked by the kernel whenever a block is accessed. Our driver architecture consists of three main components: (1) a DMT management engine that maintains tree structures in memory and executes splay operations based on access patterns, (2) a cryptographic layer that handles MAC generation, hash computation, and integrity verification, and (3) a cache layer that stores and retrieves tree nodes from disk as needed.

Reads and Writes. For read operations, the driver intercepts the kernel's block read request, fetches the encrypted block data and its associated leaf (MAC) in the DMT from the underlying storage device, verifies the MAC against the block content, then traverses the authentication path in the DMT to verify integrity against the root hash. If splay conditions are met based on configured probability and window settings, the accessed node's parent is splayed according to hotness-based distance calculations. For write operations, the driver computes a new MAC for the modified block data, updates the corresponding leaf in the DMT, and propagates hash updates up to the root. Note that splays may consist of several rotations. After a single rotation is performed, we immediately commit the tree structure change by recomputing hashes from the rotation point up to the root. Tree rotations must be handled this way to maintain internal consistency—otherwise, trying to recursively fetch and authenticate nodes after a rotation would result in failed verifications. Note that our basic data unit for tree nodes aligns with the disk I/O size (4 KB blocks) [13, 16, 36, 48].

Node Addressing Scheme. A key implementation challenge is that DMTs cannot use implicit indexing schemes like balanced trees. Unlike binary trees where parent-child relationships can be computed arithmetically from array indices, splay trees require explicit parent-child pointers that must be maintained correctly across rotations. We address this by using a two-tier node addressing scheme where leaf nodes use the lower part of a 64-bit address space, with addresses

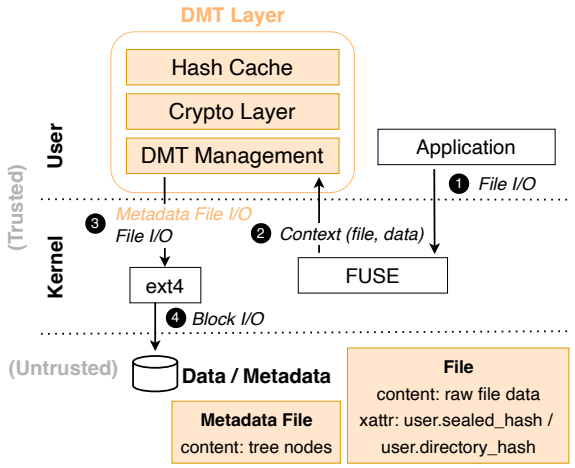


Fig. 12. DMT file system layer implementation. Our implementation leverages the FUSE framework to intercept file I/O and invoke Merkle tree operations on the critical path. The per-file Merkle root is attached to each file via an xattr, and the per-file tree is stored in a separate metadata file.

corresponding to their block IDs, while internal nodes use high addresses. This allows deterministic indexing for on-disk node lookup. Intuitively, during lookup we identify whether an address is a leaf node or internal node via the high bit. However, node structure must still explicitly store parent-child relationships as integer node IDs, enabling efficient tree traversal and updates during splay operations. We designate a fixed on-disk address to always contain a pointer to the current root node, ensuring consistent root identification despite frequent root changes during splaying.

This addressing scheme requires additional storage overhead: one parent pointer for leaves and up to three pointers (parent, left child, right child) for internal nodes. This amounts to approximately 72 B per node on disk. For example, a 1 TB disk has approx. 268M 4 KB blocks and therefore 536M tree nodes. A hash cache size of 10% would hold 53.6M nodes, and with 72 B nodes would thus occupy 3.8 GB of memory. While the storage requirement is higher, this enables the dynamic restructuring essential for DMTs, and we will show in the evaluation that the benefits outweigh the costs.

7.2 File System Layer Implementation

Our file system layer implementation moves up one layer of abstraction. Like the block layer implementation, there is rich infrastructure in place to implement custom file system hooks to execute our integrity logic. However, file systems have a much richer interface than the block layer, which requires additional state tracking across operations. Our implementation departs from prior approaches in three ways: per-file adaptive trees, global cache coordination, and incremental tree construction.

Components and Interfaces. We implement the DMT file system layer in 2 K additional lines of C++. An overview is shown in Fig. 12. We use FUSE (Filesystem in Userspace), which provides a userspace framework for implementing custom file systems without kernel modifications. FUSE intercepts file system operations through a kernel module and routes them to userspace handlers

via efficient communication channels. This enables us to enforce integrity protection on all file operations while maintaining compatibility with existing applications and underlying file systems.

The FUSE interface operates through file system operation callbacks invoked whenever files are accessed. Our implementation hooks into key operations including `open()`, `read()`, `write()`, `flush()`, and `release()` to intercept file accesses and enforce integrity verifications and updates.

Tree organization and granularity. Our file system uses per-file DMTs. We maintain a leaf MAC per block in each file. The per-file Merkle roots are stored in each file's extended attributes. This tree organization strategy enables each file's DMT structure to adapt independently to its observed access patterns. For example, a frequently accessed configuration file develops a highly unbalanced tree optimized for its hot blocks, while a large media file accessed sequentially maintains a more balanced structure. This per-file specialization enables access pattern optimization, reducing per-file hashing costs. It also reduces the scope of verifications and updates (i.e., tree traversals are localized) compared to a global tree approach, reducing hashing costs.

Cache Management. Although we use per-file trees, we use a global cache across all files, rather than maintaining separate caches per file. This is similar to how the OS page cache operates for all file system and memory blocks. Merkle tree leaf and internal nodes are indexed using composite keys (*node_id*, *inode_num*). This design provides easier cache management for administrators (e.g., re-sizing) and enables richer cache replacement policies on a per-file or per-directory basis via the keys. Note that we use the same node addressing scheme as in the block layer implementation.

Global Directory Merkle Tree. Using the per-file Merkle roots, the system constructs a global integrity tree that mirrors the file system's directory structure. This creates a variable-arity tree where each directory node has children corresponding to its files and subdirectories. Per-file Merkle tree roots are sealed in extended attributes (`user.sealed_hash`), and the directory hashes are similarly stored in their extended attributes (`user.directory_hash`). When file contents change, extended attributes are updated and propagated upwards through the directory tree to the root directory. A file's tree nodes themselves are stored in separate files `filename.leaf` and `filename.internal`, creating an integrity metadata overlay that operates independently of the underlying file data. This multi-level approach enables integrity protection at both the individual file level and the directory structure level, providing global data integrity while ensuring that files have not been moved or renamed maliciously.

Static vs. Dynamic Allocations. Prior works rely on balanced trees, which introduces significant tree management challenges. Namely, the tree size must be known ahead of time in order to construct it, then the tree must be reconstructed at runtime if the file size exceeds the tree size. This means that any changes in file size can lead to costly tree rebuilds and significant performance degradation. Prior works provide no support to deal with this problem, yet file sizes often grow or are truncated frequently by modern applications. We address this by introducing a new incremental tree construction approach that departs from static allocation strategies used in prior work. In our file system implementation, DMTs support dynamic tree growth by placing the current root's children under a new subtree and pushing a new leaf node (the new file content) under the root whenever the file size grows. Under normal circumstances, this tree construction technique may lead to a degradation of the tree structure. However, the DMT algorithm takes over and continuously re-adjusts the tree structure on-the-fly to better align with the current workload pattern. This approach provides much simpler tree management without the overhead of tree rebuilding or size estimation. Note that our implementation provides support for both static (for all tree types) and dynamic allocation (only for DMT) for experimental purposes.

Parameter	Description
Capacity	Usable capacity for data blocks
Cache size ratio	Cache size as % of tree size
Read ratio	% of read operations
I/O size	Size of application I/O
I/O depth	Max no. outstanding application I/Os
Thread count	Number of application threads

Table 1. Experiment parameters.

Reads and Writes. Read operations traverse the per-file DMT to verify integrity, then proceed upward through the global integrity tree. Finally, the underlying file data is accessed. Write operations update the corresponding DMT leaf, propagate hash updates up to the per-file root (via the `user.sealed_hash` attribute), then propagate updates through the directory tree (via the `user.directory_hash` attributes). Splay operations are triggered similar to the block layer implementation (randomly on a specified percentage of accesses). The cache size is set at init time and can be configured dynamically in response to workload changes via an IOCTL interface.

8 Evaluation

Our evaluation examines DMT performance at both the block layer and file system layer. We compare DMTs against two insecure baselines (No encryption/no integrity, Encryption/no integrity), the balanced binary trees used by Linux dm-verity and fs-verity [1, 60], and the optimal binary tree (H-OPT). We also juxtapose DMTs against the high-degree trees that have been widely used in secure memory systems [28, 57]. Finally, as noted previously (see Figure 6), we also examine DMTs with respect to 4-ary and 8-ary trees, which have not been considered by prior work. An overview of our parameters is shown in Table 1.

8.1 Experiment Setup

Testbed. We perform all experiments using AWS EC2 i4i.8xlarge instances equipped with 32 cores, 256 GB memory and locally-attached NVMe SSDs for data and metadata. Note that currently there are no available cloud instance types which both have local NVMe storage and also support confidential VM technology such as AMD SEV-SNP. We reinitialize hash trees between each experiment, use a standard LRU cache replacement policy, and we set the splay window flag $w = True$ (enables/disables splaying) and splay probability $p = 0.01$ (percentage of accesses to splay on) for DMTs.

Cryptographic Settings. Like prior works, we ensure deterministic authenticated encryption with AES-GCM [6, 57]. We use a 128-bit encryption key for blocks. The MACs produced during the encryption process are used as the leaves in the hash tree. For internal nodes, we compute 256-bit hashes using SHA-256 with a 256-bit key.

Workload Settings. We perform a broad analysis across different system and workload configurations, parameterized by disk capacity, hash cache size, read/write ratio, I/O size, thread count, and I/O depth. This enables exhaustively examining the performance space of DMTs at the storage layer. We also examine different degrees of workload skewness, from pure uniform to highly skewed. We focus especially on the Zipfian workload discussed in Section 4, which closely approximates

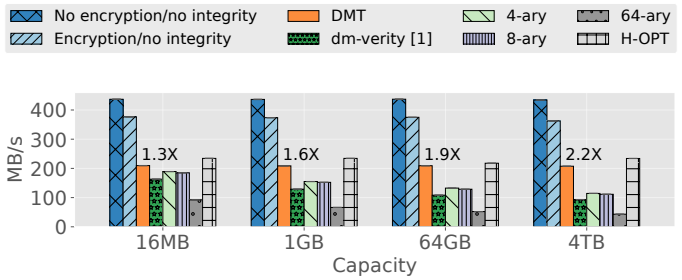


Fig. 13. Aggregate throughput with Read ratio at 1%. DMTs provide up to 2.2× higher throughput than the state of the art, demonstrating its ability to scale well with capacity.

real-world block-level access patterns, which are highly skewed (and write-heavy) [40, 63]; see Figure 25. We also use a recently published Alibaba dataset recorded from an array of 1000 volumes backing various virtual machines in a public cloud datacenter [40]. Finally, we demonstrate how storage-level improvements translate to application-level improvements with a case study of the Filebench OLTP workload [59].

Like prior works, we generate workloads with fio [8]; we record/replay traces for the optimal. Workloads have a 5 minute warmup period and 15 minute benchmark period.

8.2 Block Layer Results

We focus our block layer analysis on three questions:

(1) *How well does the state of the art (dm-verity) perform across the various system and workload settings that characterize cloud block storage deployments?*
(2) *To what extent can DMTs improve performance over the state of the art, and under what conditions?*
(3) *What memory and storage trade-offs do DMTs make?*

Scaling with Capacity. We first analyze how disk capacity (which affects tree height) and hash cache size affect performance. Where appropriate, default parameters include—Read ratio: 1%, I/O size: 32 KB, Thread count: 1, I/O depth: 32, Capacity: 64 GB, Cache size: 10%. We choose these parameters because they showcase the best performing configuration for the baselines. We examine various workload shapes ranging from uniform to highly skewed Zipfian. We focus particularly on θ: 2.5 because it closely approximates the shape of real-world storage workload patterns [40, 63] (see Figure 25).

Figure 13 shows that aggregate read/write throughput decreases w.r.t. capacity for all balanced trees. Note that the Zipfian workload is emitted from an *i.i.d.* source and is therefore an exact upper bound. We observe that 64-ary trees are the worst performing: reduced tree height can reduce the effective number of hashes that must be computed, but results in lower cache efficiency, which amplifies metadata I/O costs. The state of the art binary trees incur up to a 75% throughput loss over the Encryption/no integrity baseline at 4 TB. 4-ary and 8-ary trees similarly suffer from a 70% throughput loss at 4 TB. In contrast, DMTs consistently deliver the highest throughput and >85% of optimal throughput across all capacities. This clearly demonstrates that DMTs can scale to higher or lower capacities more efficiently. And as noted in Section 4, with even faster storage devices, the proportion of time spent hashing vs. performing the data access will grow substantially, increasing our observed DMT speedups.

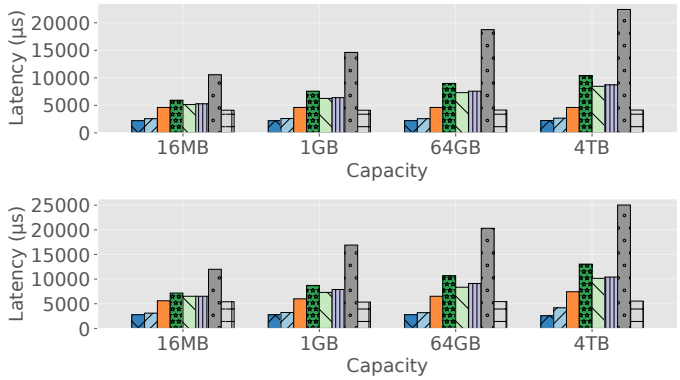


Fig. 14. P50 (top) and P99.9 (bottom) write latency. DMTs median and tail latencies reflect throughput improvements, demonstrating it can provide a stable performance guarantee.

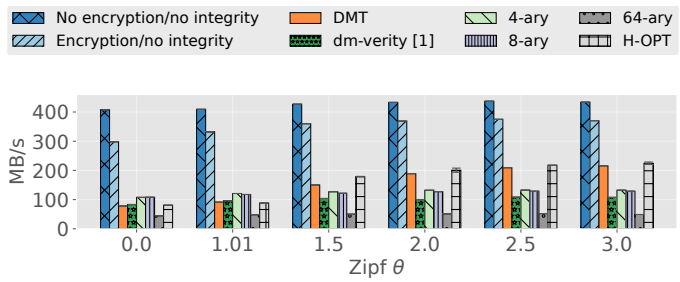


Fig. 15. Aggregate read/write throughput. DMTs provide larger speedups as the workload skew increases; under uniform workloads they observe a 6% cost over balanced binary trees due to exploratory splays.

Latency improvements are the same—DMTs splay on 1% of accesses and those costs are amortized over time because splaying occurs most frequently on hot data. Figure 14 corroborates this: DMT median and tail write latencies are still significantly lower than the state of the art.

Impact of Workload Skewness. Figure 15 shows how performance changes w.r.t. workload skewness. We observe again that 64-ary trees are the worst performing. DMTs provide up to 2× speedups over the state of the art binary trees under heavy skew, but incur a 6% cost under more uniform patterns due to exploratory splays which yield no benefit. We attribute this low cost to the fact that DMTs inherit the theoretical guarantees of splay trees, which provide $O(log\ n)$ amortized lookup (i.e., verification or update) time. Thus, DMTs perform at least as good as balanced (binary) trees on average.

We also observe that 4-ary and 8-ary trees deliver 25% higher throughput than DMTs under more uniform workloads. As discussed in Section 4, low-degree trees hit the optimal points in the design space for *balanced* trees (reduced tree height without adverse effects on cache performance); prior works have not considered this. However, when workloads become skewed, there is a substantial opportunity cost: 4-ary and 8-ary trees do not perform better than the optimal binary tree. This highlights a key observation: increasing tree degree alone is not sufficient to maximize performance.

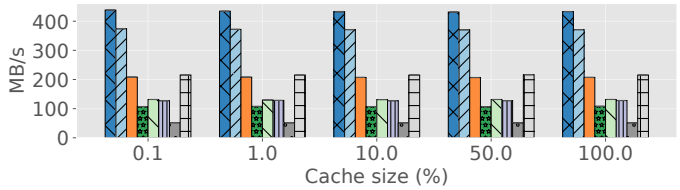


Fig. 16. Aggregate throughput. DMTs maintain the highest throughputs across both small and large cache sizes.

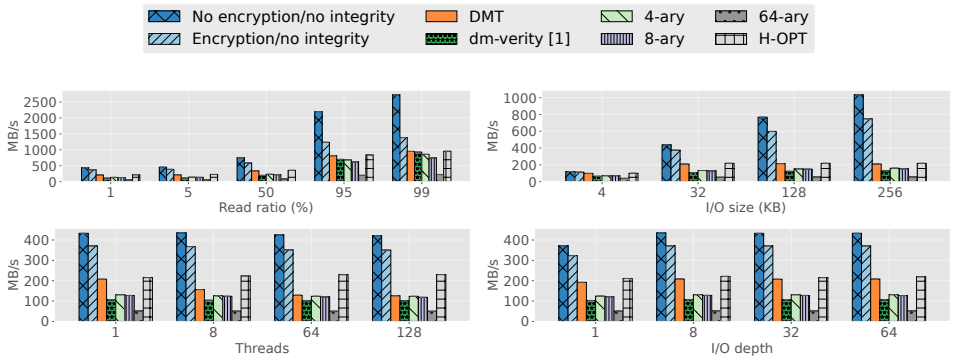


Fig. 17. DMTs show speedups across different read ratios, I/O sizes, thread counts, and I/O depths.

We believe that extending the DMT design to 4-ary and 8-ary trees will yield the most performant solution.

Impact of Cache Size. The above analysis showed that DMTs can exploit skewed patterns in workloads when present and deliver a stable performance guarantee across various capacities. Now we examine the effects that other system settings and workload characteristics have on performance. The goal is to evaluate whether these conclusions about DMTs hold broadly. We continue with the Zipf(2.5) workload.

Figure 16 shows that DMTs maintain the highest throughputs across both small and large hash caches. Note that cache size is specified as a ratio of the tree size; the absolute cache size varies with capacity. Further, caches mostly benefit read I/Os (because they enable early returns), but write I/Os still must traverse the entire path to the root. In general, we observe that increasing cache size beyond 0.1% does not yield significant performance improvements for any hash tree design; small caches are already very efficient. Yet, losses observed by all balanced tree designs are still significant. This shows that caching only helps to an extent—when caches are efficient, hash tree overheads are largely attributable to the structure of the tree. However, DMTs still deliver near-optimal performance and the highest across all cache sizes.

Impact of Read Ratio, I/O Size, Thread Count, and I/O Depth. Figure 17 (top) shows how the performance changes with respect to the read ratio. We expect that at higher read ratios, DMTs, balanced, and optimal trees will all observe higher absolute throughputs, as reads can be quickly served by early returns due to caching. However, when there is a significant proportion of writes ($\leq$ 50% read ratio), DMTs provide nearly 2$\times$ higher throughput than balanced trees. Since storage

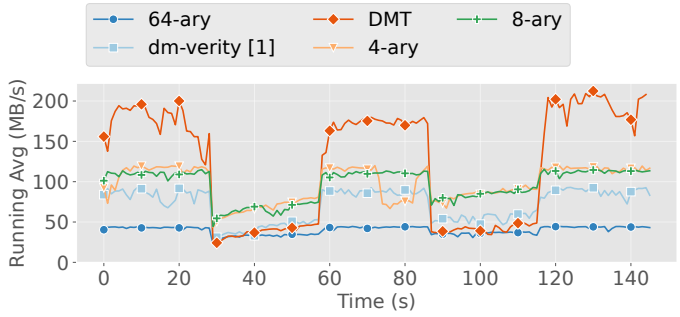


Fig. 18. DMTs can adapt quickly to changing workloads, exploiting skewed patterns when present.

access patterns tend to be write-heavy (due to application-level caches and the OS page cache), this shows that DMTs can more reliably handle write-heavy workloads, while delivering comparable performance under read-heavy workloads.

The remaining graphs in Figure 17 reflect the above observations. Baseline throughputs increase w.r.t. I/O size, but hash tree performance saturates at 32 KB I/Os—larger I/O sizes only lead to increased latencies without improved throughputs. A single thread is sufficient to saturate the device bandwidth. And an application I/O depth of 32 is sufficient to saturate the device bandwidth. DMTs still deliver up to 2× and 4× higher throughput over the state of the art binary and 64-ary trees. The extent to which DMTs see speedups is thus best attributed to the workload shape (Figure 15).

As noted in Section 4, 32 KB write I/Os require 8 sequential hash tree updates. State of the art works are not truly concurrent, but still rely on a global tree lock to serialize tree updates [28, 29, 57]. Performance has been shown to improve under high concurrency by using lazy verification (deferring and batching updates) [3], but lazy verification violates freshness guarantees. Designing concurrency-optimal hash trees (and search trees in general [54]) is an open problem.

Handling Changing Access Patterns. We now demonstrate that DMTs self-adapting nature is robust to changing workload patterns. Figure 18 shows a 150-second snapshot of sampled throughputs under a workload that exercises an extreme case where patterns alternate between uniform and skewed: Zipf(2.5) > Uniform > Zipf(2.0) > Uniform > Zipf(3.0). Phases are 30 seconds long, and the Zipfian phases are randomly centered at a new region in the address space. We observe that DMT throughput spikes within a few seconds of entering the Zipfian phases and DMTs maintain the speedup throughout. This shows that DMTs can capitalize on skewed patterns very quickly to maximize performance while delivering performance comparable to binary trees otherwise. As noted, extending DMTs to 4-ary trees can help further improve DMT performance during uniform phases.

Moreover, Figure 19 illustrates the latency distribution when splay operations are triggered during write operations. Most splay invocations exhibit latencies around 150 μs, with a long tail extending to higher latencies (up to 1 ms for the 99.9th percentile). While individual splay operations can be costly—requiring tree rotations and potentially causing additional cache misses during path restructuring—these costs are effectively amortized over subsequent accesses due to temporal locality in the workload. Once a frequently accessed node is splayed closer to the root, it reduces the path length for future accesses to that node and nearby nodes in the access sequence. The relatively low frequency of splay operations (1% of total accesses with $p = 0.01$) combined with

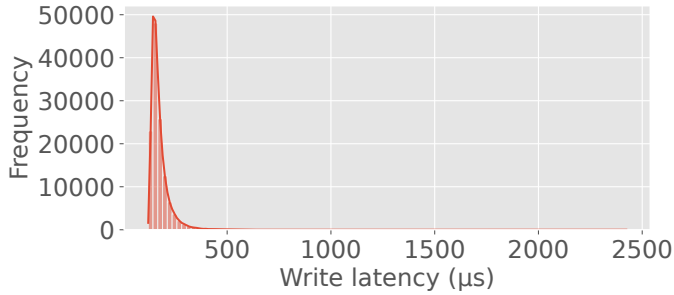


Fig. 19. Driver `write` latency when a splay is invoked.

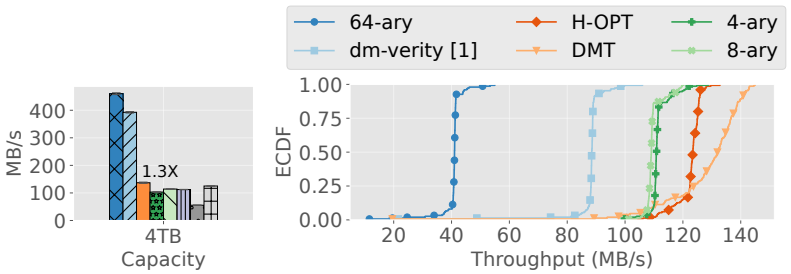


Fig. 20. On an Alibaba cloud volume trace, DMTs deliver notable speedups, while high-degree trees perform worst.

their concentration on hot data means that the performance benefits of reduced tree traversal costs for subsequent operations outweigh the occasional higher latency of the splay operation itself.

Case study: Alibaba Cloud Volumes. We showed that DMTs perform near-optimally under a broad range of system and workload settings. We now examine how these observations hold under a real workload sampled from a recently published Alibaba trace dataset [40] (logical volume ID 4). We scale the offsets and I/O sizes proportionally to the experiment capacity. Note that the remaining volume traces are qualitatively the same (mean write ratio >98% and highly skewed). Further, note that the workload is non-*i.i.d.* and therefore H-OPT can underestimate the upper bound on throughput; temporal patterns enable DMTs to perform better in some cases.

Figure 20 (left) shows the aggregate throughputs observed at a 4 TB capacity, and Figure 20 (right) shows the distribution of write throughputs (sampled at 1-second intervals). Binary trees observe a 75% throughput loss at 4 TB capacity, while 64-ary trees observe an 88% throughput loss. DMTs provide a 1.3× speedup over the binary trees and a 1.2× speedup over the 4-ary trees. Importantly, the optimal (binary) tree still observes 15% higher throughput than 4-ary and 8-ary trees. This further supports our claim that a balanced tree structure is not sufficient to maximize the performance potential. As noted above, we believe the extending DMT principles to a 4-ary tree can help achieve maximum performance.

Case study: OLTP Workload. We now consider the Filebench OLTP workload, an exemplar application that is commonly run in the cloud and requires robust security protections [25]. The goal is to evaluate how DMT device-level improvements translate to application-level improvements. The workload consists of 10 writer threads and 200 reader threads and is write-heavy. We run

	DMT	dm-verity	No enc/no integrity
write	255.4 MB/s	151.9 MB/s	318.8 MB/s
read	0.7 MB/s	0.4 MB/s	1.0 MB/s

Table 2. Application read/write throughputs for the Filebench OLTP workload. DMT driver-level improvements are reflected at application-level.

	Memory Overhead	**Storage Overhead**
leaf nodes	0.44×	0.29×
internal nodes	0.80×	0.75×

Table 3. DMTs require additional memory/storage for tree nodes, but break even on this trade-off: they provide higher performance than balanced trees, at a smaller cache budget.

the workload for 10 minutes on a 1 TB disk (with a dataset size of ≈922 GB) formatted with ext4 and using a hash cache size of 10%. Table 2 shows that DMTs have 1.8× improved read and 1.7× improved write performance over the state of the art. We are currently expanding our application analysis to other workloads.

Memory & Storage Overhead. DMTs provide several advantages, but have higher memory and storage requirements than balanced trees (Table 3). DMTs cannot use implicit indexing like balanced trees, but instead require explicitly storing parent-child pointers (as integer node IDs) both for nodes in-memory and on-disk. This implies at least one additional integer field for leaf nodes, at most three additional integer fields for internal nodes, and one additional integer hotness counter field for all nodes. However, we showed that cache hit rates are very high even for very small caches. For example, DMTs provide better performance at a cache size of 0.1% than binary trees do at a cache size of 1%. Thus, DMTs deliver better performance per dollar spent on cache memory.

Block layer evaluation summary: state of the art block layer hash trees incur substantial performance loss (up to 80%). DMTs address this by exploiting skewed workload patterns and adapting quickly to changes over time. Note that often applications might only exhibit skewed behavior over short bursty periods, which can amplify tail latencies. DMTs adapt to this behavior. We conclude that balanced trees are ill-suited as the base construction of a hash tree: maximizing performance requires tailoring the tree structure to workload patterns.

8.3 File System Layer Results

Next we examine how DMTs perform at the file system layer. We note that DMT performance across all parameters are quantitatively and qualitatively the same to those observed at the block layer. We therefore focus primarily on aspects unique to the file system layer here. We also note that for comparison against balanced trees, we must use a static allocation strategy where the a tree of a default size is allocated at file creation time (set to the size of the fio workload file). We focus our file system layer analysis on two questions:

(1) *How is DMT performance affected by workload characteristics?*

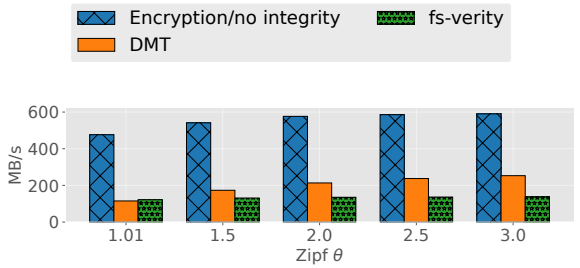


Fig. 21. Aggregate read/write throughput. DMTs provide larger speedups as the workload skew increases.

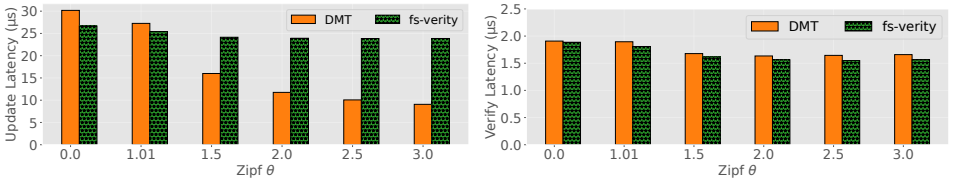


Fig. 22. These graphs shows that DMTs significantly reduce per-block Merkle tree update latencies, particularly as workload skew increases. DMTs provide verification latencies comparable to balanced trees, with negligible differences.

(2) *How well can DMTs specialize to per-file access patterns?*

Impact of Workload Skewness. We first analyze how workload skewness at the file system layer affects performance. The experiment parameters include: read ratio: 20%, I/O size: 32 KB, Thread count: 1, I/O depth: 32, File size: 64 GB, Cache size: 10%, and a Zipfian workload with $\theta = 2.5$. As above, we choose certain parameters (e.g., I/O depth) because they showcase the best performing configuration for the baselines; varying other parameters may only change absolute (but not relative) performance improvements.

Figure 21 shows that DMTs provide up to 1.8× speedups over fs-verity under heavy skew, but incur a 10% cost under more uniform patterns due to exploratory splays which yield no benefit. We attribute this low cost to the fact that DMTs inherit the theoretical guarantees of splay trees, which provide $O(log\ n)$ amortized lookup time. Thus, DMTs perform at least as good as balanced trees on average. We note that while pure uniform workloads are rarely observed in practice, if administrators are aware that workloads are in fact uniform, they can immediately disable DMT splaying via an IOCTL to avoid this penalty.

Moreover, Figure 22 shows a more fine-grained view of the tree update and verification latencies. Like Figure 21, it shows that as the workload exhibits higher skew, the splay mechanism is able to keep hot data closer to the root and reduce per-block update latencies from 23.75 μs to 8.75 μs under heavy skew (a 2.7× reduction). While updates benefit from caching to an extent, cache hits prompt early exits on nearly all verification operations, making verification latencies largely negligible. They are already very fast; the latency difference between DMTs and balanced trees is negligible. This is significant because prior works [3, 4] have relied on deferring verifications to offset costs. This shows that not only is deferred verification insecure, but it is also unnecessary. Moreover, it shows that the limitations of prior works cannot be solved by simply using a larger cache size.

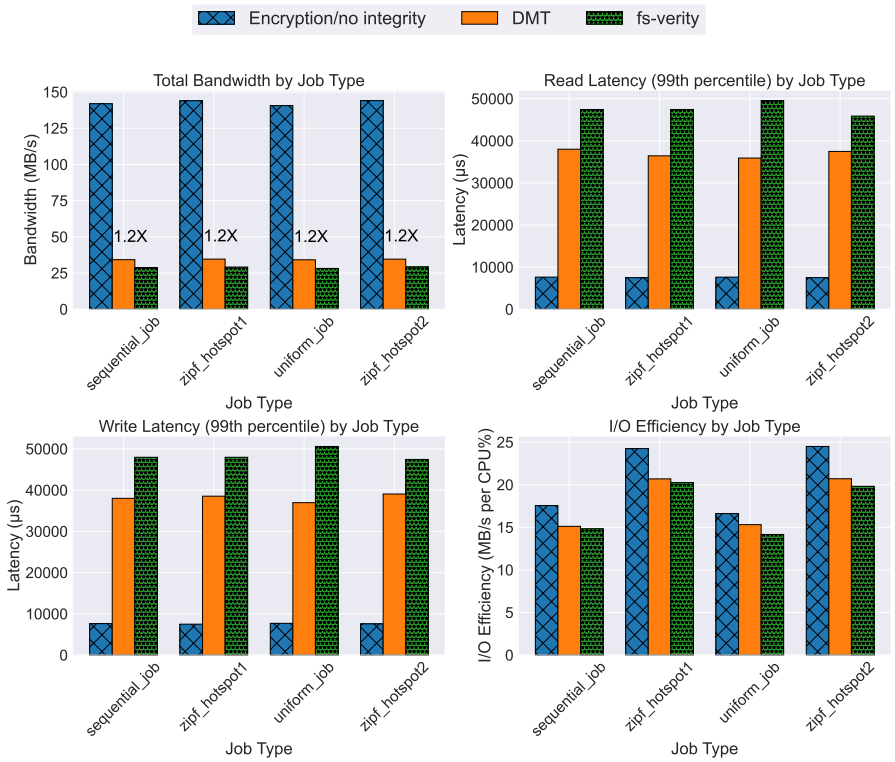


Fig. 23. This graph demonstrates per-file specialization through a workload that exercises three different access patterns: sequential, Zipfian, and uniform random. When all patterns are executed concurrently by different threads, DMTs performance improvements are distributed across all jobs, with a 1.2× average speedup over fs-verity. DMTs also provide greater I/O efficiency, demonstrating that the increased CPU cycles spent on splaying are an algorithmic efficiency gain rather than simply increased resource consumption.

Per-File Specialization. One of the key advantages of the file system layer implementation is the ability to specialize hash tree structures for individual files based on their specific access patterns. Figure 23 and Figure 24 demonstrate this capability using a workload that exercises three different access patterns: sequential, Zipfian, and uniform random. Note that a sequential workload is uniform random in nature because all blocks are accessed with equal probability. Moreover, similar to Figure 18, we observed that when running different patterns in phases across a single file, DMTs can adapt quickly to the changing workload patterns and provide up to a 1.7× speedup over fs-verity trees (see Figure 24). Here we focus on running different patterns concurrently across different files.

When jobs run concurrently (Figure 23, top left), each per-file tree adapts to its respective access pattern, but threads contend for locks when accessing the shared directory-level Merkle tree. This creates system-wide performance effects. It produces lower per-job speedups, but the performance gains appear to be distributed across all jobs. We observe a 1.2× overall improvement when running concurrently, compared to the 1.7× speedup for Zipfian patterns and 10% loss for uniform/sequential patterns when jobs run individually with stonewalling. This distribution effect likely stems from shared cache warming, where the Zipfian job's splaying operations bring frequently accessed tree

nodes into the shared cache, benefiting subsequent operations across all jobs regardless of their individual access patterns.

I/O Efficiency. We also observe in the bottom right graph that the I/O efficiency is greater than that of the fs-verity trees. We compute the I/O efficiency as MB/s per CPU%, and it exposes whether performance gains stem from algorithmic advantage or computational brute force. When DMT exhibits higher efficiency than fs-verity trees, it indicates that the opportunistic splaying strategy is making algorithmically superior decisions—each CPU cycle invested in tree adaptation yields more than proportional performance benefits. This suggests the splaying overhead is well-targeted, improving tree structure in ways that reduce the average computational cost per I/O operation. Conversely, when efficiency falls below fs-verity (even in the presence of throughput speedups), it reveals that DMT would be achieving speedups through increased computational expenditure (i.e., via brute-force) rather than algorithmic advantage. DMTs achieve up to a 10% computational efficiency gain over fs-verity, validating that they are algorithmically better rather than simply trading off one resource for another.

File system evaluation summary: DMTs deliver consistent performance improvements at the file system layer, validating their effectiveness beyond block-level storage. Per-file specialization enables optimization for heterogeneous access patterns across the same or different files, and I/O efficiency gains demonstrate algorithmic advantage of DMTs over the static balanced trees used in prior works.

8.4 Key Takeaways

DMTs perform near-optimally, and the primary determinant of DMT performance at either the block or file system layer is workload skew. In other words, if an application workload exhibits more skew, DMTs will deliver a higher speedup. Different applications exhibit different degrees of skew, some very long-tailed and some less so. Some applications might also exhibit persistent periods of skew or bursty episodes of skew (e.g., after a new product launch or software upgrade). DMTs can efficiently accommodate all of these cases, whereas prior approaches do not. One key challenge for future work, however, is improving DMTs to potentially use variable arities. Specifically, the hierarchical structure of Merkle trees inherently smoothes out the underlying leaf access distribution—i.e., the actual tree structure is naturally less skewed than the workload driving it. Leveraging a mix of slightly wider (8-ary) trees for shallow, hot subtrees may help improve performance further.

Perhaps most important are the broader implications of our DMT design beyond what our evaluation immediately shows. Beyond skew, DMTs also enable prediction-guided re-shaping: the tree can pre-position targeted leaves, or elevate entire subtrees, before the next access so those updates traverse shallow paths and thus have lower update costs. This capability is orthogonal to gains achieved from increased skew: even modest reuse or short spatial runs can be exploited by preemptively lifting subtrees, and heavier skew amplifies the benefit by keeping predicted regions hot longer. This mechanism can be per-file and opportunistic: when prediction confidence or timing slack is present, we can reshape in anticipation; otherwise, DMTs can revert to standard behavior. *Our evaluation already provided an existence proof that reshaping can reduce tree traversal costs without excessive tree management overheads.* Thus, a natural extension is to extend our heuristics to figure out how to more intelligently learn, anticipate, and preemptively reshape the tree. We defer this to future work. The key point is *capability*: unlike static, balanced trees, which cannot reduce future path cost even with a perfect predictor, DMTs make prediction actionable, transforming integrity from a fixed tax into a tunable, workload-aware cost.

9 Discussion & Future Work

Our experiments highlight several key insights and raise important questions for future research.

Limitations: The Tail Latency Trade-off. One might intuitively expect that high-degree trees (e.g., 64-ary) would offer the best performance by minimizing tree height. However, as demonstrated in Section 4, the computational cost of hashing many children nodes outweighs the benefits of reduced depth, making low-degree (e.g., binary) trees the superior design choice for modern storage hardware. Yet, optimization for low-degree trees introduces a fundamental trade-off: the maximum tree height is inherently larger ($\log_2 N$ vs $\log_k N$).

Consequently, a potential limitation of the DMT approach is the latency penalty for accessing "cold" data that resides at the bottom of a deep binary tree. While our splay mechanism effectively mitigates this for the vast majority of accesses by keeping hot data near the root—evidenced by our improved p99.9 latencies in Figure 14—spurious accesses to cold regions or pathological workloads that constantly shift the working set could theoretically expose these deeper traversal paths. This could theoretically manifest as a long extreme tail in the latency distribution (e.g., >p99.99%) for accesses that fail to benefit from locality. Future work could explore hybrid designs that enforce strict height bounds on the tree structure, potentially sacrificing some degree of optimal hot-path shortening to provide deterministic worst-case latency guarantees.

While the focus of our work is on characterizing and optimizing the underlying integrity data structure, system crashes also introduce additional concerns. As with traditional storage, if a failure occurs while updates are buffered in memory, those contents are lost and the tree must be rebuilt from persistent state to restore consistency. Future work could explore optimized checkpointing mechanisms or log-structured approaches that minimize this recovery window without sacrificing the performance gains of in-memory buffering.

Identifying the Real Performance Bottleneck. Perhaps most importantly, we observed overwhelming evidence that hash tree traversal costs are the primary performance bottleneck for storage integrity, and conventional optimizations like caching are fundamentally limited in their ability to address this bottleneck. Prior works have largely focused on memory-based integrity protection, perhaps under the presumption that storage integrity overheads are negligible. We found that this is not the case. Moreover, we have only recently been able to make this observation because emerging storage devices operate in the microsecond range, which directly contends with modern CPU capabilities. This has forced us to rethink the design of storage integrity mechanisms for the modern era. The core limitation is at the *data structure level*—static tree organizations cannot adapt to the diverse and dynamic nature of real-world workloads.

DMTs as a Necessary Condition for Optimal Performance. Our evaluation suggests that DMTs represent a necessary condition for achieving optimal performance in integrity-protected storage systems. We establish this through three key insights: (1) DMTs achieve near-optimal performance (>85% of H-OPT) while balanced trees suffer dramatic losses (up to 80%), (2) DMTs successfully exploit workload reference locality through adaptive reorganization, and (3) DMT performance remains robust across diverse system and workload parameters. Importantly, even when the optimal tree performance falls short of the insecure baseline, DMTs consistently maximize what is achievable within the constraints of cryptographic integrity guarantees. This suggests that adaptive integrity data structures are not merely beneficial optimizations, but fundamental requirements for high-performance secure storage. By transforming the integrity mechanism from a static problem into a dynamic one, DMTs establish a new paradigm where tree structures can evolve to match the workload patterns they serve.

Implications of Data Granularity. A key takeaway from our analysis is that the benefits of adaptive integrity structures depend on the granularity of the protected storage unit. In block layer implementations, the Merkle tree covers the entire capacity of the device, resulting in deep trees where traversal overheads are substantial. In contrast, file systems that utilize per-file Merkle trees partition this space; for small files, the resulting trees *could* be relatively shallow, reducing the relative impact of traversal costs. However, many important applications manage massive data objects. For example, database, archive, and backup systems often rely on massive files with frequent random access, making DMTs a valuable optimization.

Beyond Data Structures: The Need for Holistic Adaptivity. Looking ahead, several critical research questions emerge from our work. First, *can adaptivity extend beyond node movement to encompass variable tree structure?* Our results demonstrate that while high-degree trees (64-ary) perform poorly across all workloads, moderate-degree trees (4-ary, 8-ary) can offer performance improvements under some circumstances. Leveraging adaptivity across multiple dimensions simultaneously (arity, balance, etc.) could potentially address the performance ceiling we observe even with optimal binary trees. Second, *how can adaptive integrity mechanisms achieve true concurrency?* Current hash tree implementations rely on global locks that serialize updates, and our evaluation reveals this as a fundamental scalability bottleneck. Designing concurrent tree structures is an open problem, and designing a concurrent hash tree introduces additional challenges. Future work should explore lock-free or fine-grained locking techniques specifically designed for dynamic tree structures, potentially drawing from concurrent data structure research. Third, *can cross-layer co-design enhance adaptive performance?* Perhaps, exposing application-level access patterns to the storage-layer could enable better tree organization and improve performance. This raises questions about new APIs or interfaces that allow applications to hint at access patterns, and whether predictive techniques could anticipate workload shifts.

10 Related Work

Hash trees are a core component of many computing systems, including blockchains, secure memories, etc. [1, 3, 6, 15, 26, 28, 31, 39, 42, 47, 55, 57]. We discuss these related works below.

Secure Memory. Secure memories have been long-studied [28, 31, 52, 62]. Their goal is to provide a secure environment in which the secrecy and integrity of application code and data can be assured. Recently, secure memories have seen widespread commercial success through implementations such as Intel SGX [17, 42, 50]. Consequently, recent work has shown that hash trees can severely degrade memory performance, and optimizing them has been a central focus of recent research. State-of-the-art approaches, such as Penglai [28], FastVer [3], and VAULT [57], have shown that optimizations like caching and increasing tree degree can lower overheads in some contexts (e.g., at small capacities).

However, we focus on persistent storage rather than main memory. Storage systems are subject to vastly different workload characteristics, capacities, and cache behaviors than memory systems. We juxtaposed DMTs against the high-degree trees used for secure memories [29, 57]) and showed that such approaches are not applicable to storage. Additionally, some prior works have decidedly side-stepped the issue of writes by either suppressing caches during updates or limiting analysis to read-heavy or read-only workloads [3, 55]. Given that storage workloads are often write-heavy (e.g., due to dedicated application-level read caches), we closely examine write-heavy workloads. Further, experiments in these prior works have been limited to small capacities (e.g., 16GB). We report experiments with an implementation on real block devices and file systems and evaluate a wider range of system settings.

Authenticated Data Structures. Hash trees have also been examined in the broader theoretical context of authenticated data structures—data structures that have strong integrity protections [23, 45, 58]. They become a central component of mobile and embedded device storage: dm-verity has played a pivotal role in providing verified boot for Android smartphones [1]. They have also been examined in the context of blockchains [15], certificate revocation systems [43], and provable data possession schemes [24, 26]. These works have highlighted the theoretical efficiency of using hash trees over other integrity data structures. Recent works have examined optimizations for blockchains, with known limitations (i.e., a priori knowledge of access patterns) [38]. Our work builds on these efforts by showing that, while efficient in theory, traditional static, balanced hash trees still incur substantial overheads in real cloud storage systems. This motivates our search for a more efficient and practical tree structure and ultimately the design of DMTs.

11 Conclusion

Merkle hash trees provide robust integrity protections over untrusted storage, but they severely degrade performance. We performed a comprehensive analysis of performance overheads, demonstrated the root cause, and designed an optimized tree structure called DMTs. DMTs demonstrate the power of integrity structures that exploit workload patterns to improve performance. Our code is open-sourced at https://github.com/MadSP-McDaniel/dmt.

Acknowledgments

This work was supported in part by the Semiconductor Research Corporation (SRC) and DARPA.

References

[1] Android. 2023. Implementing dm-verity. https://source.android.com/docs/security/features/verifiedboot/dm-verity. Last accessed: 2023-10-16.
[2] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. 2023. Nimble: Rollback Protection for Confidential Cloud Services. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). 193–208.
[3] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath Setty, et al. 2021. Fastver: Making data integrity a commodity. In Proceedings of the 2021 International Conference on Management of Data. 89–101.
[4] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A high concurrency key-value store with integrity. In Proceedings of the 2017 ACM International Conference on Management of Data. 251–266.
[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov
[6] Roberto Avanzi, Ionut Mihalcea, David Schall, Héctor Montaner, and Andreas Sandberg. 2022. Cryptographic Protection of Random Access Memory: How Inconspicuous can Hardening Against the most Powerful Adversaries be? Cryptology ePrint Archive (2022).
[7] Amazon AWS. 2023. Amazon Elastic Block Store. https://aws.amazon.com/ebs. Last accessed: 2023-05-16.
[8] Jens Axboe. 2024. fio - Flexible I/O Tester. https://fio.readthedocs.io/en/latest/. Accessed: 2024-09-12.
[9] Microsoft Azure. 2023. Microsoft Azure Managed Disks. https://learn.microsoft.com/en-us/azure/virtual-machines/managed-disks-overview. Last accessed: 2023-05-16.
[10] Microsoft Azure. 2024. Confidential VMs Overview. https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview. Accessed: 2024-08-19.
[11] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. IBM Systems journal 5, 2 (1966), 78–101.
[12] Felix Bohling, Tobias Mueller, Michael Eckel, and Jens Lindemann. 2020. Subverting Linux'integrity measurement architecture. In Proceedings of the 15th International Conference on Availability, Reliability and Security. 1–10.

[13] Milan Brož, Mikuláš Patočka, and Vashek Matyáš. 2018. Practical cryptographic data integrity protection with full disk encryption. In ICT Systems Security and Privacy Protection: 33rd IFIP TC 11 International Conference, SEC 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, September 18-20, 2018, Proceedings 33. Springer, 79–93.

[14] Quinn Burke, Ryan Sheatsley, Rachel King, Owen Hines, Michael Swift, and Patrick McDaniel. 2025. On scalable integrity checking for secure cloud disks. In 23rd USENIX Conference on File and Storage Technologies (FAST 25). 391–405.

[15] Vitalik Buterin. 2016. Ethereum: platform review. Opportunities and Challenges for Private and Consortium Blockchains 45 (2016).

[16] Anrin Chakraborti, Bhushan Jain, Jan Kasiak, Tao Zhang, Donald Porter, and Radu Sion. 2017. Dm-x: protecting volume-level integrity for cloud volumes and local block devices. In Proceedings of the 8th Asia-Pacific Workshop on Systems. 1–7.

[17] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In 2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[18] CK Chow. 1974. On optimization of storage hierarchies. IBM Journal of Research and Development 18, 3 (1974), 194–203.

[19] Google Cloud. 2023. Google Cloud Persistent Disks. https://cloud.google.com/persistent-disk. Last accessed: 2023-05-16.

[20] Google Cloud. 2024. Confidential VMs Overview. https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview. Accessed: 2024-08-19.

[21] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing. 143–154.

[22] Jonathan Corbet. 2010. Dynamic writeback throttling. https://lwn.net/Articles/405076/. Accessed: 2024-09-15.

[23] Scott A Crosby and Dan S Wallach. 2011. Authenticated dictionaries: Real-world costs and trade-offs. ACM Transactions on Information and System Security (TISSEC) 14, 2 (2011), 1–30.

[24] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. 2016. Efficient sparse merkle trees: Caching strategies and secure (non-) membership proofs. In Secure IT Systems: 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016. Proceedings 21. Springer, 199–215.

[25] Haowen Dong, Chao Zhang, Guoliang Li, and Huanchen Zhang. 2024. Cloud-Native Databases: A Survey. IEEE Transactions on Knowledge and Data Engineering (2024).

[26] C Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. 2015. Dynamic provable data possession. ACM Transactions on Information and System Security (TISSEC) 17, 4 (2015), 1–29.

[27] Alberto Faria, Ricardo Macedo, José Pereira, and João Paulo. 2021. BDUS: implementing block devices in user space. In Proceedings of the 14th ACM International Conference on Systems and Storage. ACM, Haifa Israel, 1–11. doi:10.1145/3456727.3463768

[28] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the {PENGLAI} Enclave. In 15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21). 275–294.

[29] Alexander Freij, Huiyang Zhou, and Yan Solihin. 2021. Bonsai merkle forests: Efficiently achieving crash consistency in secure persistent memory. In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture. 1227–1240.

[30] Anna Galanou, Khushboo Bindlish, Luca Preibsch, Yvonne-Anne Pignolet, Christof Fetzer, and Rüdiger Kapitza. 2023. Trustworthy confidential virtual machines for the masses. In Proceedings of the 24th International Middleware Conference. 316–328.

[31] Blaise Gassend, G Edward Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. 2003. Caches and hash trees for efficient memory integrity verification. In The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. IEEE, 295–306.

[32] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In International Conference on Machine Learning. PMLR, 1919–1928.

[33] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 969–986.

[34] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. Proceedings of the IRE 40, 9 (1952), 1098–1101.

[35] Brian Johannesmeyer, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2024. Practical {Data-Only} Attack Generation. In 33rd USENIX Security Symposium (USENIX Security 24). 1401–1418.

[36] Louiza Khati, Nicky Mouha, and Damien Vergnaud. 2017. Full disk encryption: bridging theory and practice. In Topics in Cryptology–CT-RSA 2017: The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings. Springer, 241–257.
[37] Anil Kurmus, Nikolas Ioannou, Matthias Neugschwandtner, Nikolaos Papandreou, and Thomas Parnell. 2017. From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks. In 11th USENIX Workshop on Offensive Technologies (WOOT 17).
[38] Oleksandr Kuznetsov, Dzianis Kanonik, Alex Rusnak, Anton Yezhov, Oleksandr Domin, and Kateryna Kuznetsova. 2024. Adaptive Merkle trees for enhanced blockchain scalability. Internet of Things 27 (2024), 101315.
[39] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data. 121–132.
[40] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. 2023. An in-depth comparative analysis of cloud block storage workloads: Findings and implications. ACM Transactions on Storage 19, 2 (2023), 1–32.
[41] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. {ROTE}: Rollback protection for trusted execution. In 26th USENIX Security Symposium (USENIX Security 17). 1289–1306.
[42] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '13. ACM Press, Tel-Aviv, Israel. doi:10.1145/2487726.2488368
[43] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. 2015. {CONIKS}: Bringing key transparency to end users. In 24th USENIX Security Symposium (USENIX Security 15). 383–398.
[44] Ralph C Merkle. 1989. A certified digital signature. In Conference on the Theory and Application of Cryptology. Springer, 218–238.
[45] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated data structures, generically. ACM SIGPLAN Notices 49, 1 (2014), 411–423.
[46] Alistair Moffat. 2019. Huffman coding. ACM Computing Surveys (CSUR) 52, 4 (2019), 1–35.
[47] Moni Naor and Kobbi Nissim. 2000. Certificate revocation and certificate update. IEEE Journal on selected areas in communications 18, 4 (2000), 561–570.
[48] NIST. 2023. Report on the Block Cipher Modes of Operation in the NIST SP 800-38 Series. NIST. Retrieved 2023-10-15 from https://nvlpubs.nist.gov/nistpubs/ir/2023/NIST.IR.8459.ipd.pdf
[49] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. 2006. vTPM: virtualizing the trusted platform module. In Proc. 15th Conf. on USENIX Security Symposium. 305–320.
[50] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. 2020. SGX-LKL: Securing the Host OS Interface for Trusted Execution. arXiv:1908.11143 [cs] (Jan. 2020). http://arxiv.org/abs/1908.11143
[51] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. In 2018 IEEE Symposium on Security and Privacy (SP). IEEE, San Francisco, CA, 264–278. doi:10.1109/SP.2018.00025
[52] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007). IEEE, 183–196.
[53] Amazon Web Services. 2024. AMD SEV-SNP in Amazon EC2. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html. Accessed: 2024-08-19.
[54] Tomer Shanny and Adam Morrison. 2022. Occualizer: Optimistic concurrent search trees from sequential code. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). 321–337.
[55] Rohit Sinha and Mihai Christodorescu. 2018. Veritasdb: High throughput key-value store with integrity. Cryptology ePrint Archive (2018).
[56] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting binary search trees. Journal of the ACM (JACM) 32, 3 (1985), 652–686.
[57] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. 665–678.
[58] Roberto Tamassia. 2003. Authenticated data structures. In Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11. Springer, 2–5.
[59] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. login Usenix Mag. 41 (2016).
[60] Theodore Ts'o. 2018. File System-level Integrity Protection. (2018).

[61] Weijie Wang, Yujie Lu, Charalampos Papamanthou, and Fan Zhang. 2023. The Locality of Memory Checking. Cryptology ePrint Archive (2023).

[62] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving cost, performance, and security of memory encryption and authentication. ACM SIGARCH Computer Architecture News 34, 2 (2006), 179–190.

[63] Yue Yang and Jianwen Zhu. 2016. Write skew and zipf distribution: Evidence and implications. ACM transactions on Storage (TOS) 12, 4 (2016), 1–19.

Additional Results

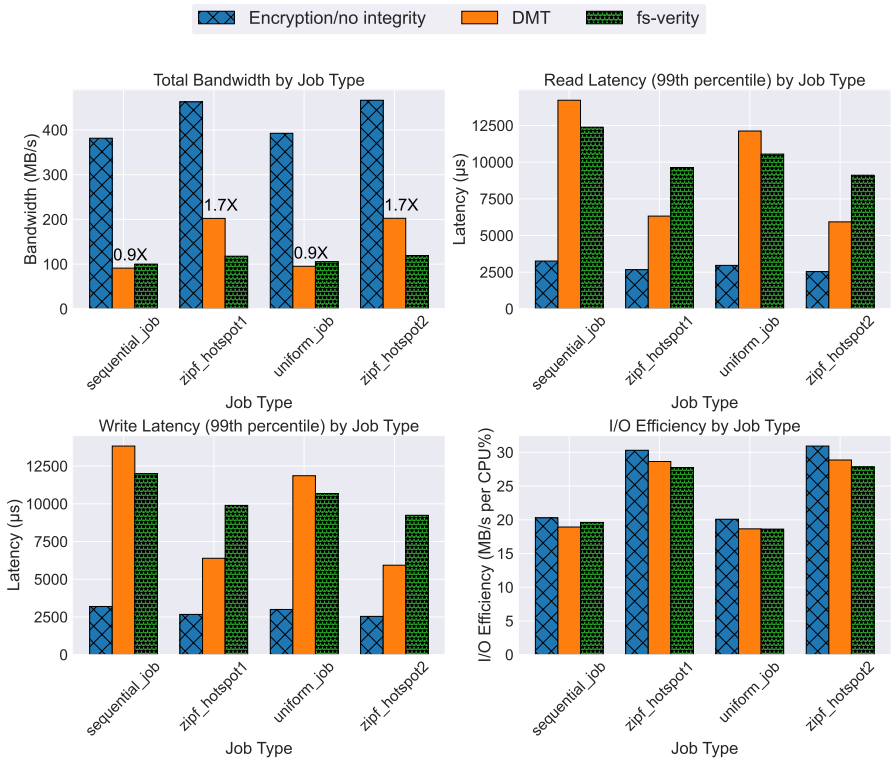


Fig. 24. This graph demonstrates per-file specialization when the access patterns are run in sequence on a single file. As we observed at the block layer, DMTs can quickly adapt to changing patterns, delivering a 1.7× speedup over fs-verity during the Zipfian phases and incurring a 10% cost during the sequential and uniform random phases.

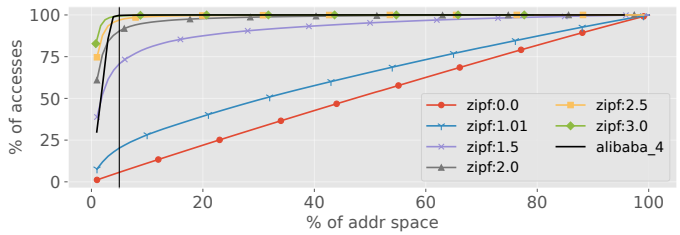


Fig. 25. Workload distributions.

Received 1 August 2025; revised 1 August 2025; accepted 1 August 2025