SwornDisk: A Log-Structured Secure Block Device for TEEs

(An *incomplete* version, for the purpose of preview only, June 5, 2022)

Yiming Zhang[1]*, Hongliang Tian[2]*, Erci Xu[3]†, Xinyuan Luo[3], Lining Hu[1],
Lujia Ying[3], Weijie Liu[2], Qing Li[2], Shaowei Song[2], and Shoumeng Yan[2],

[1]*Xiamen University* [2]*Ant Group, and* [3]*PDL*

Abstract

This paper presents SwornDisk, a log-structured secure block device for Trusted Execution Environments (TEEs). SwornDisk allows existing file systems (like Ext4) to be stacked upon it for *transparent* I/O protection against a *strong* adversary outside the TEE. Compared with prior work, SwornDisk has a unique log-structured design, which not only enables *higher* I/O performance for writes, but also achieves a *broader* range of security guarantees, namely, confidentiality, integrity, freshness, consistency, anonymity, and atomicity. We implement SwornDisk for both Intel SGX and AMD SEV. We conduct extensive experimental evaluation: on SGX, the performance of SwornDisk on random writes is up to 10× faster than the state-of-the-art solution of comparable security; on both SGX and SEV, SwornDisk's overall performance is on par with a naive encryption-only solution.

1 Introduction

Trusted Execution Environments (TEEs) are an emerging hardware-based security technology that enables users (e.g., cloud tenants) to run their sensitive applications in isolated and encrypted memory regions, which cannot be snooped or tampered with by privileged attackers (e.g., cloud operators). All major CPU architectures have TEE implementations [1,2,9,10,27,35], e.g., Intel SGX [9] and AMD SEV [1]. As TEE-enabled CPUs become mainstream, TEEs are getting increasingly adopted to resolve trust issues in various use cases, e.g., data management [47], data analytics [50], machine learning [28], and blockchains [39].

While the in-memory data can be protected by TEE hardware, the on-disk data have to be protected by TEE software. According to the threat model of TEEs, the adversaries are *privileged*, *online*, and *active*, being able to monitor, tamper with, and rollback all on-disk states. This requires the TEE software stack to provide a *broad* range of security guarantees for the on-disk data, including but not limited to confidentiality, integrity, and freshness.

However, existing solutions for securing disk I/O of TEEs are not satisfactory in terms of security and/or performance. In an SEV-protected VM, the go-to choice for the guest OS is the Linux kernel. But none of Linux's current security measures for disk I/O (e.g., eCryptFs [26], fscrypt [6], and dm-cyrpt [4]) are designed with TEEs in mind, thus falling short of the minimum security guarantees required by TEEs such as integrity and/or freshness. For Intel SGX, which is more mature and thus better studied, a number of SGX-aware solutions have been proposed. The state-of-the-art Intel SGX Protected File System (SGX-PFS) [8] protects in-place updates with encryption and a Merkle Hash Tree (MHT) [38] to achieve confidentiality, integrity, and freshness. This popular approach is also adopted by other SGX systems [3,33,51,53]. Despite the improved security, we find that SGX-PFS suffers from poor I/O performance, especially for random writes.

Realizing the difficulties encountered by the traditional approach of in-place updates and MHTs, we explore the *log-structured approach* to securing the disk I/O of TEEs. The conventional wisdom from prior log-structured storage systems [32,34,40,49] is that logging (i.e., sequential writes) is friendly to storage medium (including both HDDs [49] and SSDs [40]). Our insight is that *logging is also friendly to security protection against the strong adversaries of TEEs*. For example, an append-only log can be efficiently secured with a MHT or as a chain of blocks. Furthermore, logging benefits crash consistency as old versions of data are not overridden by new ones. Logging also reduces access pattern leakage [29] as data are not updated at fixed addresses.

With this insight, we propose SwornDisk, a *log-structured* secure block device for TEEs. The block interface of SwornDisk allows existing file systems to be stacked upon it for *transparent* I/O protection. Compared with prior work, SwornDisk has a unique log-structured design, which not only enables *higher* performance for random writes but also achieves a *broader* range of security guarantees (§4),

*Equal contribution.
†Corresponding authors.

namely, confidentiality, integrity, freshness, consistency, atomicity, and anonymity. To the best of our knowledge, SWORNDISK is *the first block-level solution that adopts the log-structured approach* to securing the file I/O of TEEs.

The design of SWORNDISK revolves around logging in three folds (§5). First, it uses an *encrypted data log* to persist the updates of user data blocks. The data log maximizes the write throughput and facilitates crash recovery. Second, it maintains a *secure index* for the encrypted data log. Using the index, the latest version of a user data block can be quickly located in the data log, and then decrypted and validated. The secure index is based on log-structured merge trees (LSM-trees) [43]. Third, it introduces a *secure journal* that logs the metadata of all on-disk updates, including those of the encrypted data log and the secure index. The secure journal is key to crash consistency as well as other security properties. The net result of this log-structured design is that the write amplication factor of a random write is reduced to $1+\varepsilon$ ($\varepsilon \ll 1$), which is well below the traditional approach's $2 \times H$.

Aiming at the broad range of security guarantees against the strong TEE adversaries, we find that the most challenging part of the design is the *secure index* (§5.3.1) and the *secure journal* (§5.3.2). The confidentiality, integrity, and freshness of user data are mainly protected with the secure index. To implement it efficiently, we propose *disk-oriented secure LSM-trees*, a tailor-made variant of LSM-trees that can organize its disk components on a raw disk (without the help of file systems), protect the disk components on the untrusted disk from the TEE adversaries, and optimize its query proccessing for block addresses. As to the secure journal, we implement it as a *chain* of encrypted and MAC-protected journal blocks to prevent the adversary from forging a bogus history. The secure journal not only ensures *(crash) consistency* but also *(flush) atomicity*.

We implement two prototypes of SWORNDISK (§6): the first is written in Rust and integrated with Occlum [51], a library OS for SGX; and the second is implemented in C and integrated with Linux for use in SEV. The two prototypes amount to around 10K lines of code. We conduct extensive experimental evaluation (**??**): on SGX, the performance of SWORNDISK on random writes is up to 10× faster than that of the state-of-the-art solution of comparable security; on both SGX and SEV, SWORNDISK's overall performance is on par with a naive encryption-only solution.

We summarize the contributions of this paper as follows.

- We present the design of SWORNDISK, the *first block-level solution that adopts the log-structured approach* to securing disk I/O of TEEs. It achieves both strong security and high performance.

- We implement two prototypes of SWORNDISK, one for SGX and the other for SEV, and conduct an extensive evaluation to show its performance advantages.

2 Background

2.1 Trusted Execution Environments (TEEs)

TEEs protect sensitive code and data from untrusted infrastructures. Modern TEEs can be classified into two categories: enclave-based (e.g., Intel SGX [9]) and VM-based (e.g., AMD SEV [1]). Enclaves create isolated and encrypted memory regions in the address space of user-space processes, while VM TEEs apply strong isolation and memory encryption to protect a VM from a malicious hypervisor or other VMs.

Both enclave-based and VM-based TEEs have several common characteristics. First, they assume powerful adversaries that are *privileged* (e.g., being a host OS or hypervisor), *online* (as they may launch attacks throughout the life cycle of a TEE), and *active* (e.g., tampering with the state of a victim system). Second, TEEs are lack of I/O protection, as I/O devices (e.g., storage or network devices) are beyond the scope of TEEs' hardware protection and thus untrusted by TEEs. Third, TEEs support *remote attestation*: an instance of TEE can authenticate itself to a remote party by showing a hardware-endorsed certificate and subsequently establish a secure communication channel with the party.

Note that the latest generation of SGX [7] can provide abundant trusted memory for enclaves. Thus, unlike prior work [20,31,44,56], we do not consider the capacity of trusted memory as a constraint.

2.2 Log-Structured Storage Systems

It has long been known that sequential writes are faster than random writes on storage medium, including both HDDs [49] and SSDs [40]. This motivates log-structured designs for storage systems, e.g., log-structured merge trees (LSM-trees) and log-structured file systems.

LSM-trees. An LSM-tree [43] is a disk-based index structure that is optimized for inserting key-value records. In modern implementations [12, 17, 23], key-value records are first buffered in a memory component called *MemTable*. When the size of the MemTable grows beyond a threshold, the entire MemTable is persisted on the disk as a *Sorted String Table (SST)*, which is an immutable format for ordered key-value records. This process is called *minor compaction*. In order to reclaim disk space and facilitate query processing, SSTs are organized in multiple levels (L_i) of exponentially growing capacities. Minor compaction creates SSTs at L_1. When the number of SSTs at level L_i reaches a threshold, some SSTs from levels L_i and L_{i+1} will be selected for merge sort, generating new SSTs at L_{i+1} to replace the old ones. This is called *major compaction*. LSM-trees are fast for insertions as the compaction writes SSTs in a log manner.

Log-structured file systems. A log-structured file system [32, 34, 40, 49] writes both metadata and data in a log manner. One way to do this is grouping a large number of consecutive disk blocks into *segments*. Updates of both metadata and data are first buffered in memory and later written to the disk out of place as whole segments.

SWORNDISK borrows the idea of prior log-structured storage systems, but has a different rationale and tackles different challenges in face of the strong TEE adversaries.

3 Motivation

The traditional approach to securing the file or disk I/O of TEEs is protecting in-place updates with Merkle Hash Trees (MHTs). This approach has been adopted by most TEE systems [3, 8, 33, 51, 53], including the state-of-the-art Intel SGX Protected File System (SGX-PFS) [8]. In this section, we will take SGX-PFS as a representative to discuss the performance limitations of the traditional approach, which motivate the design of SWORNDISK.

3.1 SGX-PFS

An open file of SGX-PFS consists of three key components: a variant of MHT (for security), a cache (for efficiency), and a recovery log (for consistency). In the MHT, every node, when stored on the disk, is secured with authenticated encryption. The leaf nodes store file data, while the non-leaf nodes maintain the encryption keys and MACs of their children. The MHT ensures the confidentiality, integrity, and freshness of the entire file. To avoid doing disk I/O for every read or write, the file has a fixed-size, in-memory cache for the most-recently-used nodes. When the cache is full, dirty nodes will be flushed to the disk for durability. Before the flush, the previous versions of the dirty nodes are saved in the recovery log. If any crash happens during the flush, the file can be recovered to its previous consistent state. Working together, the three components of SGX-PFS ensure confidentiality, integrity, freshness, and consistency.

Note that SGX-PFS is a misnomer: although it sounds like a secure file system, it only protects *individual* files. Each protected file of SGX-PFS can is repurposed to serve as a secure block device. Hence, SGX-PFS and SWORNDISK can be compared apples to apples.

3.2 Performance Problem

Write amplification. Random I/O is dominant in a variety of workloads [36]. Unfortunately, SGX-PFS has a poor performance for random writes due to write amplification, i.e., the amount of data written to storage versus the amount of data submitted by user. The write amplification stems from two sources: the MHT and the recovery log. In the MHT, the update of a data node triggers a cascade of updates to all of its ancestor nodes. This means that a random write has to update H nodes of the MHT, where $H \geq 2$ is the height of the MHT. On top of that, the recovery log needs to record the old versions of the updated MHT nodes. Thus, for sufficiently large file, the amplification factor of small random writes is close to $2 \times H$, which is a considerable overhead.

Table 1: A comparison between SGX-PFS and encrypted Ext4 (Crypt-Ext4) throughput on random writes. SGX-PFS suffers from a significant performance degradation.

I/O Sizes	Crypt-Ext4 (MB/s)	SGX-PFS (MB/s)
4KB	39	21 (-44%)
16KB	99	42 (-57%)
64KB	242	70 (-70%)

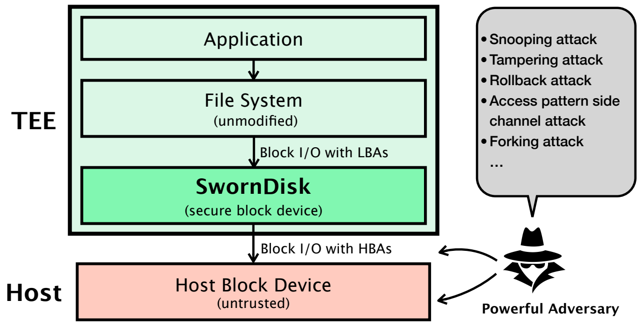


Figure 1: The threat model of SWORNDISK. As a secure block device, it *transparently* protects the disk I/O of the TEE runtime against the strong adversary outside the TEE.

Experimental evaluation. To verify the above analysis, we conducted an experiment to evaluate the random write performance of SGX-PFS. To put things in perspective, we write a strawman implementation of a user-space library that protects file I/O transparently (just like SGX-PFS), but it only uses encryption—no MHTs, nor recovery logs. We name this strawman implementation Crypt-Ext4 and use it as our baseline. To rule out the performance impact of SGX, we ran both Crypt-Ext4 and SGX-PFS in the simulation mode of SGX (instead of the hardware mode). And to make sure most writes go to the disk (instead of the OS page cache), we set the file sizes sufficiently large and issued a `fsync` system call by the end of each benchmark. Both write data to regular files on Linux's Ext4 file system.

The experimental results (see Table 1) show that the random writes of SGX-PFS are significantly slower than that of Crypt-Ext4. Since the major differences between the two of them are whether MHTs and recovery logs are used, we conclude that the performance degradation is primarily attributed to write amplification, which is *inherent* to the traditional approach of in-place updates and MHTs.

4 Threat Model

This section describes the threat model of SWORNDISK.

Usage context. We consider a typical setting of TEE usage, where applications are ported into TEE with no (or few) modifications thanks to a TEE-aware runtime. For Intel SGX, one popular choice for such a runtime is library OSes [46,51,53]. For AMD SEV, one can choose off-the-shelf OS kernels like Linux. As shown in Fig. 1, the runtime is to be integrated with SWORNDISK, which serves as a *trusted* logical block device, protecting all block I/O through it *transparently*. SWORNDISK frees the rest of the TEE from worrying about the security of I/O and enables the reuse of existing file I/O stacks.

Block interface. As a block device, SWORNDISK supports four standard block I/O commands.

1. `read(lba, n, buf)`: read data from the device into the buffer `buf`, starting from the address `lba` for `n` blocks.

2. `write(lba, n, buf)`: write data from the buffer `buf` to the device, starting from the address `lba` for `n` blocks.

3. `flush()`: ensure that all updated data are flushed to the underlying host block device.

4. `trim(lba, n)`: discard the data starting from address `lba` for `n` blocks.

All data write to or read from SWORNDISK is in plaintext. SWORNDISK is responsible for encrypting/decrypting the data transferred to/from the host block device properly. To distinguish between the block addresses on the trusted logical block device (i.e., SWORNDISK) and the untrusted host block device, we term the former as *logical block addresses (LBAs)* and the latter *host block addresses (HBAs)*.

Security assumptions. We assume that the TEE hardware is trustworthy and it protects the confidentiality and integrity of the memory along with the CPU states. We trust all software components within the TEE boundary, and assume that they do not voluntarily leak secrets and are not compromised. We also assume that the software within the TEE can do remote attestation to fetch some secrets (e.g., the root key of SWORNDISK).

Adversary capabilities. We assume a strong adversary who is (i) *privileged*, controlling any hardware and software outside the TEE on the host machine, (ii) *online*, conducting attacks at any timing throughout the lifecycle of the TEE, and (iii) *active*, having the capability of tampering with (not just monitoring) any I/O requests accessing the host block device. Specifically, the classes of possible attacks include but are not limited to: snooping attacks, tampering attacks, rollback attacks, access pattern-based side-channel attacks [29], Iago attacks [24], and forking attacks [20].

There are some attacks that are out of the scope of this paper. First, we do *not* consider denial-of-service or wiper attacks [19], which affect the availability or accessibility of data. Second, we do *not* consider side-channel attacks *except* those that infer sensitive information from disk I/O access patterns, i.e., access pattern-based side-channel attacks. Third, we do *not* consider *complete* rollback attacks, which revert the entire state of a secure disk to an older version. Throughout this paper, we assume rollback attacks to be *partial*: only part of the disk is rolled back by an adversary in an attempt to *mix* up-to-date blocks with outdated ones. Although we do not consider the above attacks, their countermeasures are usually orthogonal and thus applicable to SWORNDISK. For example, complete rollack attacks can be prevented with trusted monotonic counters [20,37,45,52], which can be integrated with SWORNDISK.

Security goals. To counter such a strong adversary, SWORNDISK aims to provide six security guarantees: *confidentiality*, *integrity*, *freshness*, *consistency*, *atomicity*, and *anonymity*. *Confidentiality* means that the user data submitted by any write are not leaked. *Integrity* promises that the user data returned from any read are genuinely generated by the user. *Freshness* ensures that the user data returned from any read are update-to-date. While the first three are "standard", the other three deserve more explanation.

Consistency (or crash consistency) ensures that *all the security guarantees are still held despite of any accidental or malicious crashes*. This means that individual blocks need to be always kept in sync with each other despite of any crashes. In many conventional usage scenarios, this is a only nice-to-have property as crashes are considered rare events. But in TEE environments, the privileged adversary may crash a TEE *arbitrarily*. So we consider consistency a must-have property.

Atomicity (or flush atomicity) demands that *all writes before a flush are persisted in an all-or-nothing manner*. In other words, the number of valid snapshots generated by SWORNDISK must be equal to the number of flushes received by SWORNDISK regardless of the atomicity, ordering, and consistency guarantees enforced at the file system or application level. As such, SWORNDISK can effectively reduce the odds of generating *unexpected* snapshots.

Although snapshots are generated less frequently under the atomicity property, the odds of data loss due to crashes are hardly raised. This is because the timings of generating snapshots are not delayed indefinitely in practice: a file system typically triggers flushes in the background periodically (usually in every few seconds) regardless of whether applications flush data or not.

Anonymity aims to mitigate access pattern-based side-channel attacks by hiding LBAs. The access patterns of I/O operations refer to their addresses, sizes, and types (read/write). Among them, the addresses are the most information-rich; all known attacks rely heavily on the addresses to infer sensitive information [18,22,29]. In SWORNDISK, addresses are classified into two types: HBAs are observable outside the TEE, but LBAs are not. And LBAs are what really useful to the adversary, as they are determined by applications. SWORNDISK hides the LBAs in the sense that (i) LBAs cannot be learned from HBAs; (ii) LBAs cannot be learned from the on-disk data. By hiding LBAs, SWORNDISK greatly reduces usable

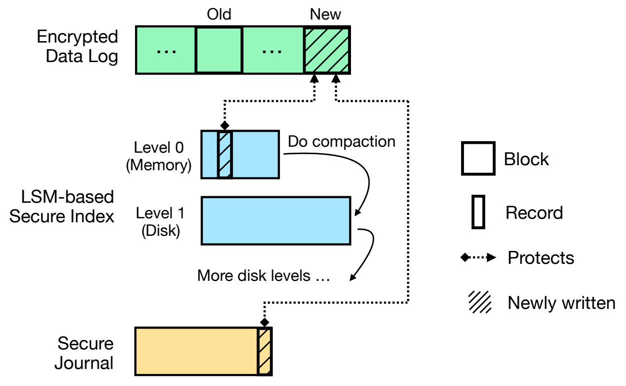


Figure 2: The log-structured approach of SWORNDISK updates a data block by logging a new data block, a journal record, and one or more index records (due to compaction). This results in a write amplication factor of $1+\varepsilon$ ($\varepsilon \ll 1$), which is well below the traditional approach's $2 \times H$ (§3.2).

access pattern information, thus mitigating the attack.

5 Design

In this section, we first present the core idea of SWORNDISK (the log-structured approach), then describe SWORNDISK's storage design, and finish with its security design.

5.1 Log-Structured Approach

Realizing the difficulties encountered by the traditional approach of in-place updates and MHTs, we explore the log-structured approach to securing the disk I/O of TEEs. Specifically, SWORNDISK has a novel log-structured design that revolves around logging in three folds (see Fig. 2).

First, it uses an *encrypted data log* to persist the updates of user data blocks. Writing user data in a log manner maximizes the raw performance of the underlying disk. Besides, the data log allows the outdated versions of user data to coexist with the latest ones, which facilitates crash recovery.

Second, it maintains a *secure index* for the encrypted data log, which maps LBAs to HBAs, encryption keys, and MACs. Using the index, the latest version of a user data block can be quickly located in the data log, and then decrypted and validated. The index itself is implemented as a tailor-made, secure variant of LSM-tree (§5.3.1), which supports updates and queries in an efficient and secure way.

Third, it introduces a *secure journal* (§5.3.2) that logs the metadata of all on-disk updates, including those of the encrypted data log and the secure index. The journal is key to crash consistency as well as other security properties. The journal itself is implemented as a *chain* of encrypted blocks, each of which contains the MAC of the previous one.

SWORNDISK makes novel use of logging to achieve both performance and security advantages. As shown in Fig. 2, one random write now only generates one data block, one log record, and one or more index records (due to compaction). The total size of the records combined is one or two orders of magnitude smaller than that of the data block. Thus, the write amplication factor is reduced to $1+\varepsilon$ (where $\varepsilon \ll 1$), which is well below the traditional approach's $2 \times H$ (§3.2). Besides the low write amplication factor, our log-structured approach also facilitates SWORNDISK's broader range of security protection, especially consistency, anonymity, and atomicity. We will give more details about how SWORNDISK realizes these performance and security advantages.

5.2 Storage Design

5.2.1 Disk Layout

SWORNDISK formats an untrusted disk on the host into five top-level regions (see Fig. 3). *The superblock region* stores the fundamental parameters of a SWORNDISK-format disk, such as the size of a block and the locations of other regions. For robustness, SWORNDISK keeps two copies of the superblock. *The data region* stores the encrypted data log. The region allows multiple logging heads [34] for I/O parallelism. *The index region* stores the LSM-tree-based secure index. The LSM-tree consists of a hierarchy of *Block Index Tables (BITs)* (§5.3.1), each of which is written in a log manner. *The journal region* serves as a large ring buffer to store the secure journal. *The checkpoint region* contains several data structures that summarize the state of SWORNDISK for fast crash recovery.

Note how the regions differ in I/O characteristics. The superblock region is read-only, the checkpoint region is updated in place, while all the rest three regions are based on logging.

5.2.2 Segment Management

Segments. Logging in both the data and index regions are based on segments. In SWORNDISK, the default size of a block is 4KiB and that of a segment is 4MiB. The segments in the data region are called the *data segments* and those in the index region the *index segments*. Logging over segments repeats the following three steps: (i) allocate a free segment in the region, (ii) log (encrypted) data into an in-memory buffer of the segment size, and (iii) write the entire buffer to the segment.

Segment management includes allocating, freeing, and more interestingly, cleaning segments. *Segment cleaning* refers to reclaiming the space of a dirty segment by migrating its valid blocks to new locations while discarding its invalid or outdated ones. Segment management is implemented with the following data structures stored in the checkpoint region.

Segment Validity Table (SVT). A SVT is a bitmap where each bit indicates whether a segment is valid or not. A valid segment contains at least one block that is up-to-date or useful.

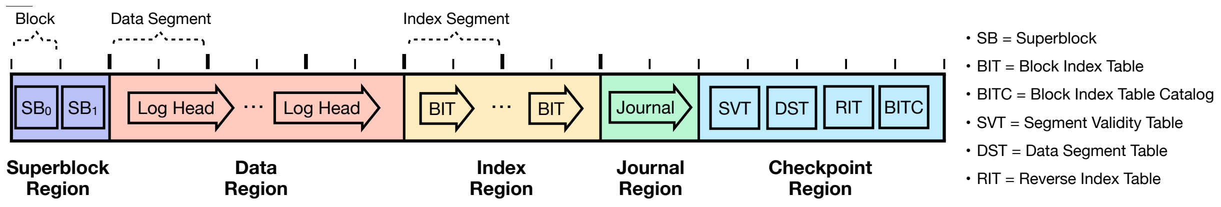


Figure 3: The disk layout of SWORNDISK. It consists of five top-level regions. Blocks (4KiB, by default) are the smallest unit of reads and writes, while segments (4MiB, by default) are used as the unit of logging in the data/index regions.

There are two SVTs, one for data segments, one for index segments. An empty data or index segment can be allocated by flipping a bit of value 0 in the corresponding SVT.

When it comes to free or clean segments, the index segments are relatively easy: as they are only used in their entirety (see BITs in §5.3.1), they can be freed by simply updating the index SVT. Thus, they do not need to be cleaned. In contrast, a data segment may contain both valid and invalid blocks. Thus, data segments cannot be freed directly and need to be cleaned first. Segment cleaning depends on the two data structures below.

Data Segment Table (DST). The DST contains some per-segment metadata of the data segments, e.g., which blocks in a segment are valid and when is a segment modified. The metadata are useful to make wise decisions as to which victim data segments should be selected for cleaning.

Reverse Index Table (RIT). The RIT contains the LBAs of the valid blocks in every data segment. As such, it can be viewed as a mapping from HBAs to LBAs, the reverse of the secure index.

Segment cleaning starts with selecting a victim data segment wisely by consulting the DST. Then, the LBAs of the valid data blocks in the victim segment are determined by looking up the RIT. Finally, these valid blocks are migrated to new on-disk locations and the index records of their LBAs are updated in the secure index. The optimization techniques that we adopt for segment cleaning are described in §6.

Security. All on-disk data structures of SWORNDISK adopt cryptographic protection for security, except the SVT and DST. The two data structures only impact SWORNDISK's decisions as to which segments to allocate or clean, which are not essential to security. On the other hand, it suffices to protect RIT with only encryption, no MACs (see our discussion on anonymity in §5.3.3 for details).

5.2.3 I/O Operations

Read. To serve a read request starting from an LBA for a specified number of blocks, SWORNDISK first retrieves the HBAs, encryption keys, and MACs of the requested user data blocks from the secure index. Then, it reads and decrypts the encrypted data blocks from the data region. After verifying the integrity, the plaintext data is returned to the user.

Write. When receiving a write request, SWORNDISK saves the new data in a segment buffer temporarily and notifies the user of the completion of the write immediately. Later, when the segment buffer becomes full or a flush request is received, SWORNDISK encrypts each block in the buffer with a randomly generated encryption key and also calculates its MAC. The buffer is written in its entirety by allocating a new free segment on the disk. New index and journal records are generated accordingly for durability and security.

Flush. The flush operation ensures durability by triggering writing the new data in the temporary segment buffers to the disk. Unlike conventional block devices, SWORNDISK's flush operation also guarantees atomicity (§5.3.2).

Trim. The trim operation is like the write operation except that no new data are persisted, only the metadata (e.g., the index) are updated.

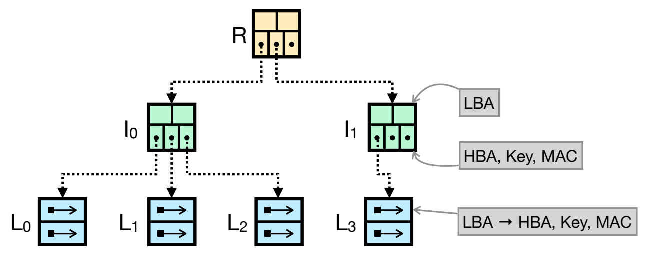
In addition to the external I/O operations triggered by users above, there are some internal I/O operations, including segment cleaning (§5.2.2), index compaction (§5.3.1), journaling, checkpointing, recovery, and commitment (§5.3.2). Most of these internal I/O operations relate closer to the security design, which will be described in the next subsection.

5.3 Security Design

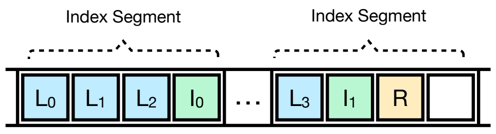
We present the security design of SWORNDISK in three parts. First, we describe the *disk-oriented secure LSM-trees*, with which the secure index is implemented to protect the confidentiality, integrity, and freshness of user data. Second, we discuss the *atomicity-guaranteed secure journaling*, the key to ensuring consistency and atomicity. Last, we discuss the cryptographic protection scheme as well as the anonymity property.

5.3.1 Disk-Oriented Secure LSM-trees

We design *Disk-Oriented Secure LSM trees (dsLSM-trees)*, a tailor-made variant of LSM-trees for efficient storage of the secure index. We call such an LSM-tree *disk-oriented* because it can (i) organize its disk components on a raw disk without the help of file systems, (ii) protect its disk components on the untrusted disk from TEE adversaries, and (iii) optimize its query processing for LBAs. These characteristics set dsLSM-



(a) The logical view



(b) The on-disk view

Figure 4: The structure of a Block Index Table (BIT). Partitions in a dsLSM-tree are BITs. A BIT is structured like the fusion of a B+tree and a MHT, where each leaf node (L_i) is an array of block index records `LBA → (HBA, key, MAC)` and each non-leaf node (I_i and R) manages its children by keeping their LBA ranges, HBAs, keys, and MACs. A BIT can be persisted efficiently in a log manner.

trees apart from the general-purpose LSM-trees and make the performance optimized for use in SWORNDISK.

Block Index Tables (BITs). Similar to modern LSM-trees, dsLSM-trees adopt the partitioning optimization technique, where the key-value records at each level are divided into a set of fixed-size partitions, i.e., SSTs mentioned in §2.2. In dsLSM-trees, however, we replace SSTs with Block Index Tables (BITs). At a high level, BITs and SSTs are alike: both are created in a log fashion and kept immutable afterward; both store sorted key-value records to facilitate fast queries; and both require doing compaction periodically to merge smaller ones into bigger ones.

However, BITs use a completely different representation: unlike a SST, which is a "flat" file, a BIT is organized into a tree structure, which can be viewed as a fusion of a B+tree and a MHT (see Fig. 4a), to serve three purposes simultaneously: disk management, query processing, and security protection. Each BIT node has a fixed size that is a multiple of the block size. When stored on the disk, a BIT node is protected with authentication encryption. The root node (R) and internal nodes (I_i) manage their child nodes by keeping their LBA ranges, HBAs, encryption keys, and MACs. Each leaf node (L_i) is an array of block index records, `LBA → (HBA, Key, MAC)`, sorted in order of `LBA`. The "3-in-1" structure not only fulfils SWORNDISK's specific needs but also benefits data locality.

BITs can be constructed and persisted efficiently. During compaction, the records of a new BIT are generated in an ascending order of LBAs. Thus, the nodes of a BIT can be generated in an order as if doing a depth-first, post-order tree traversal. Finally, the BIT nodes are written sequentially over a series of index segments (see Fig. 4b). Note that after the construction of a BIT is done, it becomes immutable.

Block Index Table Catalog (BITC). A dsLSM-tree consists of a dynamic number of BITs. To keep track of these BITs as they come and go, we introduce a data structure called BITC. It consists of multiple BIT entries, each of which contains the metadata of a BIT, mainly including the BIT ID, the level, the key range, and the HBA, key, and MAC of the root BIT node. The BITC is stored in the checkpoint region and protected with authenticated encryption.

Two-level caching. While LSM-trees are fast for insertions, they can be slow for queries. In the worst case, an LSM-tree has to access one partition (i.e., a BIT in our case) per level to answer a query and accessing each partition requires multiple disk reads. One widely-adopted optimization is using a Bloom filter [21] for each partition to avoid fruitless I/O. Unfortunately, Bloom filters are not helpful in our situation for two reasons. First, a Bloom filter, which tells if an element is in a set, is only effective for point queries, not range queries. But for dsLSM-trees, range queries are the majority as most reads and writes involve multiple blocks. Second, Bloom filters or other filters [55] require an amount of memory that is proportional to the disk capacity of SWORNDISK. A filter typically consumes 10 bits per key and the number of keys in SWORNDISK is proportional to that of data blocks. Thus, a 2TB-capacity SWORNDISK would consume over 1GB of pinned memory for the filters alone, which may be unacceptable in certain use cases.

To accelerate the query processing of dsLSM-trees, we introduce the *two-level caching* optimization which consists of two caches: the lower-level cache named *BIT Node Cache (BNC)* and the higher-level cache named *Search Lookaside Radix Tree (SLRT)*

The idea of BNC is straightforward: given the ID of a BIT node, it attempts to return the plaintext of the BIT node. If the cache hits, it saves both I/O and cryptography costs of loading a BIT node and its ancestors from the disk into the memory.

The SLRT is intended to reduce the chance of searching every level of the dsLSM-tree for a range query. Given a query for a range of LBAs, the SLRT attempts to return the IDs of the leaf nodes that contain the *latest* index records for the LBAs. The SLRT is implemented as a radix tree [11, 15], which is space-efficient and also fast in searching and updating consecutive keys.

In the best case (when both caches hit for a range query), we can first use the SLRT to get the IDs of all the "right" BIT leaf nodes and then use the BNC to retrieve their plaintext content, thus fulfilling the range query without triggering any I/O. Although cache misses occur in practice, we expect the two caches to have reasonably high hit rates for two reasons. First, it is well known that real-world workloads have some degrees of I/O locality [42] and skewness [54]. Second, both BNC and SLRT cache information about the secure index, whose size is two or three orders of magnitude smaller than that of the disk capacity. Thus, a great proportion of the information may be cached in memory.

5.3.2 Atomicity-Guaranteed Secure Journal

As a log of disk updates, the journal of SwornDisk is conceptually similar to that of other storage systems, especially file systems. Despite the similarity, it has some unique characteristics. First, the journal persists a sequence of *chained* blocks (protected with authenticated encryption) each of which embeds the MAC of the previous one, so that we can rule out the possibility of being misled by a bogus history of operations forged by an adversary. Second, the records in the journal include cryptographic information about their corresponding on-disk updates, so that the journal records can not only ensure durability but also secure the confidentiality, integrity, and freshness of the updates. Last, the journal goes beyond its core responsibility of ensuring crash consistency, providing the extra benefit of flush atomicity. Based on these characteristics, we call it *atomicity-guaranteed secure journal*. In the remainder of this subsection, we focus on how the secure journal is used to achieve (crash) consistency and (flush) atomicity for SwornDisk.

It is worth noting that although the two properties of consistency and atomicity are intertwined to some degree, they are two separate concepts; one does not lead to the other naturally. In file systems, consistency is achieved by realizing atomicity for at least some of their internal updates, with the help of journaling and/or transactions. But such atomicity is internal—it is not part of the contract provided by the file system interface—and the external behaviors vary between different systems. For example, two out of the three journaling modes (including the default) of Ext4 [5] only protect the atomicity of metadata but not data [25]. BTRFS [48] and ZFS [16], on the other hand, do protect data as well as metadata with their internal transactions. But since the users cannot control the scopes and timings of the transactions, they cannot count on these transactions for the atomicity of data. Unlike file system journaling, SwornDisk's journaling not only ensures consistency but offers atomicity as part of its contract with users for the flush requests.

SwornDisk realizes consistency and atomicity with four internal operations: journaling, checkpointing, recovery, and commitment.

Journaling. After writing an update to the disk, whether it is made to a data segment, a BIT node, or a checkpoint, SwornDisk writes a corresponding journal record for the durability and security of the update. Periodically or triggered by a flush request, SwornDisk synchronizes the disk and then flushes the latest journal records to the disk. The disk synchronization is necessary to ensure that a disk update is always persisted *before* its corresponding journal record. There are five types of journal records, as summarized in Table 2.

Checkpointing. To reclaim the disk space consumed by outdated journal records and speed up the recovery process, SwornDisk periodically transforms journal records into a more compact format called *checkpoint packs*. A checkpoint pack consists of a creation timestamp, the head and tail positions of the secure journal, the metadata of segments (i.e., SVT, DST, and RIT), and the metadata of BITs (i.e., BITC). The SVT and DST are safe to be kept in plaintext (§5.2.2); the RIT is protected with encryption only (§5.3.3); and the BITC is protected with authenticated encryption.

Checkpoint packs are persisted in the checkpoint region. For crash consistency, the region keeps two checkpoint packs so that at least one of them is valid. After persisting a checkpoint pack on the disk, SwornDisk writes in the secure journal a *checkpoint pack record* that refers to the new checkpoint pack. The record includes two classes of information: a block bitmap (specifying the set of blocks that constitutes the checkpoint pack), and cryptographic information (e.g., encryption keys and MACs, to protect the RIT and BITC).

Recovery. During initialization, SwornDisk picks the more recent one out of the two checkpoint packs, from which it reads the head and tail cursors of the secure journal. The recovery process starts by scanning from the head of the secure journal to locate the *last* checkpoint pack record, according to which SwornDisk initializes its in-memory data structures. From this point, it continues reading the rest of the journal, one record at a time, deciding whether it should be accepted or not in order to restore SwornDisk to a consistent state.

Commitment. Flush atomicity demands that *all writes before a flush are persisted in an all-or-nothing manner*. To do this, we introduce *data commit records*, which are written to the journal upon receiving flush requests from the user. We say a data write *committed* if there is a flush afterward, and similarly, a data log record *committed* if there is a data commit record afterward. If a data write or a data log record is not committed, then it is *uncomitted*. Clearly, flush atomicity essentially claims that *only the committed data count*. Therefore, we adopt the following strategy for the recovery process: accept all committed data log records while ignoring the uncommitted.

Accepting the committed data log records is obviously necessary. Next we discuss why the uncommitted records can be simply ignored (without *undo* operations). This is because SwornDisk takes some extra steps to prevent uncommitted data writes from having lasting or irreversible impacts on the data, index, or checkpoint region. (i) For the data region, SwornDisk invalidates the old versions of a logical data block only if it has a new version committed. (ii) For the index region, the in-memory component of the dsLSM-tree keeps track of whether a contained block index record is committed or not. When compacting the in-memory component, only the committed portion shall be flushed to the disk as a new BIT. (iii) For the checkpoint region, there exists an extra constraint:

Table 2: The journal record types. Data log records, BIT node records, and checkpoint pack records protect the disk updates to the data, index, and checkpoint regions, respectively. A BIT compaction record saves the metadata and progress of a BIT compaction so that if interrupted due to a crash, it can be resumed upon recovery. Data commit records are related to flush atomicity.

Journal Record Types	Description	Fields
Data log record	Summarizes changes made to a data segment	Segment ID, timestamp, blocks' LBAs and crypto info, etc.
Data commit record	Marks prior data as committed	None
BIT compaction record	Saves the progress of a BIT compaction	Compaction ID, old BIT IDs, new BIT IDs, etc.
BIT node record	Summarizes a new BIT node	BIT ID, Node ID, LBA range, HBA, crypto info, etc.
Checkpoint pack record	Summarizes a new checkpoint pack	Block bitmaps, crypto info, etc.

checkpointing can only be done right after a flush, i.e., a checkpoint pack record must be preceded by a data commit record. This ensures that checkpointing does not persist any effects of uncommitted data.

5.3.3 Directeded Graph of Cryptographic Protection

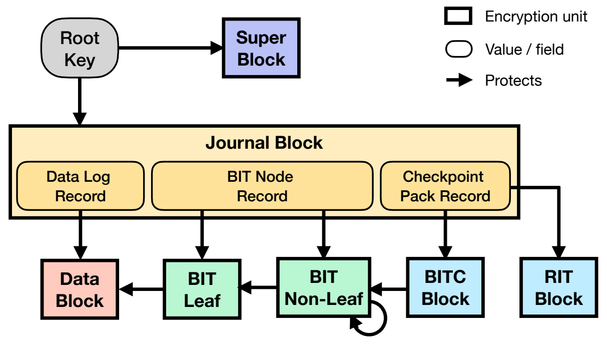


Figure 5: The directed graph of the cryptographic protection relationships between SWORNDISK's on-disk data structures. The encryption units are blocks and (BIT) nodes.

We finish the security design by explaining how SWORNDISK's cryptographic scheme protects its on-disk states and why it suffices to achieve our security goals, especially anonymity.

Protection graph. To protect the confidentiality, integrity, and freshness of data blocks, we have established a directed graph of cryptographic protection relationship between the on-disk data structures of SWORNDISK (see Fig. 5). Blocks and (BIT) nodes are the units of encryption. All blocks (except RIT blocks) and nodes are protected with authenticated encryption; RIT blocks are encrypted, but not MAC-protected (see the discussion on anonymity later). SWORNDISK can guarantee the security of user data as it can build paths from the root key to the data blocks. Thus, the security of the root key implies that of user data.

Root key. The root of trust of SWORNDISK is the root key, which is assumed to be securely possessed by the TEE owner and is given to a SWORNDISK instance upon startup. Typically, the root key is fetched securely by the TEE via remote attestation.

Superblock. The superblock is directly protected with the root key. And the encrypted content of the superblock is stored along with its MAC in the same block.

Key generation. Each block or node, except the superblock which is directly protected with the root key, is protected with a unique encryption key, which is generated by one of the two methods: random key generator and deterministic key derivation. The key of a data block, a BITC block, or a BIT node (including both leaf and non-leaf) is generated randomly and saved by its "parent" node for future retrievals. In contrast, the key of a journal or RIT block is determined via a deterministic key derivation function, whose inputs include a key derivation key (KDK) and a sequence number. For example, the KDK of journal blocks is the root key of SWORNDISK and the sequence number is the ever-increasing logical IDs of journal blocks. Using deterministic key derivation simplifies key management as only the KDK needs to be saved.

MAC management. For a data block, a BITC block, or a BIT node, its MAC is also kept in its "parent" node just like its encryption key. In contrast, each journal block has two copies of MAC. The first copy is stored as plaintext beside the encrypted journal block on the disk. For I/O efficiency, the size of a journal block is actually set to be smaller than that of a regular block so that a journal block plus its MAC can fit in a regular block. The second copy of a journal block's MAC is stored by its *next* journal block. That is, journal blocks are *chained*, so that SWORNDISK can validate the integrity of each journal block, and more importantly, the integrity of the entire secure journal.

Anonymity. In SWORNDISK, the RIT is the only on-disk data structure that is protected by encryption without MACs. The RIT needs to be encrypted as it contains the sensitive information of LBAs, which, if leaked, would violate our goal of anonymity. It is safe to store the RIT without integrity protection, because the reversed index provided by RIT can be readily validated with the secure index. In conclusion, since SWORNDISK adopts logging for disk I/O and all data structures that contain LBAs are encrypted on the disk, LBAs are guaranteed not to be leaked outside the TEE.

6 Implementation

Two prototypes. We implemented two prototypes of SWORNDISK. The first one, written in Rust, is integrated with Occlum for use on SGX. The second one is a Linux kernel module written in C, which is intended for use on SEV. The Rust prototype is about 3K lines of code (LoC), while the C prototype 4K LoC.

Virtual block devices. SWORNDISK, as well as the other two baselines described in **??**, are virtual block devices. Their implementations are based on an in-kernel abstraction layer for virtual block devices, which enables file systems to use different virtual block devices as storage in a uniform way. In the Linux kernel, the abstraction layer is called the device mapper subsystem [13]. As for Occlum, we implement such an abstraction layer, several virtual block devices, and a new file system that can use these devices for storage. These modules amount to around 4K LoC in Rust.

Key acquisition. To open a disk partition protected by SWORNDISK, one needs the root key, which is typically acquired by doing remote attestation in a user program. But user programs depend on the root file system, which itself should be mounted on SWORNDISK for I/O protection. This chicken-and-egg dilemma can be solved with *early user space* [14], which is supported by both Linux and Occlum via `initramfs`, a temporary, in-memory root file system. Its content is shipped with the OS kernel itself and its integrity can be verified through remote attestation. It is this early user space where SWORNDISK is prepared to mount the root file system.

Segment cleaning. Segment cleaning is known to be the primary overhead of the log-structured file systems [34,40,49]. To minimize the overhead, we integrate into SWORNDISK two optimization techniques from prior work. First, we support both *foreground and background cleaning*, which adopt the greedy and cost-effective [30, 49] victim selection policies, respectively. The foreground cleaning minimizes latencies, while the background one emphasizes efficiency. Second, we switch from normal logging to *threaded logging* [41] to reduce user-visible latencies when the rate of disk utility is high. Threaded logging writes new data to the "holes" of partially valid segments without cleaning them upfront.

7 Conclusion

Acknowledgments

References

[1] AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/. Accessed: 2021-11-16.

[2] Arm Confidential Compute Architecture (Arm CCA). https://developer.arm.com/architectures/architecture-security-features/confidential-computing. Accessed: 2021-11-16.

[3] Asylo project. https://github.com/google/asylo. Accessed: 2021-11-16.

[4] Device-mapper's "crypt" target. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-crypt.html. Accessed: 2021-2-1.

[5] Ext4 filesystem. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt. Accessed: 2021-11-16.

[6] Filesystem-level encryption (fscrypt). https://www.kernel.org/doc/html/latest/filesystems/fscrypt.html. Accessed: 2021-11-16.

[7] How 3rd Generation Intel® Xeon® Scalable Processor platforms support 1 TB EPCs. https://www.intel.sg/content/www/xa/en/support/articles/000059614/software/intel-security-products.html. Accessed: 2021-11-16.

[8] Intel protected file system library. https://www.intel.com/content/dam/develop/external/us/en/documents/overviewofintelprotectedfilesystemlibrary.pdf. Accessed: 2021-11-16.

[9] Intel Software Guard Extensions (SGX). https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. Accessed: 2021-11-16.

[10] Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. Accessed: 2021-11-16.

[11] Judy arrays. http://judy.sourceforge.net/doc/10minutes.htm. Accessed: 2021-11-16.

[12] LevelDB. https://github.com/google/leveldb. Accessed: 2021-11-16.

[13] Linux device mappers. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/index.html. Accessed: 2021-11-16.

[14] Linux early user spacec. https://www.kernel.org/doc/Documentation/early-userspace/README. Accessed: 2021-11-16.

[15] Linux radix trees. https://www.kernel.org/doc/html/latest/core-api/generic-radix-tree.html. Accessed: 2021-11-16.

[16] OpenZFS Wiki. https://openzfs.org/wiki/Main_Page. Accessed: 2021-11-16.

[17] RocksDB. https://github.com/facebook/rocksdb. Accessed: 2021-11-16.

[18] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[19] Jinwoo Ahn, Junghee Lee, Yungwoo Ko, Donghyun Min, Jiyun Park, Sungyong Park, and Youngjae Kim. Diskshield: A data tamper-resistant storage for intel sgx. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 799–812, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. SPEICHER: securing lsm-based key-value stores using shielded execution. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 173–190. USENIX Association, 2019.

[21] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, jul 1970.

[22] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 668–679. ACM, 2015.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.

[24] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.

[25] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243, 2013.

[26] Michael Austin Halcrow. eCryptfs: An enterprise-class encrypted filesystem for Linux. In *Proceedings of the 2005 Linux Symposium*, volume 1, pages 201–218, 2005.

[27] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. Confidential Computing for OpenPOWER. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 294–310, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *CoRR*, abs/1803.05961, 2018.

[29] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[30] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems, New Orleans, Louisiana, USA, January 16-20, 1995, Conference Proceedings*, pages 155–164. USENIX Association, 1995.

[31] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Oper. Syst. Rev.*, 40(3):102–107, 2006.

[33] Sandeep Kumar and Smruti R. Sarangi. Securefs: A secure file system for intel sgx. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, page 91–102, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In Jiri Schindler and Erez Zadok, editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 273–286. USENIX Association, 2015.

[35] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[37] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David M. Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: rollback protection for trusted execution. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1289–1306. USENIX Association, 2017.

[38] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[39] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. Proof of luck: an efficient blockchain consensus protocol. In *Proceedings of the 1st Workshop on System Software for Trusted Execution, SysTEX@Middleware 2016, Trento, Italy, December 12, 2016*, pages 2:1–2:6. ACM, 2016.

[40] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In William J. Bolosky and Jason Flinn, editors, *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, page 12. USENIX Association, 2012.

[41] Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Optimizations of LFS with slack space recycling and lazy indirect block update. In Gadi Haber, Dilma Da Silva, and Ethan L. Miller, editors, *Proceedings of of SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference, Haifa, Israel, May 24-26, 2010*, ACM International Conference Proceeding Series. ACM, 2010.

[42] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 297–306. ACM Press, 1993.

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

[44] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 238–253, New York, NY, USA, 2017. Association for Computing Machinery.

[45] Bryan Parno, Jacob R. Lorch, John R. Douceur, James W. Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 379–394. IEEE Computer Society, 2011.

[46] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host os interface for trusted execution, 2020.

[47] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 264–278. IEEE Computer Society, 2018.

[48] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: the linux b-tree filesystem. *ACM Trans. Storage*, 9(3):9:1–9:32, 2013.

[49] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In Henry M. Levy, editor, *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*, pages 1–15. ACM, 1991.

[50] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 38–54. IEEE Computer Society, 2015.

[51] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel SGX. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 955–970. ACM, 2020.

[52] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 875–892. USENIX Association, 2016.

[53] Chia-che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 645–658. USENIX Association, 2017.

[54] Yue Yang and Jianwen Zhu. Write skew and zipf distribution: Evidence and implications. *ACM Trans. Storage*, 12(4):21:1–21:19, 2016.

[55] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 323–336, New York, NY, USA, 2018. Association for Computing Machinery.

[56] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 2182–2194, New York, NY, USA, 2021. Association for Computing Machinery.