A Log-Structured Merge Tree-aware Message Authentication Scheme for Persistent Key-Value Stores

Igjae Kim, *UNIST, KAIST;* J. Hyun Kim, Minu Chung, Hyungon Moon, and Sam H. Noh, *UNIST*

https://www.usenix.org/conference/fast22/presentation/kim-igjae

This paper is included in the Proceedings of the 20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Open access to the Proceedings of the 20th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

A Log-Structured Merge Tree-aware Message Authentication Scheme for Persistent Key-Value Stores

Igjae Kim* *UNIST, KAIST*
J. Hyun Kim *UNIST*
Minu Chung *UNIST*
Hyungon Moon † *UNIST*
Sam H. Noh *UNIST*

Abstract

Persistent key-value stores (KVSs) are fundamental building blocks of modern software products. A KVS stores persistent states for the products in the form of objects associated with their keys. Confidential computing (e.g., Intel Software Guard Extensions (SGX)) can help KVS protect data from unwanted leaks or manipulation if the KVS is adapted to use the protected memory efficiently. The characteristics of KVSs accommodating a large volume of data amplify one of the well-known performance bottlenecks of SGX, the limited size of the protected memory. An existing mechanism, Speicher, applied common techniques to overcome this. However, its design decision does not scale because the required protected memory size increases rapidly as the KVS receives additional data, resulting from the design choice to hide the long latency of Merkle tree-based freshness verification. We find that the unique characteristics of the log-structured merge (LSM) tree, a data structure that most popular persistent KVSs have, help reduce the high cost of protected memory consumption. We design TWEEZER on top of this observation by extending RocksDB, one of the most popular open-source persistent KVSs. We compare the performance of TWEEZER with the reproduced version of Speicher. Our evaluation using the standard `db_bench` reveals that TWEEZER outperforms Speicher by 1.94∼6.23× resulting in a reduction of slowdown due to confidential computing from 16∼30× to 4∼9×.

1 Introduction

Persistent key-value stores (KVSs) are a cornerstone of modern software products. Many cloud services, such as Netflix [41], Facebook [58] and Uber [18], use these as a storage engine for large-scale data processing [17, 50] or database management systems [34, 36, 59]. Accordingly, KVSs are responsible for securely maintaining service data, including user credentials and private information. Thus, cloud-based services are motivated to protect their KVSs with the strongest mechanism available. This paper presents a KVS protected through hardware-based confidential computing (e.g., Intel SGX (Software Guard Extensions) [26]). While our work is not the first such work, our study is unique in that our solution is 1) tailored to the *log-structured merge* (LSM) tree, 2) general in that our solution is not tied to any particular hardware support, and 3) superior in performance (by up to 6.23×) compared to the state-of-the-art.

Hardware-based confidential computing offers strong security guarantees to such KVSs. Most KVSs run on public cloud services, leaving their content potentially open to anyone with control of the cloud platform's privileged software or physical machines. Confidential computing allows the KVSs to exclude these complex software layers and any hardware but the processor chip itself from the *trusted computing base* and rely only on the correctness of the processor implementation. The execution context within the processor chip is protected with access control mechanisms and those on external memory are protected cryptographically by encryption and the *message authentication code (MAC)*. Such a protected execution environment is commonly called an *enclave*.

This appealing security guarantee comes at the cost of performance. Among others, the cryptographic protection of external memory content introduces limitations in external memory usage. The confidentiality guarantee requires the in-memory data to be encrypted, while the integrity guarantee requires MAC computation and verification. As the cost of MAC increases with the total amount of memory that is available to an enclave, providing processors with large memory to an enclave becomes prohibitive. Thus, such memory, which is called the *enclave page cache (EPC)*, typically is available only in 128 MB or 256 MB [25] capacity depending on the choice of design and implementation. An enclave may use memory beyond EPC capacity, but the pages that do not fit in the EPC must be paged out of the EPC with similar cryptographic protection. Also, applications may access these memory pages only if the pages are loaded back into the EPC. Therefore, applications must be carefully redesigned to

*Work done mostly as an undergraduate student at UNIST.
†Corresponding Author

minimize such EPC paging and may have to store large data chunks with manual protection. This requirement has motivated many popular applications to be tailored to the enclave protection model [6, 9, 12, 16, 23, 31, 47, 54, 56].

For a persistent KVS to be protected through hardware-based confidential computing, it must be tailored considering the EPC limitations, as it is a memory-heavy application dealing with a large (e.g., more than tens of gigabytes) amount of data. Many persistent KVSs use the LSM tree for data in storage, accompanied by in-memory caches (e.g., *MemTable*) and write-ahead logs (WAL). The LSM tree and WAL must be manually protected with encryption and MAC, and the MemTable must also be tailored to efficiently employ EPC because it is relatively large by default (e.g., 64 MB). Speicher is the first work in this direction where it presents a design to efficiently protect the three large data structures that persistent KVSs commonly have [6]. The design divides the MemTable into two and places only the smaller one with more frequent access in EPC. The other two data structures are also protected with encryption and MAC with the Merkle tree [19]. However, this design choice slows down the KVS by up to 32.5× as the large Merkle tree induces longer latency for data retrieval from the LSM tree and increases the use of EPC pages by other caches as our analysis will show (§7.2).

This paper presents TWEEZER, which shares the same goal as Speicher. Similar to Speicher, TWEEZER is an extension of RocksDB [58], a popular LSM tree-based persistent KVS, that uses the MAC scheme tailored for LSM trees to run efficiently in an SGX enclave. However, TWEEZER is different from Speicher in that we make three critical design decisions on top of these invariants.

First, TWEEZER ensures the freshness of an LSM tree without constructing a Merkle tree spanning across its *sorted string tables (SSTables)*. This is possible by leveraging the principle that an LSM tree-based KVS comprises many SSTables, each containing many key-value pairs and remains immutable once built until it is compacted. Thus, if an SSTable is authenticated with a unique key and the key is never reused, an attacker cannot find other pieces of data anywhere other than the SSTable to perform the replay attack (§5.2).

Second, the uniqueness and invariant ordering of keys in each *data block* enable TWEEZER to encrypt and authenticate each key-value pair separately without losing the capability of detecting replays within an SSTable. We find that the invariant ordering among and within the data blocks enables TWEEZER to detect any attack on freshness without the Merkle tree generated for each SSTable (see §5.3).

Third, we find the classic hash chain [51] to be a good fit for authenticating the two logs: the WAL and MANIFEST logs. Hash chains allow TWEEZER to authenticate the logs without the trusted counters, which Speicher relies on, as well as to create as many new log entries as needed (§5.4).

We implement TWEEZER by extending RocksDB 6.14 [58] and using Scone [4], a library operating system designed to run unmodified applications in an SGX enclave. Besides the LSM tree-tailored message authentication scheme, we adopt the design choices for Speicher [6] for MemTable. We also reproduce Speicher for a comparison study due to the lack of an open-source version and demonstrate that the reproduced version provides similar performance.

Our experimental study using `db_bench`, the standard benchmark used for RocksDB, indicates that TWEEZER achieves the expected performance gain and EPC efficiency. When tested with extensive data, TWEEZER outperforms Speicher by 1.94∼6.23× depending on the workload and the data size. Evaluations using the same benchmark configuration that Speicher was evaluated with also exhibit similar performance benefits (1.91∼3.94×). Our analysis reveals that this improvement is primarily due to the 5.24∼7.57× reduction in EPC paging frequency.

2 Background

2.1 Intel SGX

Intel SGX [26] provides an execution environment called an enclave that has a protected memory region called the EPC [14] for programs that need protection. Only the program running in the enclave can access its EPC content that is cached in the CPU cache. When the data must be evicted to external memory, data are encrypted and authenticated using MAC by the *memory encryption engine* [27]. Thus, even strong attackers, who can replace external memory, cannot obtain or corrupt the EPC content and leave it undetected.

This memory protection mechanism is a well-known performance bottleneck [4, 12, 31]. The SGX computes the keyed *hash MAC (HMAC)* for each cache line, composes a modified version of the Merkle tree [22], and keeps its root within the CPU hardware to ensure the freshness of EPC content stored on external memory. Each cache replacement operation is accompanied by MAC verification using the Merkle tree to ensure that the EPC as a whole remains as written by the enclave. Partially for this reason, the size of the hardware-protected EPC is limited (e.g., 128 MB in general, 256 MB in recent releases [24]). For real-world applications that need larger memory, SGX provides paging of EPC, but the encryption and MAC verification make this paging expensive as well. Consequently, an application as-is that is not tailored to this policy suffers from significant performance overhead [6, 31].

Another source of performance penalty is the increased system call overhead. An enclave runs as part of a user process as a separated execution context from which an additional context switch is required to invoke system calls. Therefore, most applications running on an enclave adopt asynchronous system calls as a performance optimization [4, 6, 38, 43, 62, 63, 68], where an application creates a thread that stays in the user's context with the role to mediate system calls from the enclave. TWEEZER adopts this by running on SCONE [4].

2.2 LSM-based Key-Value Stores

RocksDB [58] is an open-source persistent KVS that is widely used in production and that uses the LSM tree [42] as its data structure for the key-value pairs in storage. The four critical components of RocksDB are the MemTable, SSTable, WAL, and MANIFEST log.

The MemTable is designed to reside in memory and stores recently added key-value pairs using a skip list for fast lookup. Every `put` operation fills the MemTable before the data are flushed to a persistent medium. If the size of the MemTable becomes larger than a configurable threshold, it is marked as immutable, and another MemTable is created to serve the following writes. At the same time, RocksDB triggers a background flush thread to move the immutable MemTable to a persistent medium in the form of an SSTable.

The new SSTables generated from a series of MemTables constitute Level 0 of the LSM tree. Any new read request must look up all of the SSTables in Level 0 because any SSTable could contain the key. Thus, RocksDB needs to keep retained the number of SSTables in Level 0, and thus, triggers an operation called *compaction* when the number of SSTables at Level 0 exceeds a configurable threshold. A compaction thread running in the background selects several SSTables, deletes duplicated keys, and compacts them to create a new SSTable stored at the lower level, Level 1 here, of the LSM tree in storage. The resulting levels satisfy an additional property of ordering. The compaction procedure ensures that one key appears at each level at most once (except for Level 0), and every SSTable is sorted, allowing the KVS to look up, at most, one SSTable per level to find a key-value pair.

One SSTable comprises several sub-blocks including *index blocks* and *data blocks*. The index block contains a sorted sequence of *index keys*. The ith index key is larger than or equal to the keys in ith data block and smaller than the keys in $i+1$th data block. At the end of an SSTable is a *footer block* containing padding to align the SSTables and a magic number marking the end of an SSTable. Speicher stores the MACs of the SSTable's key-value pairs in this footer block, resulting in increased EPC usage when the KVS becomes large.

2.3 Speicher

Bailleu et al. [6] were the first to study the problem of running RocksDB efficiently on an enclave and designed Speicher by adapting RocksDB. Speicher uses the Merkle tree to authenticate LSM tree by computing a MAC for each data block and building the Merkle tree on top. They propose three design changes to an LSM tree-based KVS considering the characteristics of the enclave and its protected memory, EPC. First, the MemTable must be adapted to reduce the EPC usage. Speicher redesigned MemTable so that a large portion of it, the values on leaves, are stored explicitly outside the EPC with cryptographic protection. This design change improves KVS throughput by reducing the number of EPC paging that Speicher causes. Second, the I/O calls must be handled at the user level by another thread to avoid leaving the enclave context on every call. Speicher runs with its own direct I/O library based on Intel SPDK [1], which reduces the cost of additional context switches. Third, the KVS should be properly timestamped to defeat the rollback and forking attacks. Speicher uses its own asynchronous monotonic counter that wraps the synchronous SGX monotonic counter because they could not use the SGX counter directly.

3 Threat Model

We assume a strong attacker could acquire complete control of a system running TWEEZER, except for the enclave's context that is protected by SGX. They may have obtained such control by exploiting a known vulnerability in the cloud provider's system or as an insider responsible for maintaining them. In particular, such attackers can read or modify the contents of memory or storage that the user's KVS uses, except for those in EPC that the SGX protects. However, the attacker cannot directly query the KVS because the KVS does not accept the attacker as an allowed client. The design and implementation of such an authentication protocol is a well-studied problem and orthogonal to the design of TWEEZER. We also do not aim to propose a new remedy to address implementation bugs that the current implementation of Intel SGX is known to have, potentially nullifying its security guarantee completely [11, 13, 60, 65, 67] as these are not fundamental flaws in its security model and will be fixed in future releases. This aligns with the assumptions made by most existing mechanisms built on Intel SGX [6, 9, 12, 16, 23, 31, 47, 54, 56].

4 Overview

TWEEZER is a persistent KVS running on an SGX enclave. To users, TWEEZER provides all operations that persistent KVSs usually implement as an extension of RocksDB [58]. The only additional requirement for users is to retrieve and keep a pair of cryptographic keys (§5.5) and place *heartbeat* transactions (§6). The key pair is required for TWEEZER to recover its data in case of a crash and the heartbeat transactions provide rollback resilience.

TWEEZER is built on top of the advances made by an earlier work, Speicher [6] (see §2.3), with three additional design decisions (D1∼D3 below).

D1. TWEEZER creates and associates one unique MAC key with each SSTable as shown in Figure 1 (①). Whenever TWEEZER stores data outside the SGX-protected memory, it computes the MAC to store along with the data to later verify the freshness. Among these data are the LSM tree, which comprises many SSTables in storage. The large size of this LSM tree, which contains almost all key-value pairs, could

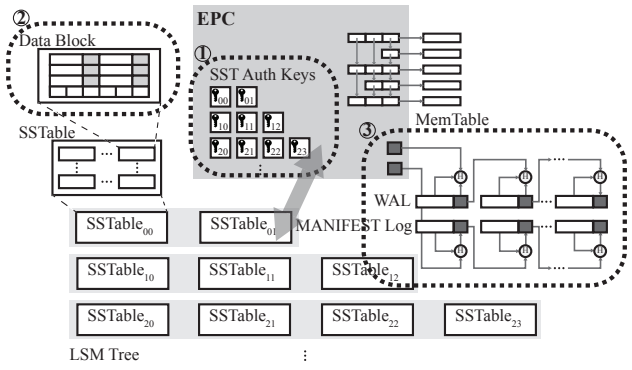


Figure 1: An overview of design choices made for TWEEZER.

make the MAC computation expensive. The best-known way to ensure the freshness of this large chunk of data is to build a Merkle tree [22], as Speicher does, at the cost of potentially long latency. TWEEZER avoids composing the Merkle tree spanning the entire LSM tree by associating a unique MAC key to each SSTable by taking advantage of the three properties of LSM trees: the immutability of each SSTable, the uniqueness of a key in each level, and the sorted keys in each data block. We elaborate on these design details in §5.2 and provide an in-depth security analysis in §6.

D2. We associate a MAC with each key-value pair rather than with each data block in an SSTable, as shown in Figure 1 (②). For encrypting and computing the MAC, the data block could be a natural unit. The SSTables are supposed to reside in storage optimized for block-level access, and RocksDB fetches and caches the key-value pairs at this granularity. What renders this design choice potentially inefficient is the limited size of EPC. For the desired security guarantee, the data block must reside in the EPC while being accessed, consuming valuable EPC space. This setup does not incur a significant performance cost when the KVS serves a relatively small data set and is configured to have only a small block cache. However, the blocks in EPC quickly become a performance bottleneck when the KVS requires a larger block cache to accommodate more data [69], which could quickly exhaust the small EPC. TWEEZER reduces this *read amplification* in EPC usage by encrypting and authenticating each key-value pair separately. This design choice enables TWEEZER to save EPC space and use non-EPC memory more effectively as a cache for SSTables. One drawback of this design choice is the increased use of storage space because the fine-grained encryption makes the subsequent per-block compression unproductive and increases the number of MACs stored in SSTables. We evaluate and discuss this in §8 (Figure 12).

D3. We overcome the absence of trusted counters [28] in the latest Intel SGX using hash chains (③ in Figure 1). Another performance-critical piece of data in persistent KVSs is the WAL that a KVS builds in storage to recover recently updated key-value pairs after a crash. The logs must be protected with encryption and MAC because they are supposed to reside in storage for persistence. Appropriate encryption and MAC computation provide confidentiality and integrity guarantees, but freshness requires each log entry to be associated with additional data. Speicher proposed to use the trusted counter [28] that an earlier version of the SGX SDK had, but which has been discontinued [21, 28]. Hence, TWEEZER constructs a hash chain to protect the content and the order of the logs. The hash chain alone is not enough to prevent the rollback attack, so TWEEZER requires the user to place a heartbeat transaction to timestamp the KVS version and use it later to verify that a snapshot of TWEEZER is the latest one. We elaborate on this aspect in §5.4.

5 Design and Implementation

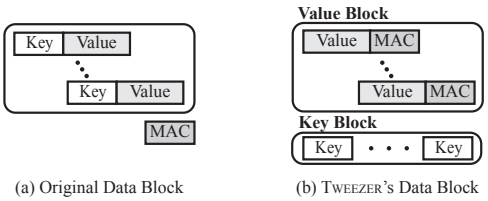
5.1 Data Encryption

TWEEZER manually encrypts all data that are stored outside EPC and decrypts them only within EPC. For example, TWEEZER ensures the SSTable content remains encrypted in both storage and memory and decrypts them only within EPC when it obtains a key-value pair from the SSTable. We use AES GCM mode with 256-bit key as the encryption scheme. As such, TWEEZER encrypts all data stored outside EPC to protect their confidentiality. For the rest of this section, we focus on how TWEEZER ensures freshness with the authentication scheme tailored for LSM trees.

5.2 Authentication with Per-SSTable Key

TWEEZER computes the HMAC of SSTables to later verify their freshness. When TWEEZER creates a new SSTable in the process of compaction or flush, it generates a new secret authentication key that is used exclusively for the particular SSTable (see ① in Figure 1) and stores it in EPC and the MANIFEST. TWEEZER then computes a MAC for each piece of data in the newly created SSTable (§5.3) and stores the MAC along with the encrypted data in the SSTable file. When TWEEZER reads the SSTable later to obtain a key-value pair, it computes the MAC again using the authentication key of the SSTables kept in EPC and compares it with the MAC stored along with the key-value pair to determine if the data has been maliciously corrupted or not.

The use of SSTable-unique keys and a set of invariant checks enable TWEEZER to guarantee the freshness of key-value pairs using HMAC. This HMAC is strong enough to prevent an attacker from generating correct MAC for an arbitrary piece of data because the correct MAC computation requires the secret key. However, a strong attacker who can obtain all pairs of data and MAC can replay the collected pairs. For example, if a KVS uses a single key to generate MAC for multiple SSTables ($SST_0, \cdots, SST_n$), the attacker can obtain pairs of



(a) Original Data Block (b) TWEEZER's Data Block

Figure 2: The structure of the original data block (left) and TWEEZER's data block (right).

SSTables and their MACs, $(SST_0, MAC_0), \cdots, (SST_n, MAC_n)$ and present one of the pairs to bypass the verification. Such a KVS using one key for multiple SSTables accepts this replayed pair because the computed MAC matches the presented MAC. To defeat this, the KVS must have a means to detect the replay, such as the Merkle tree.

TWEEZER avoids composing a Merkle tree spanning across all SSTables leveraging the uniqueness of the key for each SSTable and the immutability of SSTables. In general, the use of distinct keys suffices to prevent the replays across the set of data authenticated with different keys. However, this still does not prevent the replays within the data sharing one key because the attacker still has multiple pieces of data that they can potentially switch or replay. In particular, an attacker may reuse an older version of the data chunk (*temporal replay*) or the data chunks authenticated with the same key (*spatial replay*). TWEEZER prevents temporal replay against each SSTable by taking advantage of their immutability. By design, an update to an existing key-value pair in an LSM tree does not modify the SSTable. Instead, the new pair is stored in one of the SSTables at a lower level. The new pair is thus authenticated with a different key and the attacker cannot use the older pair to simply roll back the update. In other words, the attacker does not have an older version of key-value pairs authenticated with the same key because TWEEZER has never computed such MACs. We further discuss how TWEEZER prevents spatial replay in §5.3.

The performance improvement of Merkle tree-less authentication comes from the lower EPC usage. The large size of LSM tree makes MAC caching an essential design choice. To avoid a series of MAC computations along the Merkle tree for every SSTable read, the known good MAC for each data block must be cached within EPC. Speicher implicitly makes this design choice by storing the MAC in the footer block of each SSTable that RocksDB keeps within memory, within the EPC when it runs in an enclave (i.e., on Scone [4]). The cost of retaining the MACs remains small when a KVS accommodates a small number of SSTables, but increases quickly as the number of SSTables increases with the size of the KVS. The additional EPC usage for each SSTable varies depending on the configuration, but is roughly about 840 KB for each 64 MB SSTable, when the data block size is 4 KB and value size is 128 B. This roughly becomes a total of 840 MB if the KVS contains about 64 GB of key-value pairs, even assuming that it has no duplicates. This becomes a significant overhead considering the size of EPC, which is either 128 MB or 256 MB. Another option is to compute MACs along the Merkle tree whenever the KVS obtains a new block from storage, but this will significantly increase read latency considering the cost of a typical HMAC computation. The use of per-SSTable key enables TWEEZER to avoid storing too much data in EPC when serving a large KVS, as we show in §7.2 (Figure 10).

5.3 Fine-grained Authentication

TWEEZER authenticates each key-value pair individually to avoid read amplification. An SSTable is composed of many data blocks that are typically as large as 4 KB, containing 3∼28 key-value pairs (Figure 2a). This is a design choice considering the storage devices that are optimized for burst data transfers. Speicher chose this as the unit of encryption and authentication. It computes one MAC for each data block as shown in Figure 2a and stores the result in the SSTable's footer block. However, the encryption and authentication cost makes the inherent read amplification more expensive because the KVS must compute the MAC for the entire data block even when it reads only one key-value pair. In addition, such verified data must reside in EPC to avoid repeating the expensive verification, consuming invaluable EPC space. The block-level decryption and authentication limit the potential location of the block cache to EPC, and this could become a scalability bottleneck when the KVS is to contain large data.

In TWEEZER, we slightly rearrange the data block structure as shown in Figure 2b. This is similar to the key-value separation approach first proposed by Lu et al. [35], but rearranged for fine-grained encryption and authentication. Specifically, each data block is composed of one *value block* and one *key block*. The value block contains all values in the data block along with the corresponding MAC computed from both the key and value. The *key block* contains the sequence of keys along with the offsets of their values in the value block. This restructuring comes with two benefits. First, TWEEZER does not need to verify the freshness of the entire data block to obtain a single key-value pair, reducing read latency. Second, TWEEZER can place the block cache in untrusted memory, which is free from size limitations, because it can directly read a single key-value pair from an encrypted data block.

TWEEZER utilizes the LSM tree's invariant ordering to verify the freshness of the key-value pairs. Each data block in RocksDB's SSTable is composed of a sequence of ordered key-value pairs, and so are the keys in TWEEZER's key block. TWEEZER reads a data block when it fails to find the key in the MemTable or the SSTables at higher levels. In this process, TWEEZER firstly consults the index blocks that it keeps within EPC in plain text. By RocksDB design, each data block B_i in the LSM tree is associated with an index key in the index block, k_i. That is, the keys found in a data block B_i are not smaller than its index key k_i and not larger

Algorithm 1 Key Ordering

Input: i — The index of data block
Output: Returns *true* if the data block satisfies invariant.

```
1:  procedure CHECKORDERING(i)
2:      keyBlock = GetKeyBlock(i)
3:      firstKey = keyBlock.head
4:      lastKey = keyBlock.tail
5:      keyLowerBd = GetIndexKey(i)              ▷ Obtain index key in EPC
6:      keyUpperBd = GetIndexKey(i + 1)
7:      ret = true
8:      if firstKey < keyLowerBound then
9:          ret = false
10:     if lastKey ≥ keyUpperBound then
11:         ret = false
12:     for j in 1 ... keyBlock.length − 2 do
13:                 ▷ for each key in the key block except the first and last
14:         if keyBlock[j − 1] ≥ keyBlock[j] then
15:             ret = false
16:         else if keyBlock[j] ≥ keyBlock[j + 1] then
17:             ret = false
18:         else
19:             continue
20:     return ret
```

than the index key k_{i+1} of the next data block B_{i+1}. Utilizing this, TWEEZER performs a binary search on the index keys to obtain the data block potentially containing the key it is looking for.

To find the key from the obtained data block, TWEEZER decrypts and checks the ordering (Algorithm 1) of its key block, before looking for the key. If found, TWEEZER uses the offset associated with the key to obtain the encrypted value with the MAC. TWEEZER then computes MAC using the SSTable's authentication key, the queried key, and the value. By comparing this with the stored MAC, TWEEZER verifies the freshness of the key-value pair. If TWEEZER fails to find the key, it determines that the key does not exist in the level and moves on to the next level. Although TWEEZER does not perform MAC-based authentication for the key block, the ordering check (Algorithm 1) effectively mitigates any fault attack to deceive TWEEZER that a key does not exist in a data block. As discussed earlier, all keys in a data block must be larger than its index key and smaller than the next block's index key. TWEEZER checks this invariant by comparing the first and last key of a key block with the index keys (line 3–7). Subsequently, it compares each key in the key block with the neighboring keys (line 13–21) to ensure the ordering within the key block. As a result, any attempt to inject a fault to a key block will make TWEEZER interpret the key block as a different list of keys making the list highly unlikely to satisfy the ordering invariant. While the probability of a successful attack is not zero, as data blocks are small and the key space large, its probability will be very low. In more detail, when TWEEZER uses b-byte keys and the difference between the two index keys is D, the chance of a successful attack is roughly as small as $\frac{D}{2^{8\times b}}$. When D is 2^{64} and b is 16, this is about $5.42 \cdot 10^{-20}$ (2^{-64}). Thanks to this invariant-based freshness protection, the cost of reading a key-value pair becomes as small as one decryption of a key block, one decrypting of a value entry, and one MAC computation of a key-value pair. This is roughly $10\times$ smaller than the potential cost of a block-level scheme when a block contains 10 key-value pairs because the MAC operation dominates read performance.

This fine-grained encryption and authentication make it natural to place the block cache outside EPC, which TWEEZER does. This *untrusted block cache* is expected to be beneficial when TWEEZER is to accommodate large data in its LSM tree. By default, the block caches are placed inside EPC and Speicher left this in EPC as well because the loaded block is not encrypted. This does not become a performance bottleneck when a KVS accommodates a small amount of data. However, RocksDB is often configured to have a large block cache in production to serve a large amount of data, and the in-EPC block cache will not scale under this condition. The block caches may still be placed outside EPC, but this will significantly increase the block cache hit latency because every single cache hit triggers a decryption and verification of the whole data block. In contrast to this approach, the fine-grained encryption and authentication approach that TWEEZER takes enables it to take only a small portion of data into EPC from the block that resides in the untrusted memory outside EPC.

5.4 Protecting Logs with Hash Chains

TWEEZER ensures the integrity and freshness of the WAL and MANIFEST log using the classic hash chain [51, 52]. This hash chain is a good fit to protect those two data chunks because both are append-only lists and the freshness verification is performed only upon recovery. When TWEEZER starts to run either from an empty KVS or after recovery, it generates a nonce, considers the nonce as the first MAC (M_0), and creates a cryptographic key for MAC computation. For each new log entry (e_i), TWEEZER concatenates the encrypted data entry with the previous log ($M_{i-1}||E(e_i)$) to compute the next MAC (M_i) and stores it along with the encrypted data. The encrypted key-value pair becomes the data entry for the WAL, and the encrypted new MANIFEST becomes the data entry for the MANIFEST log. Like it does for each SSTable (see §5.2), TWEEZER generated a unique key to protect logs from replay attack. This use of a unique key prevents the attacks from replaying an entire log chain using an older one. The replayed log will be verified using a newer key, which is different from the one used for generating the older chain. Due to the differences in keys, the replayed MACs are not considered genuine ones, and TWEEZER recognizes this as a result of malicious manipulation. This hash chaining sufficiently prevents any attack on the hash chain's integrity and freshness as further discussed in §6.

We chose to use this hash chain for log protection rather than Speicher's mechanism that relies on the trusted counter for two reasons. First, the trusted counter that Speicher relies

on increments only once every 60 ms [6]. This limits the number of new log entries the KVS can create outside EPC to one per 60 ms, which is about 23.4 per second [6]. This is much lower than the expected number of write requests that a KVS is expected to serve. Speicher inevitably delays persisting new key-value pairs to overcome this limitation. In contrast, the hash chain mechanism does not suffer from this limitation. Second, support for the trusted counter on server platforms is not yet stable and its availability varies depending on the system configuration [21]. SGX is designed to use the trusted counter provided by the accompanying *Trusted Platform Module*, but not all server platforms have it. Furthermore, the SGX SDK for Linux does not provide the API as well [28]. TWEEZER's approach using the hash chain is, therefore, a more portable way to protect the logs.

5.5 Root of Trust

TWEEZER binds the confidentiality and integrity of its data to a pair of cryptographic keys and MAC computed from the MANIFEST. TWEEZER users retain these securely (e.g., in a physically isolated local machine) for full protection. TWEEZER uses the cryptographic key and MAC to recover data from the encrypted backup and to verify the backup's freshness. While running, TWEEZER uses these root keys to encrypt and authenticate the MANIFEST log that contains the KVS metadata. The other keys (§4) that TWEEZER uses are kept within the MANIFEST on persistent storage, residing in EPC during run time. This design choice allows TWEEZER to use the keys without significant delay and can later obtain a copy of those keys from the root key pairs and the MANIFEST file when it loads the data from a snapshot.

5.6 Primitive Operations

This section describes how TWEEZER execute the primitive operations for handling the requests.

PUT. TWEEZER handles a PUT request by inserting the key-value pair to WAL for persistence and to MemTable for efficient lookup. The new key-value pair is first encrypted with the dedicated log key, and the resulting data is used for computing a MAC along with the MAC of the previous entry in WAL (see §5.4). The encrypted pair is stored in WAL along with the computed MAC. TWEEZER follows a procedure similar to RocksDB's when it inserts a key-value pair to its MemTable, except for the cryptographic operations. TWEEZER's MemTable is located in both the EPC and untrusted memory, as proposed by Speicher (see §2). TWEEZER finds out the place in the untrusted memory where the value from the new pair will be stored using the internal nodes in EPC and store the encrypted value there. The MAC for the newly stored value is kept within EPC for verification of authenticity later.

GET. Upon receiving a GET request that accompanies a key, TWEEZER first looks up the MemTable within the EPC to determine if the key exists in the MemTable. If the key is found, TWEEZER obtains the encrypted value from the untrusted memory and MAC from EPC. The obtained MAC is then compared with the expected one computed using the key kept in EPC. Only if the stored MAC matches the computed one does TWEEZER consider the obtained value as a genuine one and respond to the request with it. If the key is not found from the MemTable, TWEEZER traverses the LSM tree as RocksDB does to find the pair with the requested key or determine that the key does not exist. TWEEZER finds an SSTable that is likely to contain the requested key like unmodified RocksDB, from the lowest level of the LSM tree, using the filter blocks and index blocks cached in EPC, with the sanity checks described in §5.3. From the data block, TWEEZER obtains the key block containing all keys, decrypts it in EPC, and finds the requested key. TWEEZER continues to the next level of the LSM tree if it fails to find the key from the key block. Otherwise, if the key is found, TWEEZER speculates that the key-value pair is stored in the current data block and obtains the encrypted key-value pair along with its MAC from the value block. TWEEZER verifies the obtained pair's authenticity using the authentication key for the SSTable (see §5.2), and responds to the client with the value if the computed MAC matches the stored one.

Range. As in RocksDB, TWEEZER handles range queries by first creating iterators and then traversing the data blocks in multiple levels. TWEEZER finds the starting key and initializes the iterators on each level by performing the same operations for handling GET requests. For each traversal, TWEEZER determines the latest version of the key-value pair like RocksDB, by checking the MemTable and then the LSM tree. If the key exists in MemTable, TWEEZER verifies its authenticity and decrypts the value as it does to handle a GET request. The case where TWEEZER finds the key-value pair from the LSM tree is also handled similarly, and TWEEZER verifies the absence of a key at a certain level as described in §5.3.

Recovery. TWEEZER follows the same recovery scheme that RocksDB implements, with the additional decryption and verification using the pre-shared credentials (i.e., keys and MAC). For this, TWEEZER takes the credentials as inputs in addition to the files constituting the KVS. The first piece of data that TWEEZER decrypts and verifies are the MANIFEST logs as discussed earlier (§5.5). As a result, TWEEZER obtains the latest MANIFEST that contains the structure of TWEEZER across the files and cryptographic keys needed to decrypt and verify the rest of the data chunks. In particular, TWEEZER obtains these keys from the recovered and verified MANIFEST update log called *version edit*. Each version edit contains the changes made to the KVS structure such as SSTable creation, SSTable deletion, log entry creation or log entry deletion. TWEEZER extends these records with the

additional keys that it uses such as the per-SSTable keys or log key. Aside from the additional decryption or verification steps, recovery is done following RocksDB's scheme. After recovery, TWEEZER provides the remote user with heartbeat data that represents the exact version of the snapshot (see §6).

6 Security Analysis

This section discusses in-depth about how TWEEZER ensures the integrity and freshness of its data against potential attacks.

Replay Attack. Reuse of existing encrypted data-MAC pairs is a common attack strategy against data integrity. To TWEEZER, this is the only way for attackers to pass the MAC-based verification procedure. Under our threat model the attackers can obtain these data-MAC pairs stored outside EPC because they are assumed to have full access to the memory content outside the EPC as well as the storage content. With this strong capability, an attacker may aim to replay TWEEZER's data chunks such as MANIFEST log, WAL, a whole SSTable, or individual key-value pairs.

Log Replay. The first two targets, MANIFEST log and WAL, are protected by the hash chain. TWEEZER and the remote user are assumed to have the key pairs for encryption and MAC computation along with the nonce that TWEEZER uses as the first hash. When TWEEZER recovers from a snapshot, TWEEZER correctly determines if each log entry is a replayed block or not through the MAC verification for the following reason. To replay the ith block b_i from the list of log entries $b_0, \cdots, b_n$ and pass the verification procedure, the attacker must generate or obtain MAC M'_i computed from $h_{i-1}||b'_i$ using the correct MAC key, where b'_i is the replayed block and h_{i-1} is the correct MAC of the previous (i.e., $(i-1)$th) block. However, the attacker cannot obtain such M'_i because of the uniqueness of the MAC key and the blocks in the log. The only data chunks with the corresponding MAC computed using the MAC key are the log entries. Therefore, the attacker can only choose one from $b_0, \cdots, b_n$ as the b'_i. If the attacker chooses b_j as b'_i, the only MAC available to the attacker is the one computed from $h_{j-1}||b_j$, which does not pass the verification procedure because $h_{j-1} \neq h_{i-1}$ when $j \neq i$.

Key-Value Pair Replay. TWEEZER recognizes any attack against the latter two (a whole SSTable and an individual key-value pair) when it verifies their freshness using the MAC. An attacker's strategy in this scenario can be classified into three groups. First, the attacker may try to replace one SSTable as a whole with another. TWEEZER detects this attempt when it obtains data blocks from the SSTable and verifies the block through MAC computation. Similar to the earlier scenario, the attacker cannot obtain the appropriate MAC because the key-value pair that the attacker aims to replay has never been used to compute a MAC with the target SSTable's key. Each SSTable is authenticated with its unique key, so the MACs associated with key-value pairs in another SSTable are considered incorrect by the verification procedure. Second, an attacker could try to replay data chunks within one SSTable. TWEEZER recognizes this using the invariant ordering of SSTables in an LSM tree as discussed in §5.3. If the replay is somehow performed within a key block, the attacker inevitably breaks the ordering. If the replay switches two keys k_1 and k_2 and k_1 is to appear earlier than k_2, the replay makes k_2 appear earlier than k_1, breaking the invariant ordering. Duplicating a key is not an option as well because it breaks the uniqueness principle. The last strategy that the attacker can choose is to replay across the key blocks within an SSTable, but it violates the property of the index key, which partitions the set of keys an SSTable contains into contiguous and mutually disjoint ranges.

Rollback Attack. TWEEZER ensures that the user has the latest version of its data at the granularity of heartbeat transactions. A strong attacker that we assume may place a rollback attack where they take a snapshot of TWEEZER's data at some point and later present to TWEEZER or its remote user as the genuine and latest version. The online rollback attack that an attacker performs while TWEEZER is running is infeasible because the attacker cannot replace the data stored within EPC. An offline attack in which the attacker replaces TWEEZER's files with an older version could, however, be a realistic threat. To thwart such attacks, TWEEZER relies on periodic interaction with the user to timestamp the versions by periodically issuing a write transaction to TWEEZER. Later, these resulting key-value pairs are used to determine the TWEEZER snapshot version. These additional timestamps provide rollback-resilience because the other verification mechanisms prevent the attacker from forging a fake snapshot. When given a snapshot to recover from, TWEEZER and its user verify its freshness using the root key pairs, starting from the MANIFEST log. As discussed in §5.6, an attacker who does not have these key pairs cannot make any modifications to any older TWEEZER snapshot version. The only remaining option is to present an exact copy of an older version, but the user correctly determines the copy's version from the key-value pairs from the heartbeat transactions after the verified recovery.

Existence Attack. TWEEZER also detects any attempt to deceive it into believing that an SSTable does not contain a particular key when, in fact, the SSTable has it. The LSM tree design strengthens this attack if successful because TWEEZER may consider an older version of the key-value pair found in a lower level. The LSM tree-based KVSs handle update requests by adding the new key-value pair to the higher level of the LSM tree and leave the older one in a lower level. An attacker performing the existence attack must first find the key block that contains the victim key and forge a valid key block passing TWEEZER's check. The confidentiality that TWEEZER ensures using encryption prevents this first step, which leaves an attack to an unknown key as the only remaining option. However, this option is also highly unlikely because of TWEEZER's invariant check, as discussed in §5.3.

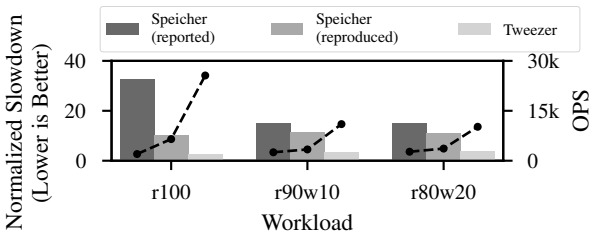


Figure 3: The normalized slowdown of TWEEZER and reproduced Speicher, along with the reported slowdown of Speicher [6], where the unmodified RocksDB is the baseline. The absolute throughput is also presented as lines using the y axis on the right.

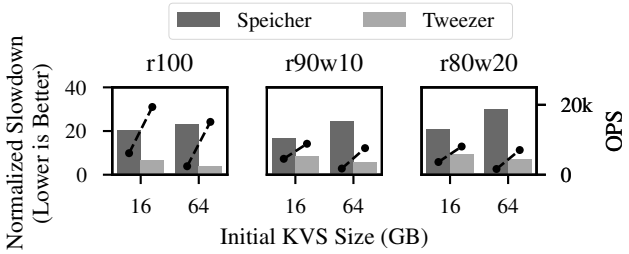


Figure 4: Normalized slowdown (lower is better) relative to the original RocksDB on SGX of TWEEZER.

7 Performance Evaluation

Environment. We evaluate the performance of TWEEZER on a machine with Intel Xeon E-2288G and 64GB of DRAM. The CPU has 32 KB instruction and data caches, 256 KB of L2 caches and a 16 MB shared L3 cache. The CPU also implements Intel SGX for confidential computing and AES-NI to speed up AES block cipher. The system runs Ubuntu 18.04 with Linux Kernel 4.15. For every cryptographic operation, we used OpenSSL 1.1.1i. Specifically, we chose AES GCM 256 as the block cipher scheme to protect the confidentiality of the data, GHASH to compute MACs for the logs and MemTable, and HMAC with SHA3-384 to compute MACs for SSTables. We followed the schemes that Speicher used to rule out the performance impact of cryptographic schemes when we compare TWEEZER and Speicher. Note that, unlike the encryption or GHASH, the HMAC computation does not benefit from hardware acceleration because the CPU that we use does not have hardware extensions to accelerate for SHA computation. We built both TWEEZER and the reproduced Speicher based on RocksDB version 6.14.

Benchmarks. We evaluate TWEEZER using `db_bench` with three workloads, r100, r90w10, r80w20 each of which is composed of 100% reads; 90% reads and 10% writes; and 80% reads and 20% writes; respectively. The key size is 16 B, the SSTable size is 64 MB, and 5 million key-value pairs were used, as done in Speicher [6]. The block size is either 4 KB, which is the default of RocksDB, or 32 KB, what Speicher used for its evaluation. In some experiments, we use `db_bench` to create KVSs as large as 16 GB and 64 GB, and use them to evaluate the performance of TWEEZER on a practical setup.

Reproducing Speicher. For comparison, we reproduced Speicher by extending RocksDB because Speicher is not open-sourced. As discussed in §1, TWEEZER adopts some of Speicher's design decisions to save EPC space and relies on Scone for asynchronous system calls. As such, the reproduced Speicher shares these aspects with TWEEZER in our implementations. Figure 3 shows the normalized throughput of TWEEZER, the reproduced Speicher, and the original Speicher as bars, and the absolute throughput as lines. Hereafter, all normalized results are normalized to the baseline RocksDB, presented with the absolute throughput. The experimental results advocate that our reproduction of Speicher is reasonable in that it exhibits similar or better performance characteristics compared with the reported number. In this experiment, we issue 5 million transactions starting from a KVS filled with 5 million entries, set the value size to 1024 B, and set the block size to 32 KB to replicate the experiments as close to those of the original setting [6]. We note that the difference in experimental setup may have also contributed to the better performance that replicated Speicher exhibits compared to the original. Speicher was evaluated with on a machine with Xeon E3-1270 v5, which has smaller (8 MB) shared L3 cache and smaller EPC (128 MB), albeit the size of main memory is the same. Larger EPC and caches potentially reduce the number of cryptographic operations while Speicher runs, reducing the overhead of storing data within EPC. Regarding the results, we observe that TWEEZER outperforms Speicher by 1.91∼3.94× despite the fact that the KVSs run with smaller amount of data. The 5 million entries are actually small enough that EPC paging does not occur, favoring Speicher considerably.

7.1 Throughput

Point Lookups. Figure 4 shows the normalized throughput of TWEEZER and Speicher on three workloads from `db_bench` with varying initial KVS sizes and 1024B values. The block size is set to 4 KB, which is the default and the best for the original RocksDB. Starting from the KVS images that we created using `db_bench`, we issue 5 million transactions to measure the throughput. Under the tests using these large KVSs, TWEEZER consistently outperforms Speicher by 1.94∼6.23×, reducing the slowdown from 16∼30× to 4∼9×. Our observation (§7.2) suggests that this performance gap is primarily due to EPC paging. As the KVS size increases, Speicher's footer cache in EPC becomes larger and causes frequent EPC paging. The use of per-SSTable keys reduces the amount of data that must be kept within EPC, enabling TWEEZER to avoid the frequent EPC paging.

Range Query. We evaluate the range query performance

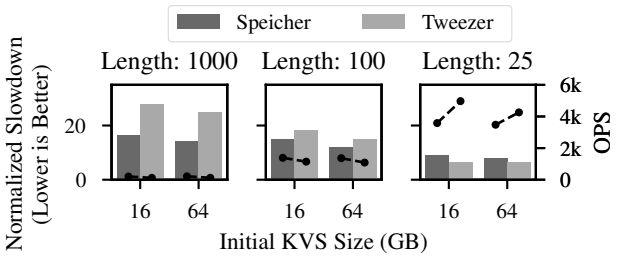


Figure 5: Normalized throughput of TWEEZER and Speicher on range queries with varying length. Length refers to the number of key-value pairs being accessed for each range query.

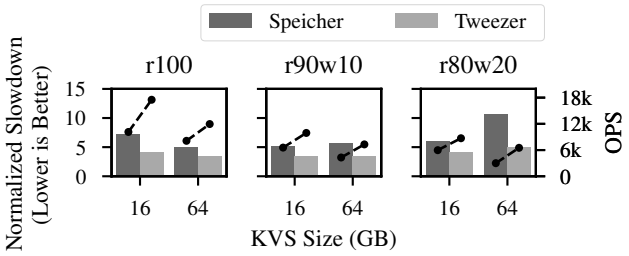


Figure 6: Normalized performance of TWEEZER and Speicher with large (32 KB) data blocks.

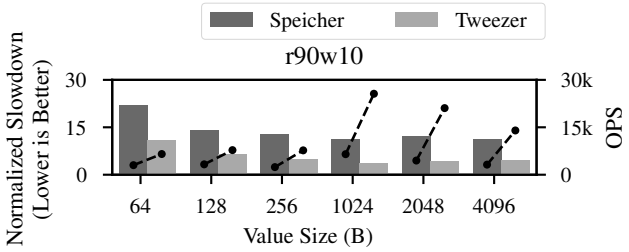


Figure 7: The normalized performance of TWEEZER and Speicher when running for different value sizes.

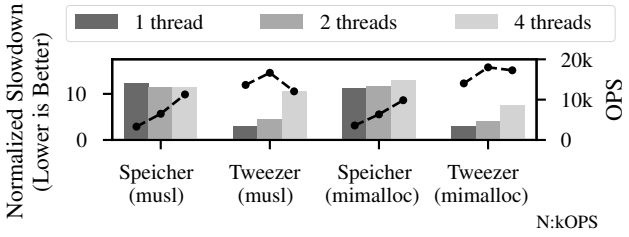


Figure 8: Normalized throughput of TWEEZER and Speicher with two different memory allocators, musl and mimalloc, as the number of threads increases.

of TWEEZER and Speicher using `seekrandom` benchmark in `db_bench`, with 1024 B values and 32 KB data blocks. As Figure 5 shows, TWEEZER exhibits higher throughput than Speicher for short queries, but the advantage diminishes as the query length increases. This result is due to the fine-grained authentication (§5.3) that is optimized only for point lookups. When obtaining a key-value pair, the cryptographic cost is smaller in TWEEZER than Speicher that decrypts and authenticates a whole data block even for a single request. However, this whole-block decryption and authentication become less costly when handling range queries because Speicher decrypts and authenticates the block only once for multiple pairs. Unlike this, TWEEZER has no choice but to handle range queries like a sequence of point lookups, thus authenticating the key-value pairs separately.

Block Sizes. The data block size in an SSTable can be configured and may affect throughput. While our experiments on point lookup performance used the default value of 4 KB as this setting results in the best performance for the baseline RocksDB, Speicher, in their experiments, used 32 KB blocks. Thus, we perform the same experiments obtained for Figure 4 except with the block size set to 32 KB. From the results in Figure 6, we observe that Speicher performance improves considerably due to the reduction in EPC usage (see §7.2). Despite this, we see that TWEEZER still outperforms Speicher by 1.46∼2.17×.

Value Sizes. Figure 7 shows how the value size affects the performance of TWEEZER and Speicher. For these experiments, we used the same setup as the comparison study in Figure 3 except for the value sizes. Overall, we observe that TWEEZER outperforms Speicher by 1.82∼4.70× in all configurations because TWEEZER also, in part, suffers from read amplification, which diminishes as the value size increases, resulting in reduced slowdown.

Number of Threads. Figure 8 shows the normalized throughput of TWEEZER and Speicher as the number of threads increases. Speicher scales similarly to RocksDB, but TWEEZER's slowdown increases as the number of thread increases. According to our analysis, this is due to the default heap allocator of Scone, musl [49] that does not scale as the number of threads increases. TWEEZER's shows scalability when we replace musl with mi-malloc [39] but the benefit was limited because Scone [55] does not support the thread local storage model that mi-malloc uses. Nevertheless, TWEEZER outperforms Speicher by 1.78× when running with 4 threads.

Untrusted Block Cache. Fine-grained authentication (§5.3) enables TWEEZER to place the block cache in untrusted memory, outside EPC. Having its block cache outside the EPC is beneficial when TWEEZER starts to serve a large KVS in which larger block cache could help reduce the average read latency. Figure 9 presents the normalized throughput of TWEEZER and Speicher as we change the block cache sizes from 8 MB (default) to 128 MB and 256 MB using the same setup as the comparison study in Figure 3. Speicher's performance overhead increases as the block cache size increases because the additional block caches cause more EPC paging. Speicher places all block cache content in EPC as it does not make any adjustment to the block cache management, increasing EPC usage and resulting in more EPC paging. On the contrary, TWEEZER does not suffer from the increased

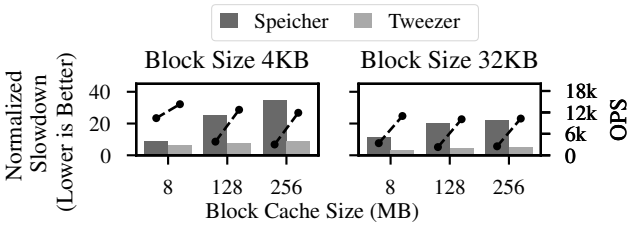


Figure 9: Normalized throughput of TWEEZER and Speicher as the block cache size increases.

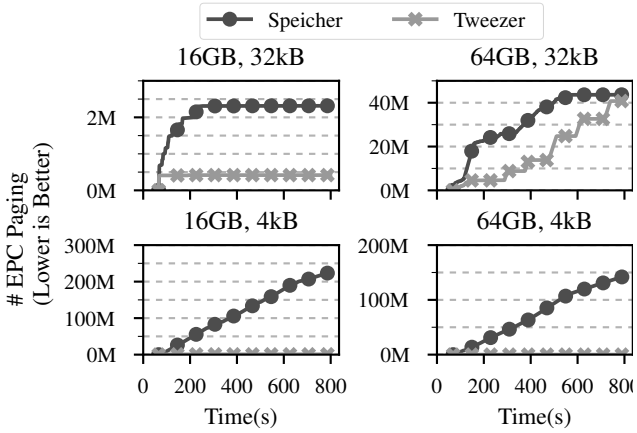


Figure 10: Cumulative number of EPC paging while running TWEEZER and Speicher with varying block sizes (4 KB and 32 KB) and initial KVS sizes (16 GB and 64 GB).

number of EPC paging because it places the block cache outside the EPC with the same cryptographic protection as the blocks in the SSTables. TWEEZER still does not benefit from the block caches as the absolute numbers show, however, due to the relatively small benefit that the block cache brings to TWEEZER compared to the unmodified RocksDB. Block cache miss penalty is high in unmodified RocksDB because it decompresses the retrieved data block on cache misses. The cache hit latency is long on TWEEZER, which additionally decrypts and authenticates the retrieved pairs.

7.2 EPC Usage

EPC Paging. We obtained the number of EPC paging using `sgxtop` [32]. Figure 10 shows the cumulative number of EPC paging observed while running the two configurations of the benchmarks used for the experiment in §7.1. Specifically, we accumulated all observed EPC paging from each run after the recovery because neither TWEEZER nor Speicher is designed to optimize the recovery phase and both experience a large number of EPC paging. When the block size is configured to 4 KB (the bottom two in Figure 10), which is the default and exhibits better performance for TWEEZER, Speicher suffer up to 430× more EPC paging, due to the cached MACs in the footer blocks. On the contrary, when we configure the block

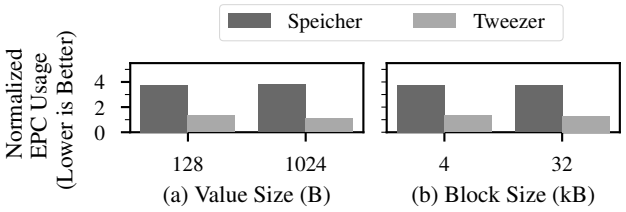


Figure 11: Normalized table cache sizes while running TWEEZER and Speicher with the r90w10 workload.

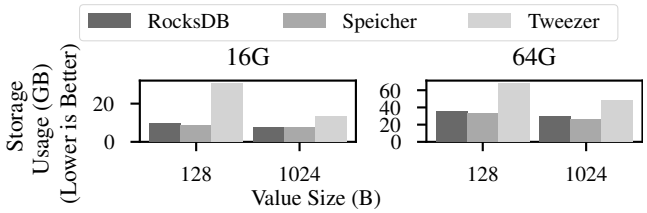


Figure 12: The amount of data stored in file system when we store the same number of key-value pairs.

size to be 32 KB, we observed that the cumulative number of EPC paging that TWEEZER experiences approaches that of Speicher, even though TWEEZER still outperforms Speicher in terms of throughput (§7.1). According to our analysis, this additional EPC paging comes from background compaction. Compared with Speicher, we observe certain periods in time in which TWEEZER's cumulative EPC paging suddenly increases. This is due to the additional memory consumption by the background compaction, which also uses EPC space to process decrypted blocks. It is worth noting that these EPC paging numbers were obtained from the runs reported in Figure 6. That is, TWEEZER still shows much higher performance albeit the EPC pagings due to the compaction. This is because the compaction does not usually block the transaction processing. TWEEZER's fine-grained authentication (§5.3) could enable compaction with encrypted SSTables and reduce these peaks, but we leave it as future work.

Amount of Data in EPC. The amount of data in EPC is another measure that shows the potential density of EPC paging over time. Programs using more EPC are likely to experience more EPC misses and longer EPC access time on average. To compare the amount of data that TWEEZER and Speicher store in EPC, we measure the size of the table cache that contains metadata for SSTables and resides in memory. The size of the table cache is a good estimate of the amount of data in EPC because the table cache is the largest component in EPC by design. Figure 11 shows the results for the workload with 90% reads as we vary the values sizes and block sizes. We observe that Speicher's table cache is 3.71∼4.17× larger compared to that of RocksDB while that of TWEEZER's is only 1.08∼1.35× larger. In other words, Speicher uses a table cache that is 2.84∼3.08× larger compared to TWEEZER. This shows that our design choice of using per-SSTable key (see

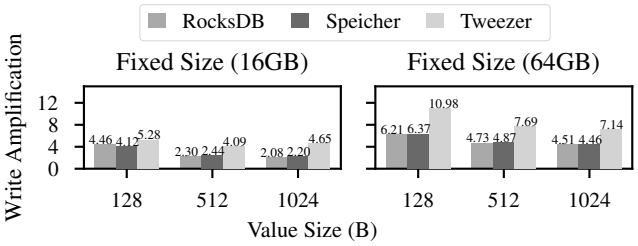


Figure 13: The amount of data that RocksDB, Speicher, and TWEEZER write to storage as the value size changes, normalized to the total size of key-value pairs that they received.

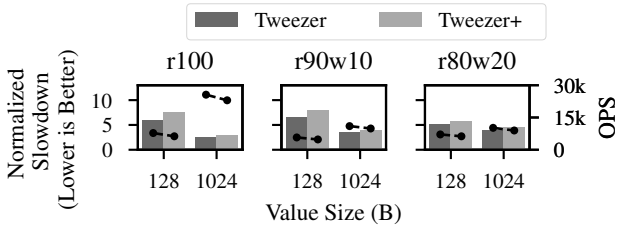


Figure 14: Throughput of TWEEZER without (TWEEZER) and with (TWEEZER+) the strong key block authentication.

§5.2) helps reduce the amount of data in EPC, contributing to reduced EPC paging.

7.3 Storage Blowup

Increased Storage Usage. One drawback of fine-grained authentication is the increase in storage usage due to the less productive compression after encryption and individual authentication for each key-value pair. As discussed in §4, TWEEZER's fine-grained authentication renders RocksDB's block compression less effective because the data is encrypted before compression, unless TWEEZER employs a specially crafted encryption and compression scheme [29, 53]. To understand this drawback quantitatively, we evaluate the corresponding storage cost by measuring the size of aggregated SSTables, with compression, constituting the KVS in varying configurations used for the evaluations in §7.1. Figure 12 shows the results, and we see that TWEEZER experiences 1.77∼3.45× storage overhead. This overhead in size increases as the value size decreases because the MAC size remains the same for each key-value pair.

Write Amplification. We also measure the amount of data that Speicher and TWEEZER write to storage and Figure 13 shows the result. We normalized the amount of written data to the number of key-value pairs that each KVS accommodates to compare their impact on write amplification. As expected, write amplification decreases as the value size increases when running unmodified RocksDB or Speicher because the amount of metadata is proportional to the number of entries. When the total size of key-value pairs is fixed, they write less metadata because they store fewer entries as the value size increases. The write amplification of TWEEZER also decreases, but much less than the other two. We presume that this is primarily due to the entropy of data in data blocks. Unlike Speicher, TWEEZER encrypts data blocks before compression, rendering compression less effective. On the evaluation with 16 GB KVS, write amplification even increases when the value size increases from 512 B to 1024 B.

8 Discussion

Data Obliviousness. TWEEZER is not designed to be data-oblivious. That is, TWEEZER is not provably immune to side-channel leakage through data-dependent access patterns. For example, attackers could learn the following information. By observing the changes of encrypted values in MemTable stored outside the EPC, attackers could learn that a write request was made and handled. However, cryptographic protection prevents the attacker from revealing or faulting the content. The relationship between leaves are also under protection in that the internal nodes are stored within EPC. Only a successful side-channel attack against the enclave [10, 33, 48, 66] could reveal such a relationship. Access patterns within an SSTable reveals relationships between the queried key and the index keys. TWEEZER does not shuffle the data blocks in an SSTable, and an attacker can determine from which data block TWEEZER found the queried key through the access pattern. Combining these two, an attacker can infer the likely range of the queried key, for example, how many keys in the SSTable would be larger than the queried key. TWEEZER could mitigate this inference by shuffling the data blocks.

Larger KVS. Our study shows that TWEEZER needs to be tailored further to have better efficiency in EPC usage when the KVS size becomes larger. TWEEZER introduces much smaller amount of additional in-memory data that must be held in EPC compared with Speicher. However, the amount of data that TWEEZER holds in EPC still increases as the KVS size increases because of some data (e.g. index block) that TWEEZER still caches within EPC, as an extension of RocksDB. We leave the optimization of RocksDB metadata to further reduce this EPC usage as future work.

Key Block Freshness. As discussed in §5.3, TWEEZER does not provably prevent the fault attack against the key block, although it is highly unlikely for an attacker to successfully perform the attack. This is due to the lack of MAC-based verification of the key blocks. As an alternative design choice, TWEEZER can be strengthened by computing and verifying MAC for the key blocks as well. Figure 14 is the result of our experiments showing the performance overhead of this design choice. As expected, the additional authentication does incur performance overhead (TWEEZER+). TWEEZER+ is

11%∼24% slower than TWEEZER depending on workloads and value sizes. The overhead increases with smaller value size because the key block size increases as the value size decreases.

9 Related work

This work is closely related to existing attempts to tailor various important applications to Intel SGX [3, 6, 12, 16, 23, 31, 54, 56, 61] as well as research on securing database systems including KVSs [15, 45].

Running Unmodified Applications on SGX. Haven [7, 8], SCONE [4], Graphene-SGX [64], Panoply [57], SGX-LKL [46] are systems designed to help unmodified applications to run on an enclave. As suggested by the authors, these enabled us to quickly work on tailoring a persistent KVS for Intel SGX. In particular, TWEEZER has been implemented and tested on SCONE. However, it is worth noting that TWEEZER can run on any of the aforementioned systems because TWEEZER does not make any assumptions on features unique to SCONE.

Persistent KVSs on SGX. As we have repeatedly discussed, Speicher [6] is the closest to our work in that it is designed to boost the performance of a persistent KVS on SGX, taking RocksDB as an example. Speicher contributes three new design features to achieve this goal, but fails to scale to large KVSs. While TWEEZER adopts many of the ideas proposed by Speicher, we propose a new message authentication scheme and restructures the data block to alleviate the scalability issue. Furthermore, TWEEZER uses a hash chain mechanism to protect persistent logs allowing for a solution that is not bound to platforms that support trusted counters. Enclage [61] is also close to TWEEZER in that it is designed to be an SGX-based secure storage engine but does not take integrity protection into account.

In-memory KVSs on SGX. ShieldStore [31] studies the design options to adapt an in-memory KVS for SGX. Compared with TWEEZER, ShieldStore is designed for in-memory KVSs and still relies on the Merkle tree for freshness. Similar to ShieldStore, EnclaveCache [12] and Avocado [5] are also designed to use SGX to protect in-memory KVSs.

Cryptographic Approaches. CryptDB is one of the pioneering systems in which unmodified database queries are proxied and handled by encrypted backend [45]. CryptDB adopts various cryptographic schemes including homomorphic encryption [20] and focuses on confidentiality guarantees. Dory goes beyond confidentiality guarantees and mitigates access pattern-based leakage, providing authenticity relying on distributed trust [15]. TWEEZER tackles the same problem at a lower level compared with these approaches in that many relational database systems use RocksDB-like persistent KVSs as storage engines. One weakness of TWEEZER, when compared to Dory, is the lack of data obliviousness. To overcome this, TWEEZER has to be strengthened with oblivious search indices [40] or file system operations [2].

Log Protection. Protection of logs from rollback attacks have long been an important problem. One of the well-known mechanisms is the hash chain [44, 51, 52] that TWEEZER adopts to protect the WAL and MANIFEST logs. However, the hash chain cannot guarantee freshness against potential rollback attacks across crashes and recoveries as discussed in Memoir [44]. Memoir overcomes this limitation and relies on local trusted non-volatile memory. Verena [30] also addresses a similar problem by using a hash server. Compared with these, TWEEZER's approach is similar to Verena in that it relies on the user, who sends heartbeat transactions to timestamp versions. ROTE is designed solely to address this weakness of requiring a trusted component to defeat the rollback attack by using multiple enclaves [37]. TWEEZER can adopt this to provide rollback resilience without relying on the heartbeat packets.

10 Conclusion

This paper presented TWEEZER, an LSM tree-based persistent key-value store tailored for confidential computing by taking advantage of the LSM tree design principles. The unique invariants that the LSM tree introduces, being a data structure optimized for storage devices, enables TWEEZER to avoid constructing a large Merkle tree to protect the integrity and freshness of the key-value pairs. Our experiments with the implementation of TWEEZER and a reproduction of a pioneering work, Speicher, shows that this new MAC scheme for the LSM tree brings considerable performance benefits. Our implementation of TWEEZER outperforms Speicher on point lookups (e.g., by 1.91∼6.23×) in all evaluation settings, and in particular, the ones with large (16∼64 GB) KVSs. We anticipate that our findings and open-sourced implementation from this work will motivate further improvements in this direction to secure our data on these key-value stores.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Patrick P. C. Lee, for their constructive reviews and comments. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2018-0-00503, Researches on next generation memory-centric computing system architecture), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2021-0-01817, Development of Next-Generation Computing Techniques for Hyper-Composable Datacenters), and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2021R1F1A1050311).

Availablity

TWEEZER is available on https://github.com/cssl-unist/tweezer.

References

[1] The storage performance development kit (spdk). https://spdk.io/.

[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[3] 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*, pages 799–812, 2020.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. Scone: Secure linux containers with intel sgx. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[5] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. Avocado: A secure in-memory distributed storage system. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, July 2021.

[6] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. Speicher: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST)*, February 2019.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3), August 2015.

[9] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, pages 1–13, 2016.

[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[11] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyperthreading side channels on sgx with contrived data races. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 178–194. IEEE, 2018.

[12] Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. Enclavecache: A secure and scalable key-value cache in multi-tenant clouds using intel sgx. In *Proceedings of the 20th International Middleware Conference*, pages 14–27, 2019.

[13] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18, 2017.

[14] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[15] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. Dory: An encrypted search system with distributed trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1101–1119, 2020.

[16] Susanne Felsen, Ágnes Kiss, Thomas Schneider, and Christian Weinert. Secure and private function evaluation with intel sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 165–181, 2019.

[17] The Apache Software Foundation. Kafka streams. https://kafka.apache.org/.

[18] Uber San Francisco. Uber. https://www.uber.com/.

[19] Blaise Gassend, E Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of ninth international symposium on high performance computer architecture*, 2003.

[20] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[21] Greg. Platform service enclave and me for intel xeon server. https://community.intel.com/t5/Intel-Software-Guard-Extensions/Platform-Service-Enclave-and-ME-for-Intel-Xeon-Server/td-p/1173098.

[22] Shay Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptol. ePrint Arch.*, 2016:204, 2016.

[23] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 99–105, 2017.

[24] Intel. 10th generation intel core processor famillies datasheet volume 1. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/10th-gen-core-families-datasheet-vol-1-datasheet.pdf.

[25] Intel. Enclave memory measurement tool for intel software guard extensions (intel sgx) enclaves. https://software.intel.com/content/dam/develop/external/us/en/documents/enclave-measurement-tool-intel-sgx-737361.pdf.

[26] Intel. Intel sgx. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.

[27] Intel. The intel sgx memory encryption engine. https://software.intel.com/content/www/us/en/develop/blogs/memory-encryption-an-intel-sgx-underpinning-technology.html.

[28] Intel. Unable to find alternatives to monotonic counter application programming interfaces (apis) in intel software guard extensions (intel sgx) for linux* to prevent sealing rollback attacks. https://www.intel.com/content/www/us/en/support/articles/000057968/software/intel-security-products.html.

[29] M. Johnson, P. Ishwar, V. Prabhakaran, D. Schonberg, and K. Ramchandran. On compressing encrypted data. *IEEE Transactions on Signal Processing*, 52(10):2992–3006, 2004.

[30] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-end integrity protection for web applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[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*, pages 1–15, 2019.

[32] Kevin Lahey. Monitoring intel sgx enclaves. https://fortanix.com/blog/2020/02/monitoring-intel-sgx-enclaves/.

[33] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, August 2017.

[34] Percona LLC. Mongorocks. https://www.percona.com/doc/percona-server-for-mongodb/3.4/mongorocks.html.

[35] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2016.

[36] MariaDB. Getting started with myrocks. https://mariadb.com/kb/en/getting-started-with-myrocks/.

[37] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. {ROTE}: Rollback protection for trusted execution. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1289–1306, 2017.

[38] Ines Messadi, Shivananda Neumann, Lennart Almstedt, and Rüdiger Kapitza. A fast and secure key-value service using hardware enclaves. In *Proceedings of the 20th International Middleware Conference Demos and Posters*, pages 1–2, 2019.

[39] Microsoft. mi-malloc. https://microsoft.github.io/mimalloc/.

[40] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[41] Netflix. Netflix. https://www.netflix.com/.

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

[43] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, RS, April 2017.

[44] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings*

of the 32nd IEEE Symposium on Security and Privacy (Oakland), Oakland, CA, May 2011.

[45] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 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. *arXiv preprint arXiv:1908.11143*, 2019.

[47] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[48] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, May 2021.

[49] et al. Rich Felker. musl libc. https://musl.libc.org/.

[50] Apache Samza. Apache samza. http://samza.apache.org/.

[51] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th USENIX Security Symposium (Security)*, San Antonio, TX, January 1998.

[52] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1999.

[53] Daniel Schonberg, Stark C. Draper, and Kannan Ramchandran. On blind compression of encrypted data approaching the source entropy rate. In *2005 13th European Signal Processing Conference*, pages 1–4, 2005.

[54] 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 *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[55] Scone. Scone sgx toolchain. https://sconedocs.github.io/SCONE$_{t}$oolchain/.

[56] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 1st ACM International Workshop on Security in SDN and NFV*, New Orleans, LA, March 2016.

[57] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*, 2017.

[58] Facebook Open Source. Rocksdb. https://rocksdb.org/, n.d.

[59] Facebook Open Source. A rocksdb storage engine with mysql. http://myrocks.io/, n.d.

[60] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure integrated circuits and systems*, pages 27–42. Springer, 2010.

[61] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. Building enclave-native storage engines for practical encrypted databases. *Proceedings of the VLDB Endowment*, 14(6):1019–1032, 2021.

[62] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. rkt-io: a direct i/o stack for shielded execution. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 490–506, 2021.

[63] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[64] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 645–658, 2017.

[65] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[66] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Breaking virtual memory protection and the sgx ecosystem with foreshadow. *IEEE Micro*, 39(3):66–74, 2019.

[67] Jinwen Wang, Yueqiang Cheng, Qi Li, and Yong Jiang. Interface-based side channel attack against intel sgx. *arXiv preprint arXiv:1811.05378*, 2018.

[68] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017.

[69] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. Ac-key: Adaptive caching for lsm-based key-value stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, July 2020.