2025 IEEE Symposium on Security and Privacy (S&P)

Efficient Storage Integrity in Adversarial Settings

Quinn Burke*, Ryan Sheatsley, Yohan Beugin, Eric Pauley,
Owen Hines, Michael Swift, Patrick McDaniel
University of Wisconsin-Madison

***Abstract*—Storage integrity is essential to systems and applications that use untrusted storage (e.g., public clouds, end-user devices). However, known methods for achieving storage integrity either suffer from high (and often prohibitive) overheads or provide weak integrity guarantees. In this work, we demonstrate a hybrid approach to storage integrity that simultaneously reduces overhead while providing strong integrity guarantees. Our system, partially asynchronous integrity checking (PAC), allows disk write commitments to be deferred while still providing guarantees around read integrity. PAC delivers a $5.5\times$ throughput and latency improvement over the state of the art, and 85% of the throughput achieved by non-integrity-assuring approaches. In this way, we show that untrusted storage can be used for integrity-critical workloads without meaningfully sacrificing performance.**

1. Introduction

Storage integrity is essential to systems and applications that use untrusted storage (e.g., public clouds, end-user devices). While robust systems for ensuring the integrity of the compute environment have become commonplace on cloud providers [1], [2], [3] and mobile devices [4], the storage [5], [6], [7] connected to these environments do not guarantee that returned data is authentic (i.e., was actually written by the application), transactionally consistent (i.e., the storage state is globally correct with respect to storage operations), and fresh (i.e., has not been rolled back to a previous state). When these properties are compromised, an adversarial storage device (be it a malicious cloud provider, backdoored physical device, or low-level system malware) can replay financial transactions, undo deployment of vulnerability mitigations, and otherwise compromise the integrity of the system as a whole [8], [9], [10], [11].

While local integrity is readily achieved using authenticated encryption (e.g., AES/AEAD), global transactional consistency and freshness on untrusted storage requires the use of globally-consistent data structures. Researchers have converged on Merkle hash trees—which recursively hash the data stored in blocks to one single checksum—to track the globally-consistent state of a storage device [4], [12], [13], [14], [15], [16]. When tracked using trusted hardware [17], [18], [19], [20] (e.g., secure counters offered by the CPU), this approach provides transactional consistency and prevents an adversary from rolling the storage system back to a previous state. Such a solution has also been adopted by practitioners, and is available in Linux as dm-verity [4].

*Corresponding author (email: qkb@cs.wisc.edu)

Unfortunately, using hash trees to achieve global consistency comes at a dramatic cost to performance, particularly as storage sizes scale [21]. Prior works have approached this problem in two ways: (a) by optimizing Merkle tree implementations using caches [13] and new tree implementations [20], [22], [23], and (b) by relaxing the storage threat model to no longer provide strong protections against adversarial data rollbacks and data manipulation [24], [25]. Under these relaxed threat models, techniques defer the durable storage of written data *and* the verification of disk reads, under the premise that these operations incur unacceptable performance overhead. However, this means that an application can proceed after reading incorrect data, taking actions that compromise overall system integrity. In effect, the existing space of disk integrity techniques presents a seemingly-fundamental tradeoff between strong security guarantees and performance.

In this work, we examine this security-performance tradeoff. Current techniques exist on two ends of a spectrum: traditional *synchronous* approaches persist and verify all writes and reads in order and as they are performed, and *asynchronous* approaches defer each of these in the interest of performance. We hypothesize that a *hybrid* of asynchronous writes and coordinated synchronous reads can virtually eliminate the overhead of transactional disk integrity, while still providing the same security guarantees as synchronous approaches. Our approach, partially asynchronous integrity checking (PAC), ensures complete transactional integrity of disk contents at runtime, and provides rollback protection of fully-committed writes under adversarial fault injection at the storage layer (the same guarantees offered by conventional disk write semantics in a benign setting).

We evaluate PAC in two stages, beginning with a formal security analysis. We establish two guarantees: (1) a *read guarantee*, wherein reads always reflects the most recent acknowledged write at the time data is returned, and (2) a *write guarantee*, which ensures that the sealed and durable Merkle root always reflects the most recent FSYNC call. We provide a set of proofs to establish these guarantees under PAC's hybrid approach.

We next implement PAC as a Linux block device driver and evaluate its performance, measuring its overhead, scalability, and memory/storage trade-off. PAC delivers $> 85\%$

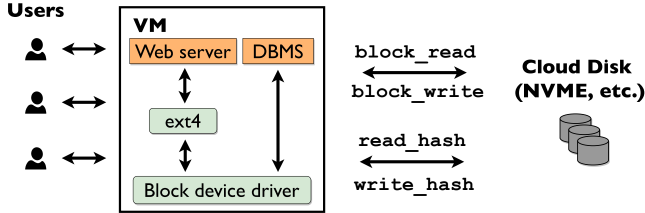


Figure 1: We consider an IaaS deployment model where an application runs inside a guest VM and stores data on a fast, local NVMe disk.

of the performance of block-level AEAD while *additionally* providing global transactional integrity. It scales to large storage devices, with throughputs up to $5.5\times$ that of prior hash tree designs. Finally, PAC achieves these with minimal additional memory requirements. In these ways, PAC enables the strongest storage integrity guarantees at minimal cost.

Software systems rely on the integrity of both computation and storage to ensure end-to-end outcomes. Coupled with advances in trusted execution technology [14], [16], [17], our work shows that the transactional integrity and freshness of underlying storage can be achieved at minimal overhead. We anticipate that PAC will enable the practical use of trusted storage in an expanded space of performance- and security-critical software systems. Our code is plug-and-play into standard Linux systems and open-sourced at https://github.com/MadSP-McDaniel/pac.

2. Background

Cloud Disks. Block storage is a backbone of modern public cloud infrastructure [5], [6], [7]. While there are various deployment models for cloud applications and storage, we consider a standard Infrastructure-as-a-Service (IaaS) deployment where an application runs inside a guest VM and reads and writes to a fast, local NVMe disk attached to the VM (Figure 1). The application may be end-user facing (e.g., a database or web server) or the last hop in a networked storage system (e.g., a file server for other cloud-based applications).

Merkle Hash Trees. Merkle hash trees are the standard method to protect the integrity (notably, the freshness) of storage—largely due to their proven theoretical efficiency [12], [13], [14], [16], [17]. They play a pivotal role in ensuring boot disk integrity with Linux *dm-verity* [4] and other emerging cloud runtimes. For storage, they are implemented as a custom block device driver that wraps a lower-level driver. The driver intercepts block I/O requests and implements the hash tree logic.

As shown in Figure 2, a Merkle hash tree is a balanced binary tree, with each node in the tree containing a hash value. A leaf node contains the MAC of a data block (and a cipher IV when encrypting data), and an internal node contains the SHA256 hash of the concatenation of the hashes of its two children. Internal node hashes are iteratively computed from leaf to root along the authentication path. The root hash *authenticates* (i.e., asserts the correctness of) the current disk contents and is stored in a secure and non-volatile memory region that is inaccessible to an attacker [19], [20].

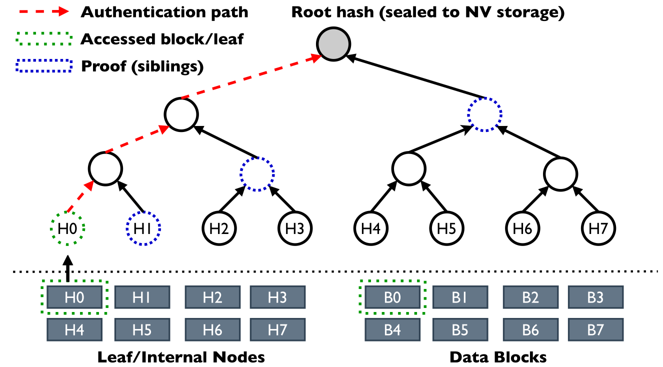


Figure 2: A Merkle hash tree protects the integrity of data read from/written to a storage device.

There are two basic operations on a hash tree: *verification* and *update*. When the driver receives a block read call, the (encrypted) block data, MAC, and cipher IV are first fetched from disk. The MAC is checked for consistency against the fetched data. Next, it is verified against the root hash by recursively fetching, concatenating, and computing hashes up to the root. If the computed hash matches the known root, verification succeeds. When a block is written, a new MAC is computed (`H0`) and the hash tree is updated (`H0` ancestors) similarly. The new root hash is then saved to the secure non-volatile location. To prevent rollback attacks, the root can be *sealed*—i.e., bound to a tamper-resistant counter, signed, and saved to non-volatile storage [26], [27], [28].

3. Security Model

Trust Model. We consider a standard IaaS deployment model (see Figure 1) where applications run inside guest VMs and read and write data to fast local NVMe disks. In this setting, we assume that all VM contents (code and data in memory) and the Merkle root are trusted and protected by hardware-based memory access controls. This can be ensured with confidential virtual machine technology such as AMD SEV-SNP [1], [2], [3]. We assume the storage devices and the hypervisor that manages access to storage (virtual or physical) are untrusted. The trusted and untrusted components have a simple block read/write interface (see Figure 3).

Threat Model. We consider a privileged attacker who has access to the hypervisor or storage backbone in a public cloud data center [17], [29]. This could be a malicious co-tenant with escalated privilege or a malicious cloud administrator. We assume the attacker attempts to execute *data-only attacks*, attacks that are based on maliciously crafted data being returned to the block device driver by a malicious storage device; they present a significant threat to modern cloud applications [8], [10], [11], [30], [31]. The attacker has

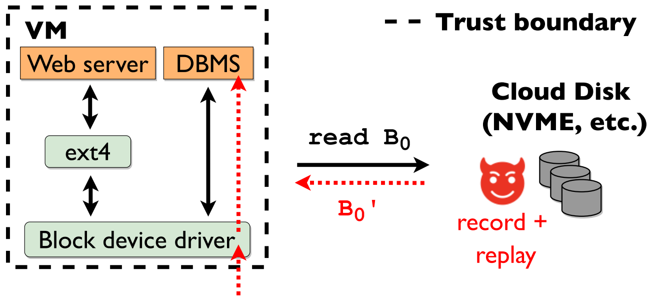


Figure 3: We assume that VM memory contents are trusted and cloud disks are untrusted; VM memory can be protected with trusted execution primitives [1].

the ability to access, corrupt, swap, drop, record, inject, or replay any data across the storage interface. We also assume that the attacker has control over vCPU scheduling and can thus arbitrarily suspend or delay vCPU execution [32], [33]. Note that we consider denial-of-service attacks out-of-scope, but delaying program execution can also affect integrity guarantees; we defer further discussion to Section 5.

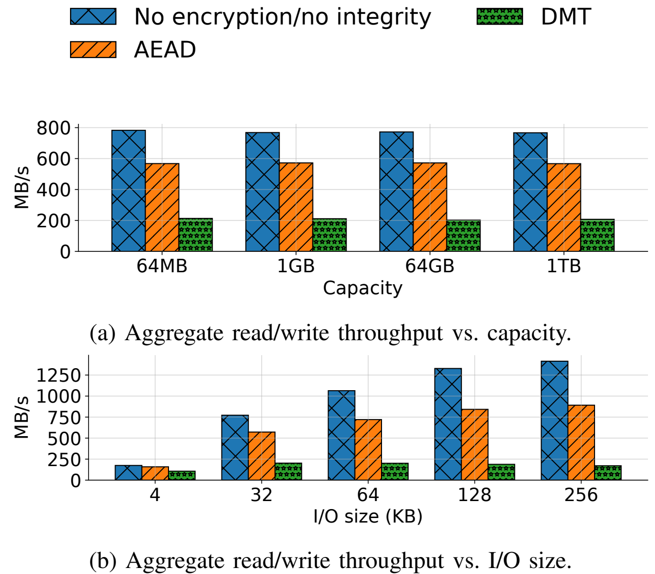
We focus particularly on replay attacks; the other attacks are protected via keyed hashes (block MACs). Replay attacks could manifest in the OS loader reading old, vulnerable versions of binaries, or file systems making incorrect access control decisions, among other attacks. Replay attacks cause externally visible effects (i.e., affect client decision-making) which are infeasible to undo and thus must be prevented.

Security Requirements. To prevent unauthorized access, the confidentiality of data can be ensured with standard AES-GCM encryption. Ensuring integrity requires guaranteeing authenticity (i.e., that some data originates from a trusted party) and freshness (i.e., that read data is the most recent version written by a trusted party). The MACs output by AES-GCM provide authenticity. Storing MACs in a secure location (after each update) then ensures freshness. However, keeping all MACs in secure memory is infeasible. A Merkle hash tree thus provides an efficient data structure to facilitate verifications and updates; it only requires keeping the Merkle root in secure memory [13], [24], [34], [35].

4. Problem Statement

Merkle hash trees have been widely celebrated because of their proven theoretical efficiency [21]. Asymptotically, the $O(\log n)$ traversal costs have translated into low performance overheads compared to other data structures that ensure integrity. However, recent works have challenged this assertion. For example, it was shown that tree traversal costs can manifest in more than 10× and 3× slowdowns in the context of secure memories [20], [36] and cloud storage [23], [24], [25], respectively.

Reducing performance overheads requires fundamentally rethinking the hash tree design at both the data structure- and algorithm-level. However, there is a tradeoff between security and performance. This section shows that prior works have failed to deliver a solution that provides both and motivates our search for a better approach.



(a) Aggregate read/write throughput vs. capacity.

(b) Aggregate read/write throughput vs. I/O size.

Figure 4: Aggregate read/write throughput vs. capacity and I/O size. Experiment parameters: Workload: Zipf(2.5), Read ratio: 1%, Cache size: 10%, Capacity: 1 TB, I/O size: 32 KB, Threads: 1, I/O depth: 32.

Data Structure Optimizations. We begin with a motivating experiment in Figure 4 examining the state-of-the-art hash tree design Dynamic Merkle Trees (DMTs) [23]. DMTs were recently proposed as alternative to the balanced, binary hash trees traditionally used to secure disks (e.g., through Linux `dm-verity`). They are based on splay trees, which are dynamic, unbalanced tree structures that self-adjust on the fly based on observed workload patterns. DMTs were shown to closely approximate an optimal tree structure. We defer implementation details to Section 7, but note that like prior works, the hash tree is implemented in a block device driver that intercepts block I/Os and performs verifications and updates when synchronously reading and writing data.

Figure 4 shows how the performance of DMTs changes with respect to disk capacity and I/O size. Figure 4a shows that DMTs deliver 200 MB/s throughput across capacities representative of very small and very large disks. However, at 200 MB/s DMTs still observe a 67% throughput loss relative to the AEAD (Encryption/no integrity) baseline. Note that this baseline represents an encrypted disk with no integrity protection, while DMT represents an encrypted disk that is protected by a Merkle tree. Further, disk throughput typically grows with larger I/O sizes, but Figure 4b shows that DMT performance does not scale with I/O size: performance plateaus and losses are nearly 80% with 256 KB I/Os.

Caching hashes in secure memory (i.e., in a protected memory region) is also a standard optimization [13], [24]. Caching reduces I/O costs associated with fetching hashes

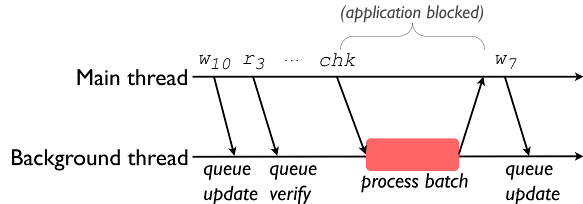
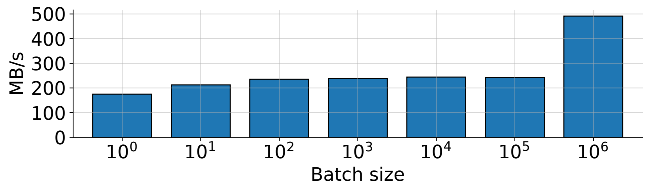
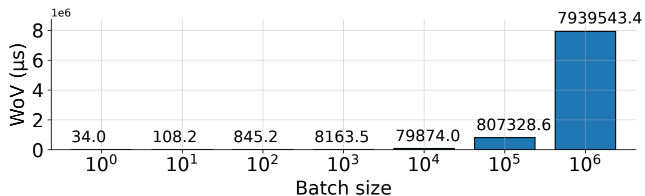


Figure 5: Batched processing (Note: chk=checkpoint step).



(a) Aggregate read/write throughput vs.batch size.



(b) Window of vulnerability (WoV) length vs. batch size.

Figure 6: Meaningful performance gains are only realized with larger batch sizes, but large batch sizes introduce long windows of vulnerability.

during a verification or update and prompts early exits during verification (as the hashes were previously authenticated before being placed in the cache). We found that with a small cache, hit rates are consistently high (99.9%). The implications of this are twofold. Verification costs are thus largely hidden—particularly under read-heavy workloads. However, these graphs examine write-heavy workloads, which reflect observations of real-world block storage systems [37]. For write-heavy workloads, hash caching clearly only helps to an extent, and new optimizations are needed.

Algorithm Optimizations. Given the above, the most immediate question is how we can leverage algorithm-level optimizations to improve performance. A natural starting point is to use *asynchronous execution*, which is a building block of modern storage systems [38], [39]. Intuitively, asynchronous execution means that hash tree operations are decoupled from data operations and have minimal performance impact on the critical path. This can be achieved through means such as batched processing.

In a batching scheme, whenever data is read or written, a request object for a verification or update is created and placed on a queue. Whenever the queue hits the specified $batch_size$ limit, the application thread blocks (i.e., *checkpoints*) until all pending verifications and updates are complete. This can improve performance by removing hash tree operations from the critical path entirely. Costs can then be amortized by accumulating and processing large batches of requests at a more opportune time. An overview of this process is shown in Figure 5.

Recent works have begun to embrace this idea to optimize integrity checks in real-world key-value stores [24], [25]. However, batching introduces a significant security risk. Notably, reads always return unverified data up the call stack, and thus data-only attacks are not immediately detectable. This directly violates integrity (freshness) guarantees.

To quantify the extent to which these vulnerabilities can be practically exploited, Figure 6 characterizes the window of vulnerability (WoV) introduced by batching. The window of vulnerability describes the time between when data was returned to the caller and when it was actually verified. There are two key observations. First, Figure 6a shows that throughput improvements are only observed with large batch sizes ($> 10^5$). Moreover, Figure 6b shows that the mean window of vulnerability grows with a power law w.r.t. batch size. At a batch size of 10^5 blocks, the mean window of vulnerability is 800 ms, while at a batch size of 10^6 it is 8 seconds. For reference, FastVer [24] uses a batch size of 4 M requests. There is a clear security-performance tradeoff here: batching only helps with large batch sizes, but large batch sizes introduce long windows of vulnerability.

Our Objective. These prior works have led the community to an understanding that in order to achieve storage integrity one must either give up performance or weaken security guarantees. We posit that this tradeoff can be avoided. This leads us to the question: *Can we capture the performance advantages of asynchrony without sacrificing integrity guarantees?* In the following, we show that it is indeed possible.

5. Partially Asynchronous Integrity Checking (PAC)

This section shows how we can use concurrency to maintain the security guarantees of a synchronous hash tree design while exploiting the performance advantages of an asynchronous hash tree design.

Security Guarantees. To provide integrity on untrusted storage, we design our system around two security guarantees:

1) **Read Guarantee:** The integrity of read data is always verified before the data is returned to applications; reads are never speculative. This ensures that applications will always receive correct and fresh data and closes the window of vulnerability entirely.
2) **Write Guarantee:** Hash tree updates are executed asynchronously, but the main thread coordinates with the background thread to ensure liveness of hash tree updates (and state updates). This prevents rollback attacks introduced by asynchrony. The sealed Merkle root always accurately reflects the latest FSYNC call.

TABLE 1: Comparison of security guarantees and core mechanisms of different approaches to integrity checking. Broadly, whether each approach provides runtime consistency (integrity) checks and/or rollback protection depends on when updates and verifications are executed. For example, Batching (i.e., DMT + Batching) does not provide runtime consistency because verifications/updates are asynchronous (and thus replay detection is deferred), and achieves good performance. PAC (i.e., DMT + PAC) provides runtime consistency via synchronous verifications and rollback protection, while leveraging asynchronous updates to improve performance.

	AEAD	dm-verity [4][1]	DMT [23]	DMT + Batching	DMT + PAC
Runtime consistency	○	●	●	◐[2]	●
Rollback protection	○	○	●	●	●
Updates (writes)	*Sync*	*Sync*	*Sync*	*Async*	*Async*
Verifications (reads)	*Sync*	*Sync*	*Sync*	*Async*	*Sync*

[1] This denotes dm-verity variants that support writes. [2] Batching has deferred runtime consistency checking.

5.1. PAC Overview

We introduce a partially asynchronous integrity checking (PAC) method. PAC exploits asynchronous execution through concurrency: it uses a background thread that assists in the verification and update process. The design centers around a shared queue that resides in secure memory and holds pending update requests. It is driven by two key mechanisms: *synchronous verifications* and *flush-consistent asynchronous updates*. Note that these mechanisms are transparent to applications and handled entirely within the block device driver. An overview is shown in Figure 7 and comparison to prior approaches is shown in Table 1.

Synchronous verification means that data is always verified by the main thread before being returned to the caller; these costs are minimized via the hash cache. In contrast, update requests are queued by the main thread and executed asynchronously by the background thread. Updates are executed one at a time. The background thread polls the queue at a configurable rate; we use a thread sleep to enforce this. Flush-consistency means that when the driver receives a device flush call (triggered via an FSYNC system call), the main thread blocks until all pending updates are complete. As device flushes are used to provide durability guarantees to applications (i.e., that data is in fact persistent and not residing in the OS page cache or disk hardware caches), this ensures that the Merkle root always accurately reflects what disk contents the application believes to be durable.

The result is that: (1) reads are never speculative, and (2) the main thread can largely proceed with other useful work (e.g., other file system or application logic) while the background thread processes update requests. Realizing this requires several additional support structures.

5.2. Write Path

Block Writes. When the driver receives a write call, the block data is encrypted and a new MAC is generated. This happens in the context of the main thread. A request object is then created and appended to the shared queue. The object contains the Merkle tree node object, new MAC, and new cipher IV. The actual block data is immediately written out to disk (but not flushed through the disk cache, i.e., made durable). This process is shown in Figure 8.

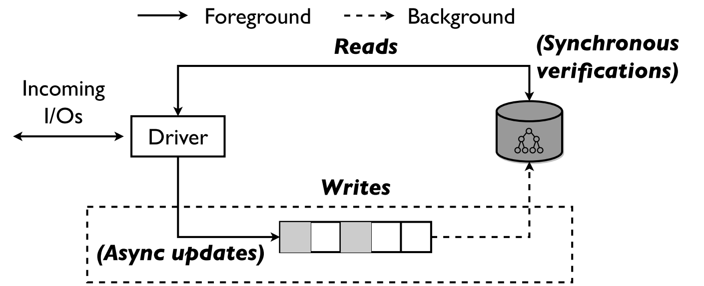


Figure 7: PAC overview. Verifications are executed synchronously by the main thread, while updates are queued by the main thread and executed asynchronously by the background thread. We refer to Figure 8 for a timeline diagram of further technical details.

For example, when the driver receives a write for block 10 (w_{10}), the main thread writes the data to disk, queues an update request, and returns execution to user space while the background thread processes the update. Later, the background thread will raise an alert if the update failed.

During the write call, PAC must also handle the scenarios where the incoming write is to a block that still has a pending update, or where the queue is full.

Overriding Updates. If during the write call there is a pending update request for the same block, the main thread overrides the pending request with the incoming one. Thus, only the latest update to a block is tracked in the queue and executed by the background thread. An alternative would be to queue each update request individually, which would require additional memory for the queue and additional compute resources to complete updates. PAC avoids the additional memory requirement and doing wasted work on earlier update requests which were effectively voided by more recent updates.

Rate Limiting Updates. In a real deployment, the size of the queue will be capped to some percentage of the system memory capacity to more efficiently allocate resources to

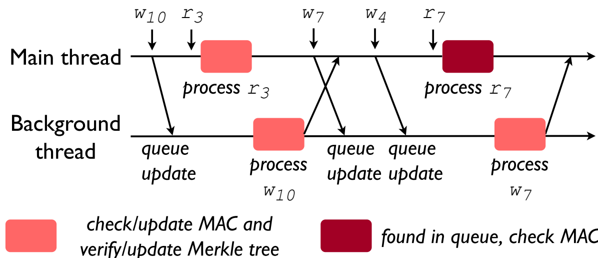


Figure 8: In PAC, verifications are executed synchronously to ensure freshness on reads, while updates are executed asynchronously in a background thread to exploit parallelism.

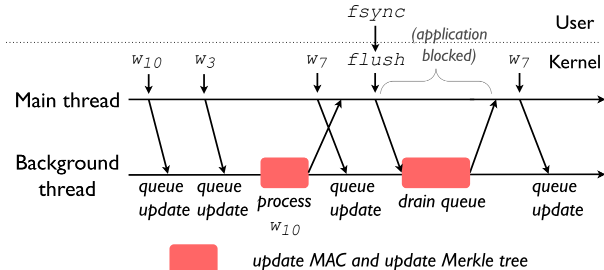


Figure 9: In PAC, device flush commands (triggered by an FSYNC system call) force the main thread to block while all pending update requests complete and a state update is committed (i.e., Merkle root is sealed). This ensures liveness of the background thread and rollback protection.

running applications. To handle a finite queue capacity, PAC implements a rate limiting mechanism. During a write call, if the queue is already full, the main thread blocks until at least one free queue slot opens up. At the same time, the background thread enters a draining state where it ignores the sleep timer and executes updates as quickly as possible. Thus, when the queue becomes full, performance falls back to (in essence) doing updates synchronously, because the main thread will block while an update completes and opens a queue slot. The queue size and high/low thresholds that control rate limiting are configurable by a system administrator and can be changed dynamically at runtime.

5.3. Read Path

Block Reads. When the driver receives a read call, the main thread first reads the data from disk. It then checks the queue for the latest version of the block MAC. If there is a pending update request, it verifies that the MAC (in the queued request) is consistent with the fetched data then decrypts the data. Otherwise, it proceeds with the normal read/verify procedure: it fetches the node from the hash cache (or disk), decrypts the data with it, then verifies the MAC in the hash tree. This process is shown in Figure 8.

For example, when the driver receives a read call for block 3 (r_3), there is no pending update request for block 3 in the queue, so the main thread proceeds with the normal read/verify procedure. However, when the driver later receives a read call for block 7 (r_7), there is still a pending update request for block 7, so the main thread pulls the MAC directly from the queued request. If the data read from disk is consistent with this MAC, verification is complete and the data is returned to the caller—there is no need to traverse the hash tree. Thus, the queue can also prompt early exits during verification like the hash cache can.

5.4. Rollback Protection

While our focus is not rollback attacks, executing updates asynchronously complicates rollback guarantees. This section describes how PAC mitigates rollback vulnerabilities introduced by using concurrency.

Rollback Challenges. Standard methods to provide rollback protection and recovery are known. For example, sealing the Merkle root (the disk *state*) with a tamper-resistant counter can prevent an attacker from using a crash to “reset” the Merkle root and replay an old version of the entire disk to applications [26], [27], [28]. However, privileged attackers still have control over vCPU scheduling and other operating system tasks [32], [33]. If hash tree updates (and state updates) are handled by a separate thread, the attacker can arbitrarily delay the update thread to control the degree to which state updates are sealed. In the worst case, they can prevent the sealed state from ever advancing past the initial state, and after a crash the system would simply recover to the initial state. The problem is that there must be a *checkpointing* mechanism through which the main thread can validate that hash tree updates are in fact progressing in the background (i.e., liveness guarantee).

Coordinated State Updates. To design the checkpointing mechanism, we must first consider how to securely manage *state*. For storage, state consists of the set of disk blocks, but is represented by the Merkle root. Rollback protection can be ensured by sealing the state with a tamper-resistant counter. However, storage systems are presented an additional challenge with regards to managing state: while in theory the state is a small 32 B Merkle root, updating state is effectively a two-step process: all hash tree updates must first be reflected in the Merkle root, and the final Merkle root must then be sealed. We refer to the former as the *preparation* phase and the latter as the *sealing* phase.

In PAC, both threads coordinate in performing state updates. The background thread handles preparation. Sealing is invoked at the discretion of the application and can be handled in either the main thread or background thread. However, the main thread checkpoints by blocking in the device flush function until both preparation and sealing are complete. This is shown in Figure 9. Note that without blocking, the main thread would have no guarantee that the Merkle root was updated and sealed properly. Thus, the checkpoint mechanism guarantees both liveness and rollback

protection: any subsequent integrity check against the Merkle root must either be fresh or fail.

Checkpointing during a flush (triggered by an `FSYNC` system call) is desirable because it is a natural synchronization point upon which crash-consistency protocols are built; it signifies a clear intent to commit a state update. From an application point of view, data (whether buffered in memory or written out to disk) is not considered persistent until an fsync is called and a flush command is sent to the device. Upon a crash, updates queued since the last flush would indeed be lost, but alongside the dirty (buffered) data as well. We note that ideally applications should flush data in a timely fashion. However, reducing (expensive) flushes is common practice when tuning application configurations. As an additional measure of protection, flushes could be opportunistically (or eagerly) induced in the driver.

6. Security Analysis

In this section, we show that `PAC` provides strong integrity guarantees (e.g., prevent replay and rollback attacks) during normal operation and following system crashes.

Theorem 1. *In the absence of system crashes,* `PAC` *always returns the most recent version of data to the caller on reads or the integrity check fails. (Read guarantee)*

Proof. Let R_i denote the Merkle root at version i. Without loss of generality, consider the block at address j. We denote with B and $h(B)$ the current data stored in the block at address j and its hash, respectively. Now, assume that a write operation updated the block at address j to the value B'. We define the most *recent* data (B' here) to be the version that reflects the latest write issued to the driver (but not necessarily flushed through disk caches, i.e., durable).

When the block was updated, the data is sent to be written out to disk and an update request was placed on the queue. Upon the read request, one of two scenarios will occur. Either the update will have completed, in which case there will not be a pending update in the queue and the new block hash $h(B')$ would be reflected in a new Merkle root R_{i+1}. In this case, the driver will read the updated block B' from disk, fetch $h(B')$ from disk (or the hash cache), proceed with the normal recursive verification process, and return B' if it succeeds or an integrity check fault if not. Otherwise, if the update had not completed, then there will be a pending update request in the queue and the driver will read the updated block B' from disk, check the consistency of the block data with the queued hash $h(B')$, then return B' if it succeeds or an integrity check fault if it not.

The most recent version of the data B' is always reflected in the Merkle tree or in the queue. Thus, for an attacker to successfully replay data they would have to present a value B^* and hash $h(B^*)$ that either passes the verification check against the new Merkle root R_{i+1} or the queued hash $h(B')$. Because of the collision resistance property of the cryptographic hash function used, the adversary is unable to find B^* such that, $h(B^*) = h(B')$. Further, hardware-based memory access controls (recall our trust model in Section 3) ensure that the Merkle root is sealed and that it can not be modified by the attacker to facilitate the attack. Thus, the adversary could not provide a block B^* that would be verified against stored hash values without triggering an integrity check fault. $\square$

Theorem 2. *After a system crash,* `PAC` *always recovers the most recent state perceived as durable by the application or the integrity check fails. (Write guarantee)*

Proof. Let S_i denote the disk state (set of all block data) at version i and R_i denote the corresponding Merkle root that is sealed during a `FSYNC` call to commit (make durable) this disk state S_i. Assume a crash occurred, where the most recent sealed root R_k is returned to the system during recovery. An attacker would not be able to produce a set of data $S_{k*} \neq S_k$ that would hash to the protected and sealed root R_k because of collision resistance of the hash function. Thus, the recovery process will always return data of the most recent state made durable or the integrity verification process will fail. $\square$

7. Performance Evaluation

In this section we evaluate the performance of `PAC` across several realistic workloads. We compare `PAC` with state-of-the-art hash trees DMTs [23] [1] and report results against two non-integrity verified baselines: No encryption/no integrity and AEAD (Encryption/no integrity). Our evaluation focuses on the following research questions:

- **RQ1**. What overheads does `PAC` incur and under what conditions?
- **RQ2**. Is `PAC` scalable (i.e., broadly maintains a stable performance guarantee)?
- **RQ3**. What memory and storage costs are associated with `PAC`?

7.1. Experimental Setup

Implementation. We implement the hash trees in 5k lines of C++. We use BDUS to implement a custom block device driver that wraps a lower-level driver [40]. BDUS has a kernel module that exposes block layer hooks to user space [2]. The three functions of interest are `read()`, `write()`, and `flush()` which are invoked by the kernel whenever a block is read, a block is written, or an `FSYNC` is called by the application. Our basic data unit aligns with the disk I/O size (4 KB blocks) [15], [43], [44]. Note that best-known methods have shown that sealing can be done within single-digit milliseconds for small state sizes (< 1 KB) [28], [45]. We simulate a state update by putting the main thread to

1. We note that `PAC` is a general approach and the underlying data structure can be swapped out for alternatives (e.g., list-based ones).

2. We note that other frameworks could be used to implement `PAC`, including pure kernel, pure user space, or hybrid approaches. There are inherent trade-offs to each (e.g., less isolation and larger attack surface in kernel space); this is an orthogonal issue that is an active area of research [41], [42].

TABLE 2: Key experimental parameters.

Parameter	Description
Capacity	Usable capacity for data blocks
Cache size	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
Background rate	Polling rate of the background thread
FSYNC period	Duration between application flush calls

sleep for 5 ms at the end of the device flush function. A parameter overview is shown in Table 2.

Like prior works, we ensure authenticated encryption with AES-GCM [20], [34]. We use a 128-bit encryption key for blocks. The MACs produced during encryption are used as the leaves in the tree. For internal nodes, we compute 256-bit hashes using SHA-256 with a 256-bit key.

Testbed. We perform all experiments on a cloud server equipped with a 48-core 2.8 GHz AMD EPYC 7402P processor, 128 GB memory, and two local NVMe SSDs (one for data, one for metadata). Note that where metadata resided did not significantly impact results; small hash caches tend to be very efficient. We reinitialize hash trees between each experiment and use a standard LRU cache replacement policy. For DMTs and consistent with prior evaluations, we set the splay window flag $w = True$ and splay probability $p = 0.01$, which means that splaying only occurs 1% of the time. For PAC, we set the queue size to 1024 and the low watermark threshold for rate limiting to 0.75.

Workload Settings. We adopt (and extend) workloads and evaluation strategies similar to those used in prior work on storage integrity. Here we generate workloads with *fio* [46]. Workloads have a 5-minute warm-up period and 15-minute benchmark period. Unless stated otherwise, mean latency or throughput is reported in each graph.

When selecting workloads, we must consider how PAC would be deployed in a real system. Exactly when the read, write, and flush functions are called depends on how the application interacts with the disk. It may read and write directly to the raw device file—e.g., as the PostgreSQL database does. In contrast, it may interact with the disk through a file system, which is what many applications use to store unstructured data. In either case, the application might bypass the OS page cache (by opening the regular file or device file with the O_DIRECT flag) or rely on the page cache for buffered I/O. In the latter case, the sizes and timing of block I/Os issued to the driver depend on how often the application explicitly calls FSYNC (to write out dirty pages), or if memory pressure causes the kernel to begin writing dirty pages back to disk.

We examine the performance space under these different scenarios by running experiments directly against the storage device and varying key system and workload parameters (capacity, cache size, etc.). This allows us to provide generalizable insights on PAC performance and offer guidance

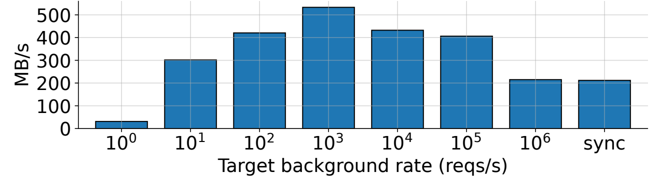


Figure 10: Aggregate read/write throughput vs. target background processing rate. “sync” denotes synchronous DMT, and the remaining data points denote DMT+PAC at various rates. There is a region of optimal performance between 100 and 100k requests/s, with the maximum throughput observed at 1k requests/s.

on steering applications towards optimal configurations (e.g., tuning flush frequency). We toggle the O_DIRECT flag in *fio* to do this, per best practice [47]. We focus especially on write-heavy workloads with a Zipfian shape, which closely approximate real-world storage access patterns [37], [48].

Unless stated otherwise, default parameters include—Read ratio: 1%, I/O size: 32 KB, Thread count: 1, I/O depth: 32, Capacity: 1 TB, Cache size: 10%, Target background rate: 1000 requests/s, FSYNC period: 1000 updates. These parameters showcase the best performing configuration for the baselines and reflect the shape and behavior of real-world storage workload patterns [37], [48].

7.2. Results

7.2.1. PAC Performance

PAC's performance is affected both by its configuration parameters (key among these being target background processing rate) and the workload's tolerance of uncommitted writes (represented in our workloads as the FSYNC period). We briefly evaluate and optimize system parameters, followed by a performance evaluation of PAC on diverse workloads.

Impact of Target Background Processing Rate. The target background processing rate describes the target rate that the background thread polls the queue and executes updates. Figure 10 shows that maximum throughput is achieved at 1k requests/s, with a 2.3× speedup over synchronous DMT (the bar labelled “sync”). At a rate of 1M requests/s, the throughput is approximately equal to synchronous DMT.

The target rate that the background thread polls the queue influences the degree of rate limiting and update overriding. If it polls too slowly (i.e., the polling rate is less than the workload IOPS), the queue quickly accumulates update requests and becomes full. This leads to a significant amount of rate limiting to free up queue slots. Moreover, the background thread remains largely idle when it could have been polling faster. In contrast, if it polls too quickly, updates tend to be processed very quickly, which reduces the opportunity to override updates. If the processing of certain updates had been delayed for slightly longer, this wasted work could have been eliminated.

This manifests in Figure 10 as a region of optimal performance between 100 and 100k requests/s. To understand

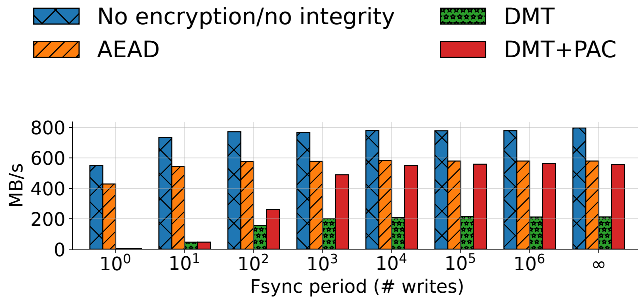


Figure 11: Aggregate read/write throughput vs. FSYNC period. Longer periods between flush calls enable PAC to better exploit parallelism.

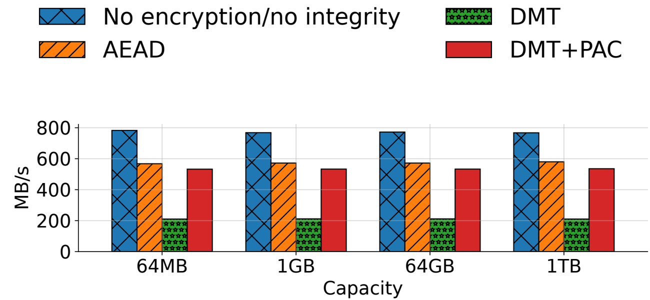


Figure 12: Aggregate read/write throughput vs. capacity. Larger capacities have larger tree heights and thus higher hashing costs. PAC still delivers near-baseline performance.

why, consider that synchronous DMT has a throughput of approximately 200 MB/s, which amounts to approximately 6100 read/write IOPS for 32 KB I/Os. This workload is 99% writes, so 6k of those I/Os were writes (which prompted an update request to be queued). A target rate of 1k requests/s balances this tradeoff best: it is sufficiently fast to prevent the queue from becoming full too quickly, but still gives the background thread sufficient time to exploit reference locality by overriding updates.

Impact of FSYNC Period. We fix the target background processing rate at 1k requests/s and next examine how the FSYNC period influences performance. The FSYNC period describes the duration (in number of write I/Os) between FSYNC calls. It is important because two critical events happen during device flushes—the queue drain and Merkle root sealing. Analyzing how throughput changes w.r.t. the FSYNC period allows us to characterize to what degree parallelism helps when applications issue FSYNC calls more or less frequently.

Figure 11 shows performance of each integrity technique as workload FSYNC period varies. PAC outperforms synchronous DMTs when FSYNCs occur every 100 writes or more, with performance $> 85\%$ of AEAD when FSYNCs occur every 1000 writes or more. In effect, when FSYNCs are sufficiently infrequent, PAC virtually eliminates the overhead of integrity checking. When the FSYNC period is ≤ 1000 updates, flushes are more frequent so the background thread executes most updates while draining the queue during the flush. Thus, there is little opportunity for the background thread to amortize update costs.

These observations imply that the background thread effectively races the main thread to amortize update costs. If the driver is kept sufficiently busy (i.e., ≥ 1000 update requests) between flushes and flushes are infrequent, then update (and sealing) costs can largely be amortized. However, if the driver is starved of I/O requests (e.g., because they are still sitting in the page cache) or flushes are frequent, PAC falls back to (in essence) doing updates synchronously.

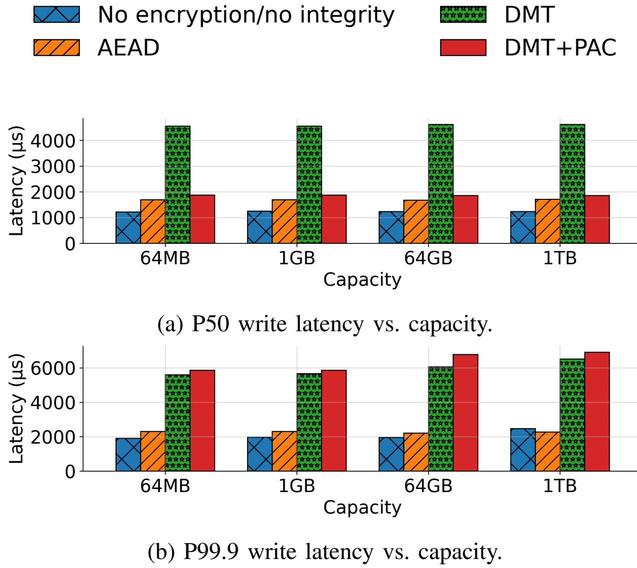
It is worth noting that prior works have tackled this problem (in the synchronous case) with *batched state updates* [28], [45], which artificially increases the FSYNC period, but weakens security guarantees. We argue that this is not only insecure, but unnecessary. Rather than batching state updates, we can reframe the FSYNC period as a principled tuning knob for applications—maintaining strong security guarantees and eliminating overheads by instead restructuring the way in which applications issue dirty page writebacks and flushes [49], [50]. Tools like auto-tuners are increasingly being used to help improve application performance by finding optimal application configurations (i.e., optimal write-back and flush behaviors) [51]. These tools therefore can (and should) be used to guide applications towards the optimal region—which we prescribe is at an FSYNC period > 1000.

Takeaway: When properly configured, PAC delivers 85% of the throughput of the AEAD baseline (**RQ1**). This shows that it is possible to provide strong storage integrity guarantees with minimal overhead.

7.2.2. System Scalability

Our above analysis showed that if flushes are carefully managed, PAC can deliver up to a 2.8× speedup over DMTs and achieve near-zero overhead. Next we examine the scalability of PAC—i.e., whether it can deliver a stable performance guarantee across other system and workload settings that characterize storage deployments. This will elicit how viable PAC is as a general solution.

Scaling with Capacity. Figure 12 shows how disk capacity impacts performance. Disk capacity affects the size of the tree—notably, the height. Update and verification costs increase at larger capacities because the tree depth increases. We observe across all capacities that aggregate read/write throughput is approximately 200 MB/s for DMTs and 500 MB/s for PAC. This amounts to a 2.5× speedup and near-baseline performance. Note that DMTs were previously shown to deliver a stable performance guarantee w.r.t. capacity. This shows that PAC can maintain a stable performance guarantee. Latency improvements reflect these observations.



(a) P50 write latency vs. capacity.

(b) P99.9 write latency vs. capacity.

Figure 13: PAC median and tail latency improvements reflect throughput improvements.

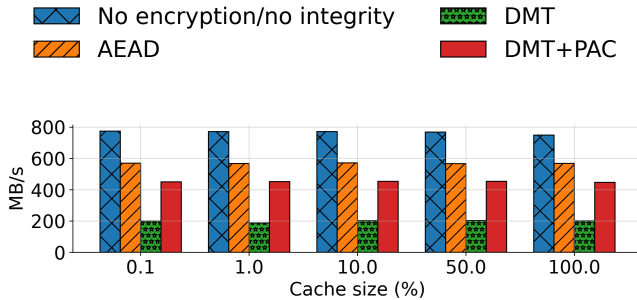


Figure 14: Aggregate read/write throughput vs. cache size. Larger caches help reduce metadata I/O costs.

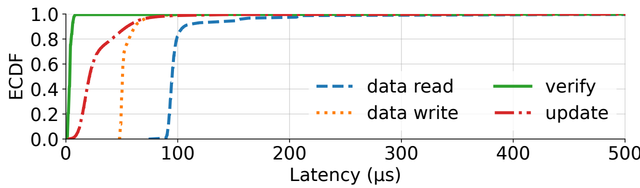


Figure 15: Distribution of verification and update latencies. Verification costs are negligible. Thus, deferred verification [24], [25] is not only insecure, but it is also unnecessary.

Figure 13 shows that median latency has a 2.5× improvement across all capacities.

Impact of Cache Size. Next we examine how the hash cache affects performance. The hash cache plays a central role in minimizing metadata I/O and prompting early exits during verifications. The goal here is to understand whether

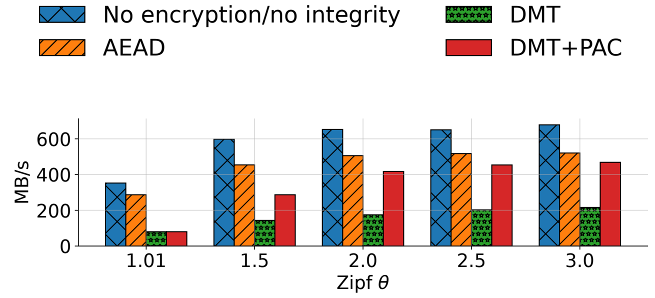


Figure 16: Aggregate read/write throughput vs. workload skewness. More skewed workloads enable PAC to override updates and better amortize update costs.

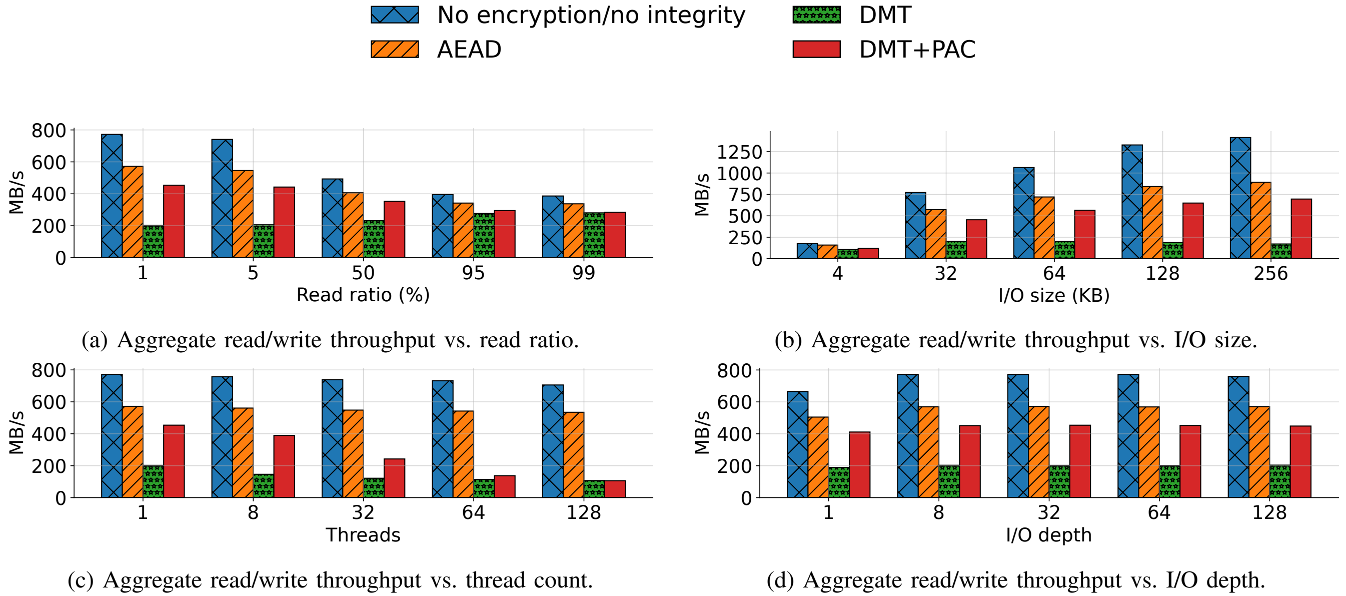
the limitations of DMTs described in Section 4 cannot be solved by simply using a larger cache.

Figure 14 shows that at a cache size of 0.1%, PAC delivers a 2.5× speedup over DMTs and near-baseline performance. In fact, it maintains that speedup to a cache size of 100%—small caches (e.g., 0.1%) are very efficient, and performance gains are negligible beyond that. This cache efficiency is also reflected in the per-block verification and update latencies; for 32 KB I/Os this means that 8 verifications or updates are executed per read/write. Figure 15 shows the latency distribution at a cache size of 10%. While updates benefit from caching to an extent, the key insight is that cache hits prompt early exits on nearly all verification operations, making verification latencies largely negligible. We observe that verifications take approximately 2 μs per block on average, compared to updates which take a median of 25 μs per block. This is significant because prior works [24], [25] have relied on deferring verifications to offset costs. This shows that not only is deferred verification insecure, but it is also unnecessary.

Note that for 72 B nodes (in DMT) and a 1 TB disk, a cache size of 0.1% amounts to 38.6 MB of memory. Examining cache sizes smaller than this is moot; the effective price for DRAM at these capacities is the same. Thus, the limitations of DMTs cannot be solved by simply using a larger hash cache: efficiently executing updates requires more judicious algorithm design. PAC provides a promising direction towards this.

Impact of Workload Skewness. Next we examine how workload skewness affects performance. The dynamic nature of DMTs enables them to exploit reference locality in skewed workloads, but PAC can further exploit locality by overriding updates. We therefore anticipate significant speedups under more skewed workloads.

Figure 16 shows that PAC delivers a 2× speedup over DMTs under a Zipf(1.5) workload and a 2.5× speedup under a Zipf(2.5) workload. Performance saturates for both trees when $\theta = 2.5$. However, PAC delivers >90% of the AEAD baseline throughput when $\theta > 1.5$.



(a) Aggregate read/write throughput vs. read ratio.

(b) Aggregate read/write throughput vs. I/O size.

(c) Aggregate read/write throughput vs. thread count.

(d) Aggregate read/write throughput vs. I/O depth.

Figure 17: `PAC` delivers higher throughputs under write-heavy workloads, larger I/O sizes, and higher I/O depths, and comparable performance otherwise.

Impact of Read Ratio, I/O Size, Thread Count, and I/O Depth. Next we examine how other workload parameters affect performance. The read ratio describes the percentage of I/Os that are reads vs. writes; `PAC` performance should converge to that of its base data structure (DMT) when workloads are read-heavy, but capitalize on asynchronous updates otherwise. Examining I/O sizes and depths allows understanding how performance changes depending on the nature of how I/Os are issued to the driver from the kernel (e.g., during dirty page writeback). Examining thread counts will elicit whether multiple application threads help or hurt performance in general. We anticipate that `PAC` can best amortize update costs under write-heavy workloads, larger I/O sizes, and higher I/O depths. All of these settings provide sufficient conditions to keep the background thread busy while the main thread proceeds with other useful work.

Figure 17a shows that `PAC` delivers a $2.5\times$ speedup under write-heavy workloads (read ratio: 1%). When workloads are read-heavy, `PAC` performance is the same as DMTs, because verifications are done synchronously. `PAC` scales better with I/O size: at 256 KB I/Os, `PAC` observes a speedup of $5.5\times$. Across application thread counts (Figure 17c), a single thread is sufficient to saturate disk throughput; beyond that, contention leads to lower throughputs in general. Across application I/O depths (for applications that use asynchronous I/O on files), `PAC` speedups remain largely unaffected.

Takeaway: Across storage sizes, workload characteristics, and cache designs, `PAC` provides stable performance guarantees (and up to a $5.5\times$ speedup over DMTs) that approach those of AEAD (**RQ2**). Even in the worst case, `PAC`'s tail latency is comparable to prior work that provides the same guarantees.

7.2.3. Memory & Storage Overhead

`PAC` requires additional memory capacity for the queue, but no additional storage capacity over DMTs. DMTs have a 72 B Merkle tree node object. In `PAC`, request objects in the queue thus occupy $72 + 16 + 12 = 100$ B, for a memory overhead of $1.38\times$. However, we showed that `PAC` maintains a $> 1.38\times$ speedup with $1.38\times$ less cache memory on average across cache sizes ranging from 0.1% to 100%.

Takeaway: `PAC` delivers better performance per dollar spent on cache memory (**RQ3**). As a result, it makes efficient use of system resources while providing complete integrity guarantees.

Evaluation Summary: `PAC` shows that, in practical settings, it is possible to achieve near-zero overhead integrity protection. In contrast to conclusions made in prior works, reads never need to be speculative, and challenges associated with writes (state update costs/rollback guarantees) can be largely addressed with judicious parameter selection.

8. Discussion & Future Work

Optimized Checkpointing. State updates should in fact be committed before acknowledging to the caller that a state update is durable. However, because sealing costs may be high for some use cases, periodic or batched state updates could be used to improve performance—i.e., only sealing after N state updates. Effectively all prior works have been implemented and evaluated using these methods [28], [45].

This weakens rollback guarantees. For example, applications rely on flush commands for durability guarantees. Consider that an application receives acknowledgment that a state update $S_0 \rightarrow S_1$ is durable, then externalizes some data or acknowledgment to an end-user. If the state update was not actually sealed yet, a malicious crash could be used to roll the system back to state S_0 while the user perceives the system to be in state S_1. Thus, users would continue execution under the guise of a successful recovery.

`PAC` does not use these optimizations, and we described in our evaluation how practical performance can feasibly be achieved by tuning application write-back parameters (which is standard practice) rather than security parameters.

Crash Resilience and Recovery. Being able to recover from both malicious and non-malicious crashes is an important problem. Currently, `PAC` operates in a fail-stop model, similar to Nimble [28] and other recent works. This means that correctness is upheld in `PAC` after a crash, but recovery is not guaranteed. How to comprehensively recover from crashes in this context is an open problem that introduces new threat models, security-performance trade-offs, and requires different solutions.

In general, crashes can be non-malicious or malicious, and they can result from either power loss or software fault (e.g., kernel panic). Non-malicious and malicious crashes resulting from power loss can be handled gracefully by using an uninterruptible power supply or other battery-backed hardware to provide temporary power. Non-malicious and malicious crashes resulting from software faults can be handled (partially) through techniques like journaling. However, an attacker could cause additional crashes during recovery or corrupt, reorder, replay, or truncate parts of the journal. Thus, authentication and freshness of the journal itself must be ensured, among other concerns. This introduces a trade-off space between integrity and availability guarantees that must be modelled and evaluated. We left analysis of these problems to future work.

Implications for Practical Deployments. To date, storage integrity methods have either deferred the freshness and consistency problems (e.g., only encrypting and authenticating at the block level), or targeted constrained settings such as key-value stores (where performance requirements are often easier to achieve due to larger object sizes). In contrast, our work demonstrates that disk authenticity, transactional consistency, and freshness can be achieved for general-purpose block devices, and with minimal overhead compared to encryption-only approaches. In the same way that trusted execution environments (TEEs) have enabled general-purpose compute workloads to be directly provided with new guarantees, `PAC` can be used to ensure storage integrity for arbitrary application workloads with untrusted storage hardware.

Such an abstraction enables new practical applications. For instance, abstraction to the block layer enables commodity databases to provide strong storage integrity guarantees. Virtual machine images could be freshness-guaranteed at boot time to prevent downgrade attacks; such protections are increasingly valuable as diverse software supply chains increase the risk of severe security vulnerabilities in older versions of software. Most importantly, these assurances can be achieved without trusting the underlying storage provider. They could prove especially useful for providers that are required to assure protection of user data, such as under the EU GDPR, as both private compute and data storage can now be fully attested even if the underlying physical provider falls outside national jurisdiction.

Asynchronous Integrity Checking as a Pipeline Problem. Ultimately, faulty data from untrusted storage only has negative security consequences when the application performs operations based on that data. Our current threat model assumes that any code executed based on erroneous data constitutes an integrity violation, but future work could see further performance improvements by precisely characterizing the flow from erroneous data to externally-visible actions.

In effect, read verification could be treated the same way as memory reads in an out-of-order CPU pipeline. Verification could be deferred until an externally-visible action (e.g., another disk write, network requests, or even mutation of shared memory) is taken as a result of that data. Because read verification *always* succeeds absent an injected fault (an unrecoverable event), application code can proceed under the assumption that verification succeeds, and a segmentation fault can be thrown in the event verification fails. The effectiveness of such an approach will rely on whether side effects of reads can be accurately identified with minimal performance overhead.

9. Related Work

Merkle hash trees have emerged as a fundamental building block of secure storage systems, secure memory systems, and authenticated data structures broadly [4], [13], [14], [20], [24], [34], [36], [52], [53], [54], [55], [56]. We discuss these related works below.

Secure Storage & Memory. Attacks against cloud services have motivated a significant amount of research on building secure storage and memory systems. For storage, authenticated disk encryption has become the standard method to protect the confidentiality and authenticity of cloud disks [15], [43]. Linux `dm-verity` implements a block-level Merkle hash tree and is seeing increasing use in emerging cloud services [57]. It has also become critical to providing verified boot for mobile and embedded device storage [4]. Secure memories like Intel SGX also rely on Merkle trees to protect

the integrity (notably, the freshness) of volatile memory [13], [14], [16], [17], [36], [58], [59].

Recent works such as FastVer [24], Concerto [25], dm-x [22], and DMT [23] have designed optimizations to reduce Merkle tree overheads for storage. For example, FastVer and Concerto use the batching approach discussed in Section 4, which sacrifices integrity guarantees. dm-x relies on high-degree (e.g., 128-ary) trees to reduce the tree height and thus the number of hashes that have to be computed on every read or write to storage. DMT implements a hash tree that dynamically adjusts itself at runtime based on workload patterns; this work also showed that the high-degree trees used by dm-x have significant overheads at large capacities and thus high-degree hash trees are not a suitable solution for real-world systems. Recent works such as VAULT [20] and Penglai [36] have also demonstrated that hash tree overheads are significant for secure memories and designed optimizations addressing them. These works have relied primarily on using high-degree trees and various caching techniques to reduce overheads.

Our work builds on these efforts by moving beyond data structure-level optimizations and examining the potential to exploit concurrency to cut costs entirely. We show that with careful design, it is indeed possible to deliver integrity protection with near-zero overhead.

Integrity Data Structures. The theoretical properties of Merkle hash trees and other authenticated data structures (e.g., authenticated skip lists) have also been examined by the cryptography community in the context of blockchains [52], certificate revocation systems [60], and provable data possession schemes [21], [54], [61], [62], [63]. These works have broadly examined the problem of integrity checking outsourced storage in an *offline* model. In an offline model, integrity checks occur on-demand as requested by the data owner. This model reflects a batching approach. It suits archival cloud storage, where users issue infrequent integrity checks over their data and performance is not a primary concern. In contrast, integrity checks are tightly coupled with applications in an *online* model: they are executed each time data is read from or written to disk. This model is necessary for cloud applications that make real-time control decisions based on data read from an untrusted disk, and where performance is a primary concern.

Prior works have converged on Merkle hash trees as the dominant solution to protecting integrity [21]. However, while their logarithmic complexity has translated into relatively lower overheads for an offline setting, recent works have shown that their overheads in an online setting are often prohibitive [20], [23], [24]. `PAC` showed that it is possible to minimize these overheads in the context of high-performance storage devices.

Rollback Protection. Rollback attacks are similar to but distinct from replay attacks [26], [27], [45]. Storage systems require protection against both. Our focus is not on designing new rollback protection protocols; secure methods to prevent rollbacks are known—for example, by sealing the Merkle root with a tamper-resistant counter. We discuss rollbacks, because introducing asynchrony complicates rollback guarantees. Specifically, `PAC` uses a checkpointing mechanism to ensure the liveness of the background thread (i.e., that state updates are not being delayed) and to ensure that state updates are in fact being committed. This happens during an `FSYNC` system call.

These prior works do not prescribe precisely *when* state updates should occur, only the means through which an update will occur if requested [26], [27], [45], [64]. The common wisdom is that deciding when it is appropriate to a commit a state update is application-specific and has thus largely been left up to application developers. Optimizations such as batch committing a state change after N state updates have also been proposed to reduce costs associated with sealing. Effectively all prior works have been implemented and evaluated using these methods. The result is a lack of sound guidance on when and how to properly commit state updates, and weak rollback guarantees in general.

`PAC` prescribes that state updates are committed at the time of a flush, which exactly reflects the disk state that applications perceive to be durable. Moreover, our evaluation showed that batching commits is not necessary to achieve good performance: `PAC` commits an update on every flush call and still delivers speedups.

10. Conclusion

Merkle hash trees provide robust integrity guarantees for data stored in the cloud. However, they introduce additional compute and I/O costs on the I/O critical path, which degrades performance. We showed that closing the performance gap requires moving beyond data structure-level optimizations to algorithm-level optimizations. We proposed a new integrity checking method called `PAC` that exploits concurrency to reduce hashing costs on the critical path while still providing equivalent security guarantees. Our evaluation showed that it is possible to achieve near-zero overhead integrity protection for untrusted cloud storage.

Acknowledgments

We thank the anonymous reviewers and shepherd for their insightful feedback. This work was supported in part by the Semiconductor Research Corporation (SRC) and DARPA.

References

[1] Amazon Web Services, "Amd sev-snp in amazon ec2," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html, 2024.

[2] Google Cloud, "Confidential vms overview," https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview, 2024.

[3] Microsoft Azure, "Confidential vms overview," https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview, 2024.

[4] Android, "Implementing dm-verity," https://source.android.com/docs/security/features/verifiedboot/dm-verity, Google, Inc., 2023.

[5] Amazon AWS, "Amazon elastic block store," https://aws.amazon.com/ebs, Amazon Web Services, Inc., 2023.

[6] Google Cloud, "Google cloud persistent disks," https://cloud.google.com/persistent-disk, Google, Inc., 2023.

[7] Microsoft Azure, "Microsoft azure managed disks," https://learn.microsoft.com/en-us/azure/virtual-machines/managed-disks-overview, Google, Inc., 2023.

[8] B. Johannesmeyer, A. Slowinska, H. Bos, and C. Giuffrida, "Practical data-only attack generation," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1401–1418.

[9] C.-T. Huang, L. Huang, Z. Qin, H. Yuan, L. Zhou, V. Varadharajan, and C.-C. J. Kuo, "Survey on securing data storage in the cloud," *APSIPA Transactions on Signal and Information Processing*, vol. 3, p. e7, 2014. [Online]. Available: https://www.cambridge.org/core/product/identifier/S2048770314000067/type/journal_article

[10] F. Bohling, T. Mueller, M. Eckel, and J. Lindemann, "Subverting linux'integrity measurement architecture," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020, pp. 1–10.

[11] A. Kurmus, N. Ioannou, M. Neugschwandtner, N. Papandreou, and T. Parnell, "From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[12] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 218–238.

[13] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "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, 2003, pp. 295–306.

[14] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "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*. Tel-Aviv, Israel: ACM Press, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2487726.2488368

[15] L. Khati, N. Mouha, and D. Vergnaud, "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, 2017, pp. 241–257.

[16] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "SGX-LKL: Securing the Host OS Interface for Trusted Execution," *arXiv:1908.11143 [cs]*, Jan. 2020. [Online]. Available: http://arxiv.org/abs/1908.11143

[17] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 645–658. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[18] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A Secure Database Using SGX," in *2018 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA: IEEE, May 2018, pp. 264–278. [Online]. Available: https://ieeexplore.ieee.org/document/8418608/

[19] R. Perez, R. Sailer, L. van Doorn *et al.*, "vtpm: virtualizing the trusted platform module," in *Proc. 15th Conf. on USENIX Security Symposium*, 2006, pp. 305–320.

[20] M. Taassori, A. Shafiee, and R. Balasubramonian, "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*, 2018, pp. 665–678.

[21] S. A. Crosby and D. S. Wallach, "Authenticated dictionaries: Real-world costs and trade-offs," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 2, pp. 1–30, 2011.

[22] A. Chakraborti, B. Jain, J. Kasiak, T. Zhang, D. Porter, and R. Sion, "Dm-x: protecting volume-level integrity for cloud volumes and local block devices," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, 2017, pp. 1–7.

[23] Q. Burke, R. Sheatsley, R. King, O. Hines, M. Swift, and P. McDaniel, "On Scalable Integrity Checking for Secure Cloud Disks," in *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, 2025, pp. 391–405.

[24] A. Arasu, B. Chandramouli, J. Gehrke, E. Ghosh, D. Kossmann, J. Protzenko, R. Ramamurthy, T. Ramananandro, A. Rastogi, S. Setty *et al.*, "Fastver: Making data integrity a commodity," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 89–101.

[25] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy, "Concerto: A high concurrency key-value store with integrity," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 251–266.

[26] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 379–394.

[27] R. Strackx and F. Piessens, "Ariadne: A minimal approach to state continuity," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 875–892.

[28] S. Angel, A. Basu, W. Cui, T. Jaeger, S. Lau, S. Setty, and S. Singanamalla, "Nimble: Rollback protection for confidential cloud services," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 193–208.

[29] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure linux containers with intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[30] A. Galanou, K. Bindlish, L. Preibsch, Y.-A. Pignolet, C. Fetzer, and R. Kapitza, "Trustworthy confidential virtual machines for the masses," in *Proceedings of the 24th International Middleware Conference*, 2023, pp. 316–328.

[31] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[32] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, "Exploiting unprotected i/o operations in amd's secure encrypted virtualization," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1257–1272.

[33] B. Schlüter, S. Sridhara, A. Bertschi, and S. Shinde, "Wesee: Using malicious# vc interrupts to break amd sev-snp," *arXiv preprint arXiv:2404.03526*, 2024.

[34] R. Avanzi, I. Mihalcea, D. Schall, H. Montaner, and A. Sandberg, "Cryptographic protection of random access memory: How inconspicuous can hardening against the most powerful adversaries be?" *Cryptology ePrint Archive*, 2022.

[35] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "Rote: Rollback protection for trusted execution," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1289–1306.

[36] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the penglai enclave," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 275–294.

[37] J. Li, Q. Wang, P. P. Lee, and C. Shi, "An in-depth comparative analysis of cloud block storage workloads: Findings and implications," *ACM Transactions on Storage*, vol. 19, no. 2, pp. 1–32, 2023.

[38] D. Didona, J. Pfefferle, N. Ioannou, B. Metzler, and A. Trivedi, "Understanding modern storage apis: a systematic study of libaio, spdk, and io_uring," in *Proceedings of the 15th ACM International Conference on Systems and Storage*, 2022, pp. 120–127.

[39] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous I/O stack: A low-latency kernel I/O stack for Ultra-Low latency SSDs," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 603–616. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/lee-gyusun

[40] A. Faria, R. Macedo, J. Pereira, and J. Paulo, "BDUS: implementing block devices in user space," in *Proceedings of the 14th ACM International Conference on Systems and Storage*. Haifa Israel: ACM, Jun. 2021, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/3456727.3463768

[41] A. Jacobs, M. Gülmez, A. Andries, S. Volckaert, and A. Voulimeneas, "System call interposition without compromise," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024, pp. 183–194.

[42] K. Yasukata, H. Tazaki, P.-L. Aublin, and K. Ishiguro, "zpoline: a system call hook mechanism based on binary rewriting," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 293–300.

[43] M. Brož, M. Patočka, and V. Matyáš, "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, 2018, pp. 79–93.

[44] NIST. (2023) Report on the block cipher modes of operation in the nist sp 800-38 series. NIST. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2023/NIST.IR.8459.ipd.pdf

[45] J. Niu, W. Peng, X. Zhang, and Y. Zhang, "Narrator: Secure and practical state continuity for trusted execution in the cloud," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2385–2399.

[46] J. Axboe, "fio - flexible i/o tester," https://fio.readthedocs.io/en/latest/, 2024.

[47] "Benchmarking persistent disk performance on linux," https://cloud.google.com/compute/docs/disks/benchmarking-pd-performance-linux, 2024.

[48] Y. Yang and J. Zhu, "Write skew and zipf distribution: Evidence and implications," *ACM transactions on Storage (TOS)*, vol. 12, no. 4, pp. 1–19, 2016.

[49] M. Larabel, "Fresh take on linux uncached buffered i/o 'rwf_uncached' nets 65~75% improvement," 2024. [Online]. Available: https://www.phoronix.com/news/Linux-RWF_UNCACHED-2024

[50] Y. Qian, M.-A. Vef, P. Farrell, A. Dilger, X. Li, S. Ihara, Y. Fu, W. Xue, and A. Brinkmann, "Combining buffered i/o and direct i/o in distributed file systems," in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, 2024, pp. 17–33.

[51] J. Freischuetz, K. Kanellis, B. Kroth, and S. Venkataraman, "Tuna: Tuning unstable and noisy cloud applications," in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 954–973.

[52] V. Buterin, "Ethereum: platform review," *Opportunities and Challenges for Private and Consortium Blockchains*, vol. 45, 2016.

[53] M. Naor and K. Nissim, "Certificate revocation and certificate update," *IEEE Journal on selected areas in communications*, vol. 18, no. 4, pp. 561–570, 2000.

[54] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 4, pp. 1–29, 2015.

[55] R. Sinha and M. Christodorescu, "Veritasdb: High throughput key-value store with integrity," *Cryptology ePrint Archive*, 2018.

[56] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 121–132.

[57] Amazon Web Services, "Bottlerocket - aws," https://aws.amazon.com/bottlerocket/, Amazon Web Services, Inc.

[58] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 179–190, 2006.

[59] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "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, 2007, pp. 183–196.

[60] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "Coniks: Bringing key transparency to end users," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 383–398.

[61] R. Dahlberg, T. Pulls, and R. Peeters, "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, 2016, pp. 199–215.

[62] R. Tamassia, "Authenticated data structures," in *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*. Springer, 2003, pp. 2–5.

[63] A. Miller, M. Hicks, J. Katz, and E. Shi, "Authenticated data structures, generically," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 411–423, 2014.

[64] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and forking detection for trusted execution environments using lightweight collective memory," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 157–168.