usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution

Maurice Bailleu, Jörg Thalheim, and Pramod Bhatotia, *The University of Edinburgh;* Christof Fetzer, *TU Dresden;* Michio Honda, *NEC Labs;* Kapil Vaswani, *Microsoft Research*

https://www.usenix.org/conference/fast19/presentation/bailleu

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Open access to the Proceedings of the
17th USENIX Conference on File and
Storage Technologies (FAST '19)
is sponsored by

NetApp®

SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution

Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia
The University of Edinburgh

Christof Fetzer[†], Michio Honda[‡], Kapil Vaswani[*]
[†]*TU Dresden,* [‡]*NEC Labs,* [*]*Microsoft Research*

Abstract

We introduce SPEICHER, a secure storage system that not only provides strong confidentiality and integrity properties, but also ensures data freshness to protect against rollback/forking attacks. SPEICHER exports a Key-Value (KV) interface backed by Log-Structured Merge Tree (LSM) for supporting secure data storage and query operations. SPEICHER enforces these security properties on an untrusted host by leveraging shielded execution based on a hardware-assisted trusted execution environment (TEE)—specifically, Intel SGX. However, the design of SPEICHER extends the trust in shielded execution beyond the secure SGX enclave memory region to ensure that the security properties are also preserved in the stateful (or non-volatile) setting of an untrusted storage medium, including system crash, reboot, or migration.

More specifically, we have designed an authenticated and confidentiality-preserving LSM data structure. We have further hardened the LSM data structure to ensure data freshness by designing asynchronous trusted counters. Lastly, we designed a direct I/O library for shielded execution based on Intel SPDK to overcome the I/O bottlenecks in the SGX enclave. We have implemented SPEICHER as a fully-functional storage system by extending RocksDB, and evaluated its performance using the RocksDB benchmark. Our experimental evaluation shows that SPEICHER incurs reasonable overheads for providing strong security guarantees, while keeping the trusted computing base (TCB) small.

1 Introduction

With the growth in cloud computing adoption, online data stored in data centers is growing at an ever increasing rate [11]. Modern online services ubiquitously use persistent key-value (KV) storage systems to store data with a high degree of reliability and performance [39, 65]. Therefore, persistent KV stores have become a fundamental part of the cloud infrastructure.

At the same time, the risks of security violations in storage systems have increased significantly for the third-party cloud computing infrastructure [66]. In an untrusted environment, an attacker can compromise the security properties of the stored data and query operations. In fact, many studies show that software bugs, configuration errors, and security vulnerabilities pose a serious threat to storage systems [9, 12, 16, 20, 24, 35, 37].

However, securing a storage system is quite challenging because modern storage systems are quite complex [9, 49, 64, 72]. For instance, a persistent KV store based on the Log-Structured Merge Tree (LSM) data structure [54] is composed of multiple software layers to enable a data path to the storage persistence layer. Thereby, the enforcement of security policies needs to be carried out by various layers in the system stack, which could expose the data to security vulnerabilities. Furthermore, since the data is stored outside the control of the data owner, the third-party storage platform provides an additional attack vector. The clients currently have limited support to verify whether the third-party operator, even with good intentions, can handle the data with the stated security guarantees.

In this landscape, the advancements in trusted execution environments (TEEs), such as Intel SGX [4] or ARM TrustZone [7], provide an appealing approach to build secure systems. In fact, given the importance of security threats in the cloud, there is a recent surge in leveraging TEEs for shielded execution of applications in the untrusted infrastructure [8, 10, 55, 69, 75]. *Shielded execution* aims to provide strong security properties using a hardware-protected secure memory region or *enclave*.

While the shielded execution frameworks provide strong security guarantees against a powerful adversary, they are primarily designed for securing “stateless" (or volatile) in-memory computations and data. Unfortunately, these stateless techniques are not sufficient for building a secure storage system, where the data is persistently stored on an untrusted storage medium, such as an SSD or HDD. The challenge is *how to extend the trust beyond the “secure, but stateless/volatile" enclave memory region to the “untrusted and persistent" storage medium, while ensuring that the security properties are preserved in the “stateful settings", i.e., even across the system reboot, migration, or crash.*

To answer this question, we aim to build a secure storage system using shielded execution targeting all three important security properties for the data storage and query processing: (a) *confidentiality* — unauthorized entities cannot read the data, (b) *integrity* — unauthorized changes to the data can be detected, and (c) *freshness* — stale state of data can be detected as such.

To achieve these security properties, more specifically, we need to address the following three architectural limitations of shielded execution in the context of building a secure storage system: Firstly, the secure enclave memory region is quite limited in size, and incurs high performance overheads for memory accesses. It implies that the storage engine cannot store the data inside the enclave memory; thus, the in-memory data needs to be stored in the untrusted host memory. Furthermore, the storage engine persists the data on an untrusted storage medium, such as SSDs. Since the TEE cannot give any security guarantees beyond the enclave memory, we need to design mechanisms for extending the trust to secure the data in the untrusted host memory and also on the persistent storage medium.

Secondly, the syscall-based I/O operations are quite expensive in the context of shielded execution since the thread executing the system call has to exit the enclave, and perform a secure context switch, including TLB flushing, security checks, etc. While existing shielded execution frameworks [8, 55] proposed an *asynchronous* system call interface [70], it is clearly not well-suited for building a storage system that requires frequent I/O calls. To mitigate the expensive enclave exits caused by I/O syscalls, we need to design a direct I/O library for shielded execution to completely eliminate the expensive context switch from the data path.

Lastly, we also aim to ensure data freshness to protect against *rollback* (replay old state) or *forking attacks* (create second instance). Therefore, we need a protection mechanism based on a trusted monotonic counter [57], for example, SGX trusted counters [3]. Unfortunately, the SGX trusted counters are extremely slow and they wear out within a couple of days of operation. To overcome the limitations of the SGX counters, we need to redesign the trusted monotonic counters to suit the requirements of modern storage systems.

To overcome these design challenges, we propose SPEICHER, a secure LSM-based KV storage system. More specifically, we make the following contributions.

- **I/O library for shielded execution:** We have designed a direct I/O library for shielded execution based on Intel SPDK. The I/O library performs the I/O operations without exiting the secure enclave; thus it avoids expensive system calls on the data path.
- **Asynchronous trusted monotonic counter:** We have designed trusted counters to ensure data freshness. Our counters leverage the lag in the sync operations in modern KV stores to asynchronously update the counters. Thus, they overcome the limitations of the native SGX counters.
- **Secure LSM data structure:** We have designed a secure LSM data structure that resides outside of the enclave memory while ensuring the integrity, confidentiality and freshness of the data. Thus, our LSM data structure overcomes the memory and I/O limitations of Intel SGX.
- **Algorithms:** We present the design and implementation of all storage and query operations in persistent KV stores: get, put, range queries, iterators, compaction, and restore.

We have built a fully-functional prototype of SPEICHER based on RocksDB [65], and extensively evaluated it using the RocksDB benchmark suite. Our evaluation shows that SPEICHER incurs reasonable overheads, while providing strong security properties against powerful adversaries.

2 Background and Threat Model

2.1 Intel SGX and Shielded Execution

Intel Software Guard Extension (SGX) is a set of x86 ISA extensions for Trusted Execution Environment (TEE) [15]. SGX provides an abstraction of secure *enclave*—a hardware-protected memory region for which the CPU guarantees the confidentiality and integrity of the data and code residing in the enclave memory. The enclave memory is located in the Enclave Page Cache (EPC)—a dedicated memory region protected by an on-chip Memory Encryption Engine (MEE). The MEE encrypts and decrypts cache lines with writes and reads in the EPC, respectively. Intel SGX supports a call-gate mechanism to control entry and exit into the TEE.

Shielded execution based on Intel SGX aims to provide strong confidentiality and integrity guarantees for applications deployed on an untrusted computing infrastructure [8, 10, 55, 69, 75]. Our work builds on the SCONE [8] shielded execution framework. In SCONE, the applications are statically compiled and linked against a modified standard C library (SCONE libc). In this model, application's address space is confined to the enclave memory, and interaction with the untrusted memory is performed via the system call interface. In particular, SCONE runtime provides an *asynchronous system call* mechanism [70] in which threads outside the enclave asynchronously execute the system calls. SCONE protects the executing application against Iago attacks [13] through *shields*. Furthermore, it ensures memory safety for the applications running inside the SGX enclaves [36]. Lastly, SCONE provides an integration to Docker for seamlessly deploying containers.

2.2 Persistent Key-Value (KV) Stores

Our work focuses on persistent KV stores based on the LSM data structure [54], such as LevelDB [39] and RocksDB [65]. In particular, we base our design on RocksDB. RocksDB organizes the data using three constructs: MemTable, static sorted table (SSTable), and log files.

RocksDB inserts `put` requests to a memory-resident *MemTable* that is organized as a skip list [62]. For crash recovery, these `puts` are also sequentially logged to the write-ahead-log (WAL) file backed by persistent storage medium with checksums. When the MemTable fills up, it is moved to an *SSTable* file backed by an SSD or HDD in a batch to ensure sequential device access (this thus can cause scanning the skip list).

The SSTable files are grouped into levels with increasing size (typically 10×). The process of moving data to the next level is called *compaction*, which ensures the SSTables to be sorted by keys, including the ones being merged from the previous level. Since SSTables are immutable, compaction always creates new SSTables on the persistent storage medium. Any state changes in the entire storage system, such as creation and deletion of SSTable and WAL files, are recorded to the *Manifest*, which is a transactional and persistent log.

On a `get` request, RocksDB first searches the MemTable for the key, then searches the SSTables from the lowest level in turn; at each level, it binary-searches the corresponding SSTable. Using this primitive, it is trivial to process range and iterator queries, where the latter only differs in the client interface. RocksDB maintains an index table with a Bloom filter attached to each SSTable in order to avoid searching unnecessary SSTables.

While restarting, RocksDB establishes the latest state in a `restore` operation. To this end, the Manifest and the WAL are read and replayed.

2.3 Threat Model

In addition to the standard SGX threat model [10], we also consider the security attacks that can be launched using an *untrusted* storage medium, e.g., persistent state stored on an SSD or HDD. More specifically, we aim to protect against a powerful adversary in the virtualized cloud computing infrastructure [10]. In this setting, the adversary can control the entire system software stack, including the OS or hypervisor, and is able to launch physical attacks, such as performing memory probes.

For the untrusted storage component, we also aim to protect against rollback attacks [57], where the adversary can arbitrarily shut down the system, and replay from a stale state. We also aim to protect against forking attacks [40], where the adversary can attempt to fork the storage system, e.g., by running multiple replicas of the storage system.

Even under the extreme threat model, our goal is to guarantee the data integrity, confidentiality, and freshness. Lastly, we also aim to provide crash consistency for the storage system [58].

However, we do not protect against side-channel attacks, such as exploiting cache timing and speculative execution [78], or memory access patterns [25, 81]. Mitigating side channel attacks in the TEEs is an active area of research [53]. Further, we do not consider the denial of service attacks since these attacks are trivial for a third-party operator controlling the underlying infrastructure [10]. Lastly, we assume that the adversary cannot physically open the processor packaging to extract secrets or corrupt the CPU system state.

3 Design

SPEICHER is a secure persistent KV storage system designed to operate on an untrusted host. SPEICHER provides strong confidentiality, integrity, and freshness guarantees for the data storage and query operations: `get`, `put`, `range queries`, `iterators`, `compaction`, and `restore`. In this paper, we implemented SPEICHER by extending RocksDB [65], but our architecture can be generalized to other LSM-based KV stores.

3.1 Design Challenges

As a strawman design, we could try to secure a storage system by running the storage engine inside the enclave memory. However, the design of a practical and secure system requires addressing the following four important architectural limitations of Intel SGX.

I: Limited EPC size. The strawman design would be able to protect the in-memory state of the MemTable using the EPC memory. However, EPC is a limited and shared resource. Currently, the size of EPC is 128 MiB. Approximately 94 MiB are available to the user, the rest is reserved for the metadata. To allow creation of enclaves with sizes beyond that of EPC, SGX features a secure paging mechanism. The OS can evict EPC pages to an unprotected memory using SGX instructions. During eviction, the page is re-encrypted. Similarly, when an evicted page is brought back, it is decrypted and its integrity is checked. However, the EPC paging incurs high performance overheads (2×—2000×) [8].

Therefore, we need to redesign the shielded storage engine, where we allocate the MemTable(s) outside the enclave in the untrusted host memory. Since the secure enclave region cannot give any guarantees for the data stored in the host memory, and the native MemTable is not designed for security—we designed a new MemTable data structure to guarantee the confidentiality, integrity and freshness properties.

II: Untrusted storage medium. The storage engine does not exclusively store the data in the in-memory MemTable, but also on a persistent storage medium, such as on an SSD or HDD. In particular, the storage engine stores three types of files on a persistent storage medium: SSTable, WAL and the Manifest. However, Intel SGX is designed to protect only the volatile state residing in the enclave memory. Unfortunately, SGX does not provide any security guarantees for stateful computations, i.e., across system reboot or crash. Further, the trust from the TEE does not naturally extend to the untrusted persistent storage medium.

To achieve the end-to-end security properties, we further redesigned the LSM data structure, including the persistent storage state in the SSTable and log files, to extend the trust to the untrusted storage medium.

III: Expensive I/O syscall. To access data stored on an SSD or HDD (in the SSTable, WAL or Manifest files), conventional systems leverage the system call interface. However, the system call execution in the SGX environment incurs high performance overheads. This is because the thread executing the system call has to exit the enclave, and the syscall arguments need to be copied in and out of the enclave memory. These enclave transitions are expensive because of security checks and TLB flushes.

To mitigate the context switch overhead, shielded execution frameworks, such as SCONE [8] or Eleos [55], provide an *asynchronous* system call interface [70], where a thread outside the enclave asynchronously executes the system calls without forcing the enclave threads to exit the enclave. While such an asynchronous interface is useful for many applications, it is not clearly suited for building a storage system that needs to support frequent I/O system calls.

To support frequent I/O calls within the enclave, we designed a new I/O mechanism based on a direct I/O library for shielded execution leveraging storage performance development kit (SPDK) [28].

IV: Trusted counter. In addition to guaranteeing the integrity and confidentiality, we also aim to ensure the freshness of the stored data to protect against rollback attacks [57]. To achieve the freshness property, we need to protect the data stored in the untrusted host memory (MemTable), and those on the untrusted persistent storage medium (SSTable, WAL and Manifest files).

For the first part, i.e., to ensure the freshness of MemTable allocated in the untrusted host memory, we can leverage the EPC of SGX. In particular, the Memory Encryption Engine (MEE) in SGX already protects the EPC against rollback attack. Therefore, we use the EPC to store a *freshness signature* of the MemTable, which we use at runtime to verify the freshness of data stored as part of the MemTable in the untrusted host memory.

However, the second part is quite tedious, i.e., to ensure the freshness of the data stored on untrusted persistent storage (SSTables and log files), because the rollback protected EPC memory is stateless, or it cannot be used to verify the freshness properties after the system reboots or crashes. Therefore, we need a rollback protection mechanism based on a trusted monotonic counter [57]. For example, we could use SGX trusted counters [3]. Unfortunately, the SGX trusted counters are extremely slow (60 – 250 ms) [45]. Furthermore, the counter memory allows only a limited number of write operations to NVRAM, and it easily becomes unusable due to wear out within a couple of days of operation. Therefore, the SGX counters are impractical to design a storage system.

To overcome the limitations of SGX counters, we designed an asynchronous trusted monotonic counter that drastically improves the throughput and mitigates wear-out by taking advantage of the crash consistency properties of modern storage systems.

3.2 System Components

We next detail the system components of SPEICHER. Figure 1 illustrates the high-level architecture and building blocks of SPEICHER. The system is composed of the controller, a direct-I/O library for shielded execution, a trusted monotonic counter, the storage engine (RocksDB engine), and a secure LSM data structure (MemTable, SSTable, and log files).

SPEICHER controller. The controller provides the trusted execution environment based on Intel SGX [8]. Clients communicate over a mutually authenticated encrypted channel (TLS) to the controller. The TLS channel is terminated inside the controller. In particular, we built the controller based on the SCONE shielded execution framework [8], where we leverage SCONE's container support for secure deployment of the SPEICHER executable on an untrusted host.

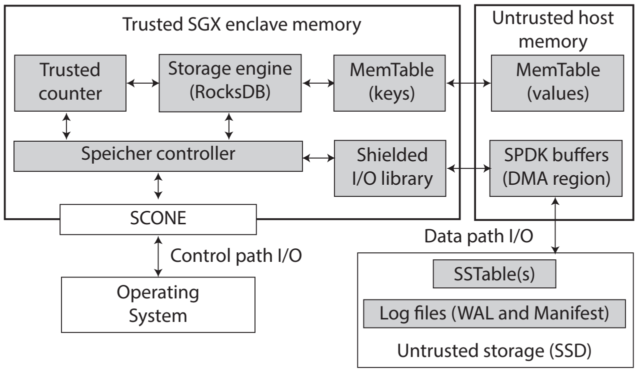

Figure 1: SPEICHER overview (shaded boxes depict the system components)

The controller provides the remote attestation service to the clients [6, 32]. In particular, the SGX enclave generates a signed measured of its identity, whose authenticity can be verified by a third party. After successful attestation, the client provides its encryption keys to the controller. The controller uses the client certificate to perform the access control operation. The controller also provides runtime support for user-level multithreading and memory management inside the enclave. The controller leverages the asynchronous system calls interface (SCONE libc) on the control path for the system configuration. For the data path I/O, we built a direct I/O library, which we describe next.

Shielded direct I/O library. The I/O library allows the storage engine to access the SSD or HDD from inside the SGX enclave, without issuing the expensive enclave exit operations. We achieve this by building a direct I/O library for shielded execution based on SPDK [28].

SPDK is a high-performance user-mode storage library, based on Data Plane Development Kit (DPDK) [2]. It eliminates the need to issue system calls to the kernel for read and write operations by having the NVMe driver in the user space. SPDK enables zero-copy I/O by mapping DMA buffers to the user address space. It relies on actively polling the device instead of interrupts.

These SPDK features align with the goal of SPEICHER of exit less I/O operations in the enclave, i.e., to allow the shielded storage engine to interact with the SSD directly. However, we need to adapt the design of SPDK to overcome the limitations of the enclave memory region. In particular, our shielded I/O library allocates huge pages and SPDK ring buffers outside the enclave for DMA. The host system maps the device in an allocated DMA region. Afterwards SPDK can initialize the device. To reduce the number of enclave exits, SPDK's device driver runs inside the enclave. This enables efficient delivery of requests from the storage engine to the driver, which explicitly copies the data between the host and the enclave memory.

Trusted counter. In order to protect the system from rollback attacks, we need a trusted counter whose value is stored alongside with the LSM data structure. Intel SGX provides monotonic counters, but their update frequency is in a range of 10 updates per second, and we indeed measured approximately 250 ms to increment a counter once. This is far too slow for modern KV stores [26].

To overcome the limitations of SGX counters, we designed an Asynchronous Monotonic Counter (AMC) based on the observation that many contemporary KV stores do not persist their inserted data immediately. This allows AMC to defer the counter increment until the data is persisted without loosing any availability guarantees. As a result, AMC achieves 70K updates per second in the current implementation.

AMC provides an asynchronous increment interface, because it takes a while since the counter value is incremented until it becomes *stable*, which means the counter value cannot be rolled back without being detected. At an increment, AMC returns three pieces of information: the current stable value, the incremented counter value, and the *expected time* for the value to be stable. Due to the expected time and the controller having to be re-authenticated after a shutdown, the client only has to keep the values until the stable time has elapsed, to prevent any data loss in case of a sudden shutdown.

AMC's flexible interface allows us to optimize update throughput and latency by increasing the time until a trusted counter is stable. This also allows users to adjust trade-off between the wear out of the SGX monotonic counter and the maximum number of unstable counter increments, which a client might have to account for. SPEICHER generates multiple counters by storing their state to a file, whose freshness is guaranteed through the use of a synchronous trusted monotonic counter. For instance, we can employ SGX monotonic counters [3], ROTE [45] or Ariadne [71] to support our asynchronous interface. Therefore, we can have a counter with deterministic increments for WAL and the Manifest, making it possible to argue about the freshness of each record in the files.

MemTable. As detailed in §3.1, the EPC is limited in size and the EPC paging incurs very high overheads. Therefore, it is not judicious to store large MemTables or multiple MemTables within the EPC. Further, since SPEICHER uses the EPC memory region to secure the storage engine (RocksDB) and the shielded I/O library driver, it further shrinks the available space.

Due to this memory restriction, we need to store the MemTable in the host memory. Since the host memory is untrusted, we need to devise a mechanism to ensure the confidentiality, integrity, and freshness of the MemTable.

In our project, we tried three different designs for the MemTable. Firstly, we explored a native Merkle tree that generates hashes of the leafs and stores them in each node. Thus, we can verify the data integrity by checking the root node hash and each hash down to the leaf storing the KV, while allowing the MemTable to be stored outside the EPC memory. However, the native Merkle tree suffers from slow lookups as the key has to be decrypted on each traversal. Further, it requires multiple hash recalculations on each lookup and insertion.

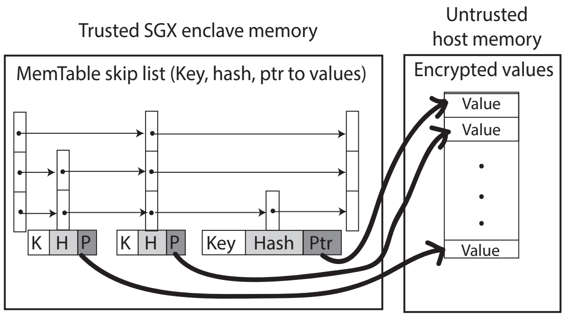
Trusted SGX enclave memory
Untrusted host memory
MemTable skip list (Key, hash, ptr to values)
Encrypted values
Value
Value
Value
K
H
P
K
H
P
Key
Hash
Ptr

Figure 2: SPEICHER MemTable format

Secondly, we tried a modified Merkle tree design based on a prefix array, where a fixed size prefix is used as an index into the array of Merkle trees. An array entry holds the root node of a Merkle tree, which holds the actual data. This should reduce the depth of the search tree compared to the native Merkle tree; thus, reducing the number of necessary hash calculations and decryptions of keys. However, while we were able to increase the lookup speed compared to the native Merkle tree, it still suffered from the same problem of having to decrypt a large number of keys in a lookup, and causing a large number of hash calculations.

Lastly, our third attempt of the MemTable design reuses the existing skip list data structure for the MemTable in RocksDB. Figure 2 shows SPEICHER's MemTable format. In particular, we partition the existing MemTable in two parts: key path and value path. In the key path, we store the keys as part of the skip list inside the enclave. Whereas, the encrypted values in the MemTable are stored in the untrusted host memory as part of the value path. This partitioning allows SPEICHER to provide confidentially by encrypting the value, while still enabling fast key lookups inside the enclave. To prevent attacks on the integrity or the freshness of the values, SPEICHER stores a cryptographic hash of the value in each skip list node together with the host memory location of the value.

While the first two designs removed almost the entire MemTable from the EPC, the last design still maintains the keys and hash values inside the enclave memory. To determine the space requirements of our MemTable in comparison to the regular RocksDB's MemTable, we use the following formula:

$$S = n*(k+v) + \sum_{i=0}^{m} p^{i} * n * ptr$$

Where S represents the entire size of the skip list, n is the number of KV pairs, k is the key size, v is the value size or the size of the pointer plus hash value for our skip list, p is the probability for being added into a specific layer of the skip list, m is the maximum number of layers, and ptr is the size of a pointer in the system.

For instance, in case of the default setting for RocksDB, with a maximum size of 64 MiB, key size of 16 B, value size

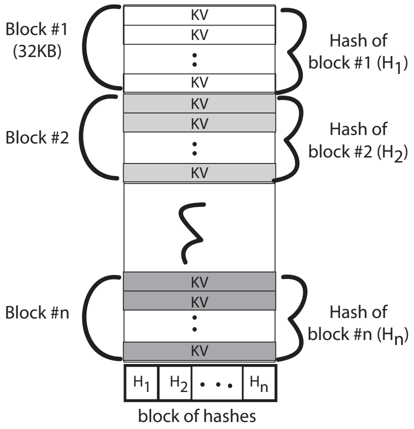

Figure 3: Speicher SSTable file format

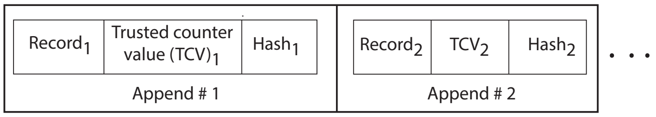

Figure 4: Speicher append-only log file format

of 1024 B, pointer size of 8 B, p of 1/4, m of 12 and for Speicher's skip list a hash size of 16 B — Speicher's MemTable achieves a space reduction of approximately 95.2 %. Further, the reduction ratio increases with increased value size.

SSTables. The SSTable files maintain the KV pairs persistently. These files store KV pairs in the ascending order of keys. This organization allows for a binary search within the SSTable, requiring only a few reads to find a KV-pair within the file. Since SSTable files are optimized for block devices, such as SSDs, they group KV pairs together into blocks (the default block size is 32 KiB).

Speicher adapts SSTable file format to ensure the security properties (see Figure 3 for Speicher's SSTable file format). The confidentiality is secured by encrypting each block of the SSTable file before it is written to the persistent storage medium. Additionally, Speicher calculates a cryptographic hash over each block. These hashes are then grouped together in a block of hashes and appended at the end of the SSTable file. When reading Speicher can check the integrity of each block by calculating the block's hash and comparing it to the corresponding hash stored in the footer. To protect the integrity of the footer an additional hash over the footer is calculated and stored in the Manifest. Since the Manifest is protected against rollback attacks using a trusted counter, the footer hash value stored in the Manifest is also protected from the rollback attacks. Thus, Speicher can use this hash to guarantee the freshness of the SSTable file's footer and transitively the freshness of each block in the SSTable file.

Log files. RocksDB uses two different log files to keep track of the state of the KV store: (a) WAL for persisting inserted KV pairs until a top-level compaction; and (b) the Manifest to keep track of live files, i.e., the set of files of which the current state of the KV store consists. Speicher adapted these log files to ensure the desired security properties, as shown in Figure 4.

Regarding WAL, every `put` operation appends a record to the current WAL. This record consists of the encrypted KV pair, and an encrypted trusted counter value for the WAL at the moment of insertion, and a cryptographic hash over both. Since the records are only appended to the WAL, Speicher can use the trusted counter value and the hash value to verify the KV pair, and to replay the operations in a restore event.

The Manifest is similar to the WAL; it is a write-append log consisting of records storing changes of live files. We use the same scheme for the Manifest file as we do for the WAL.

3.3 Algorithms

We next present the algorithms for all storage operations in Speicher. The associated pseudocodes are detailed in the appendix.

I: Put. Put is used to insert a new KV pair into the KV store, or to update an existing one. We need to perform two operations to insert the KV pair into the store (see Algorithm 1). First, we need to append the KV pair to the WAL for persistence. Second, we need to write the KV pair to the MemTable for fast lookups.

Inserting the KV pair into the WAL guarantees that the state of the KV store can be restored after an unexpected reboot. Therefore, the KV pair should be inserted into the WAL before it is inserted into the MemTable. To add a KV pair to the WAL, Speicher encrypts the pair together with the next WAL trusted counter value and a cryptographic hash over both the data and the counter. The encrypted block is then appended to the WAL (see the log file format in Figure 4). Thereafter, the trusted counter is incremented to the value stored in the appended block. In addition, the client is notified when the KV pair will be stable; thereafter, the state cannot be rolled back. In case of a system crash between generating the data block and increasing the trusted counter value, the data block would be invalid at reboot, because the trusted counter would point the block to a future time. This operation is safe as the client can detect a reboot when Speicher tries to authenticate itself. After the reboot the client can ask the KV store about what the last added key was, or can simply `put` the KV pair again in the store as another request with the same key supersedes any old value with the same key.

In the second step, Speicher writes the KV pair into the MemTable and thereby making the `put` visible to later `gets`. Speicher first encrypts the value of the KV pair and generates a hash over the encrypted data. The encrypted value is then copied to the untrusted host memory, while the hash with a pointer to the value is inserted into the skip list in the enclave, in accordance to Speicher's MemTable format (Figure 2). Since the KV pair is first inserted into the WAL, and only if this is successful, i.e., the WAL and trusted counter are updated, we can guarantee that only KV value pairs whose freshness is secured by the trusted counter are returned.

II: Get. Get may involve searching multiple levels in the LSM data structure to find the latest value. Within each level, Speicher has to generate either the proof of existence, or the proof of non-existence of the key. This is necessary to detect insertion or deletion of the KV pairs by an attacker.

Algorithm 2 details the `get` operation in SPEICHER. In particular, SPEICHER begins with searching the MemTable. SPEICHER searches the skip list for the node with the key. Either the key is in the MemTable, then the hash value is calculated over the value and compared to the hash stored in the skip list, or the key could not be found in the skip list. Since the skip list resides inside the protected memory region, SPEICHER does not need to make the non-existence proof for the MemTable because an attacker cannot access the skip list. If the KV store finds a key in the MemTable and the existence proof is correct, i.e., the calculated hash value is equal to the stored hash value, the value is returned to the client. If the proof is incorrect, the client is informed that the MemTable is corrupted. Since the MemTable can be reconstructed from the WAL, the client can then instruct the SPEICHER to recreate the KV store state in the case of an incorrect proof.

When the key is not found in the MemTable, the next level is searched. All levels below the MemTable are stored in SSTables. The SSTable files are organized in a way that no two SSTables in the same level have an overlapping key-range. Additionally, all the keys are sorted within an SSTable file. Due to this, any given key can only exist in one position in one SSTable file per level. This allows SPEICHER to construct a Merkle tree on top of the SSTable files of a level. With the ordering inside the SSTable, SPEICHER can correlate a block in the file with the key. This allows SPEICHER to calculate a hash over this block, which then can be checked against the stored hash in the footer. The hash of the footer can then be checked against the Merkle tree over the SSTable files in that level. It gives SPEICHER the proof of non-/existence for the lookup, and possibly the value belonging to the key. If the proof fails, the client is informed. In contrast to an incorrect proof in the MemTable, SPEICHER is not able to recover from this problem since the data is stored on the untrusted storage medium. If SPEICHER finds the KV pair and the proof is correct, it returns the value to the client. If the key does not exist, that is SPEICHER could not find it in any level and all level proofs are correct, an empty value is returned.

The freshness of data is guaranteed either by checking the value against the securely stored hash in the EPC for the case where the key has been found in the MemTable, or by checking the hash values of the SSTables against a Merkle tree. Additionally, as any key can only be stored in one position within a level, SPEICHER can also check against deletion of the key in a higher level, which is also necessary to guarantee freshness.

III: Range queries. Range queries are used to access all KV pairs, with a key greater than or equal to a start key and lesser than an end key (see Algorithm 3). To find the start KV pair, we need to do the same operation as in get requests. Furthermore, it requires to initialize an iterator in each level, pointing to the KV pair with a key greater or equal to the starting key. These iterators are necessary as higher levels have the more recent updates, due to keys being inserted into the highest level and being compacted over time to the lower levels, and lower being larger in size and therefore having more KV pairs. If the next KV pair is requested the next key of all iterators is checked and the iterators with the smallest next key are forwarded.

In case the next key is in multiple levels, the highest level KV pair is chosen. Therefore, SPEICHER has to do a non-/existence proof at all the levels, before it returns the chosen KV pair. If any of these proofs fails, the client is informed about the failed proof. Identical to the `get` operation, the client can then decide to either restore the KV store or to restore a backup.

Similar to the `get` operation, the hash value stored in the EPC and the Merkle tree over the SSTables are used to guarantee the freshness of the returned values.

IV: Iterators. Iterators work identical to the range queries; they just have a different interface (see Algorithm 4).

V: Restore. After a reboot, the KV store has to restore its last state (see Algorithm 5). This process is performed in two steps, first collecting all files belonging to the KV store, and then replaying all changes to the MemTable. In the first step the Manifest file is read. It contains all necessary information about the other files, such as live SSTable files, live WAL files, smallest key of each SSTable file. Each changing event about the live file is logged into the Manifest by appending a record describing the event. Therefore, at a restore all changes committed in the Manifest have to be replayed. This means that the SSTable files have to be put in the correct level. Each record in the Manifest is integrity-checked by a hash, and the freshness is guaranteed by the trusted counter for the Manifest. Since the counter value is incremented in a deterministic way, SPEICHER can use this value to check if all blocks are present in the Manifest. After the SSTable files in the levels are restored, and the freshness of all the SSTable files is checked against the Manifest by comparing the hash with the hash stored in the Manifest, the WAL is replayed.

Since each `put` operation is persisted in the WAL before it is written into the MemTable, replaying the `put` operations from the WAL allows SPEICHER to reconstruct the MemTable at the moment of the shutdown. Each `put` in the WAL has to be checked against the stored hash in the record, and the stored counter value. Additionally, since the counter value of the WAL is checked whether it equals to that of the Manifest counter, SPEICHER can check for the missing records. Records that have a counter value being in the future, i.e. a counter value higher than the stored stable trusted counter value are ignored at restore. Further, due to the deterministic increase of the counter, SPEICHER can check against the missing records in the log files. If in any of these steps one of the checks fails, SPEICHER returns the information to the client, because SPEICHER is not able to recover from such a state.

VI: Compaction. Compaction is triggered when a level holds data beyond a pre-defined threshold in size. In compaction (see Algorithm 6), a file from Level$_n$ is merged with all SSTable files in Level$_{n+1}$ covering the same key range. The new SSTables are added to Level$_{n+1}$, while all SSTables in the previous level are discarded. Before keys are added to the new SSTable file, the non-/existence proof is done on the files being merged. This is necessary to prevent the compaction process from skipping keys or writing old KV to the new SSTable files.

Since hash values are calculated over blocks of the SSTable files, a new block has to be constructed in the enclave memory, before it is written to the SSD. Also, all hash values of the blocks have to be stored in the protected memory until the footer is written and a hash over the footer is created. The file names of newly created SSTables and footer hashes are then written to the Manifest file, with the new trusted counter value. This is similar to the `put` operation. After the write operation to the Manifest completes and the trusted counter is incremented, the old SSTable files are removed from the KV store and the new files are added to Level$_{n+1}$. Since the hash values of the new SSTables are secured with a trusted counter value in the Manifest file, the SSTables cannot be rolled back after the compaction process.

3.4 Optimizations

Timer performance. As described in §3.2, in order to prevent every request from blocking for the trusted counter increment, we leverage asynchronous counters written in files whose freshness is guaranteed by synchronous counters (or SGX counters). We use one counter for the WAL and another for the Manifest so that SPEICHER can operate on them independently. Although this method drastically improves throughput by allowing SPEICHER to process many requests without waiting for the counter to be stable, it also poses on the client the need for holding its write requests until the counter value is stable. This is why we designed and implemented the interface of AMC that reports the expected time for the counter to be stable. Because of this interface, the client does not need to frequently issue the requests to check the current stable counter value.

SPDK performance. SPDK is designed to eliminate system calls from the data path, but in reality its data path issues two system calls on every I/O request: one for obtaining the process identifier and the other for obtaining the time. They are executed once in an I/O request that covers multiple blocks and their costs are normally amortized. However, since the context switch to and from the enclave is an order of magnitude more expensive, these costs are not amortized enough. We modified them to obtain the values from a cache within the enclave that are updated only at the vantage points. As a result, we achieved 25× improvements over the naive port of SPDK to the enclave.

4 Implementation

Direct I/O library. Our direct I/O library for shielded execution extends Intel SPDK. Further, the memory management routines and the `uio` kernel module that maps the device memory to the user space are based on Intel DPDK [2]. Although the device DMA target is configured outside the enclave, the SSD device driver and library code, including BlobFS in which SPEICHER stores RocksDB files, entirely run within the enclave.

We use SPDK 18.01.1-pre and DPDK 18.02. In SPDK, 56 LoC are added, and 22 LoC are removed. In DPDK, 138 LoC are added and 72 LoC are removed. These changes were made to replace the routines that cannot be executed in the enclave.

Trusted counters. AMCs are implemented using the Intel SGX SDK. A dedicated thread continually checks if any monotonic counter value has changed. If a counter value has been incremented, the thread writes the current value to the file. The storage engine can query the *stable value* of any of its counters, i.e., the last value that has been written to disk. Note that this value cannot be rolled back since it is protected by the synchronous SGX monotonic counter. Overall, our trusted counter consists of 922 LoC.

SPEICHER controller. The SPEICHER controller is based on SCONE. We leverage the Docker integration in SCONE to seamlessly deploy SPEICHER binary on an untrusted host. Further, we implemented a custom memory allocator for the storage engine. The memory allocator manages the unprotected host memory, and exploits RocksDB's memory allocation pattern, which allows us to build a lightweight allocator with just 119 LoC. Further, the controller employs our direct I/O library on the data path, and the asynchronous `syscall` interface of SCONE on the control path for system configuration. The controller also implements a TLS-based remote attestation for the clients [32]. Lastly, we integrated the trusted counter as a part of the controller, and exported the APIs to the storage engine.

Storage engine. We implemented the storage engine by extending a version of RocksDB that leverages SPDK. In particular, we extended the RocksDB engine to run within the enclave, also integrated our direct I/O library. Since the RocksDB engine with SPDK does not support data encryption and decryption, we also ported encryption support from the regular RocksDB engine using the Botan Library [1] (1000 LoC). In addition to encrypting data files, we extended the encryption support to ensure the confidentiality of the WAL and Manifest files. We further modified the storage engine to replace the LSM data structure and log files with our secure MemTable, SSTables, and log files. Altogether, the changes in RocksDB account for 5029 new LoC and 319 changed LoC.

MemTables. RocksDB as default uses a skip list for MemTable. However, it does not offer any authentication or freshness guarantees. Therefore, we replaced MemTable with an authenticated data structure coupled with mechanisms to ensure the freshness property. Our MemTable uses the `Inlineskiplist` of RocksDB and replaces the value part of the KV-pair with a node storing a pointer to and the size of the value as well as an HMAC. For the en-/decryption as well as for the HMAC we used OpenSSLs AES128 in GCM mode. This results in a 16 B wide HMAC. This implementation consists of 459 LoC. As discussed previously, we also implemented MemTable with a native Merkle tree (1186 LoC) and a Merkle tree with a prefix array (528 LoC). However, we did not use them eventually since their performance was quite low.

SSTables. To preserve the integrity of the SSTable blocks, we changed the block layer in RocksDB to calculate the hash before it issues a write request to the underlying layer. The hash is then cached until the file is flushed (258 LoC). Thereafter, hashes of all blocks are appended to the file coupled with the information about the total number of blocks, and the hash of this footer. When a file is opened, our hash layer loads the footer into the protected memory and calculates the hash of the footer. It then compares the value against the hash stored in the Manifest file. Only if these checks are passed, it opens the corresponding SSTable file and normal operations proceed. At reading, the hash of the block is calculated and checked against the hashes stored in the protected memory area, before the block data is handed to the block layer of RocksDB. We further enabled AES128 encryption to ensure the confidentiality of the blocks (188 LoC). The hashes used in the SSTables are SHA-3 with 384 bit.

Log files. Log files including the WAL and the Manifest use the same encryption layer as the SSTable files. However, the validation layer is different, and comes before the block layer since the operation requires knowledge of the record size. While writing, the validation layer adds the hash and the trusted counter value to the log files.

The validation layer uses the knowledge that log files are only read sequentially at startup for restoring purpose. Therefore, at the start up, the layer allows any action written in the log file as long as the hash is correct, and the stored counter increases as expected. At the end of the file, SPEICHER checks if the stored counter is equal to the trusted counter. The last record's freshness is guaranteed through the trusted counter. Integrity of all the records is guaranteed through the hash value protecting also the stored counter value. This value can then be checked against the expected counter value for that block. Since the counter lives longer than the log files, the start record value has to be secured too. In case of WAL, this is achieved by storing the start counter value of the WAL in the Manifest. The start record of the Manifest is implicitly secured, since the record must describe the state of the entire KV store.

5 Evaluation

Our evaluation answers the following questions.

- What is the performance (IOPS and throughput) of the direct I/O library for shielded execution? (§5.2)
- What is the impact of the EPC paging on the MemTable? (§5.3)
- What are the performance overheads of SPEICHER in terms of throughput and latency measurements? (§5.4)
- What is the performance of our asynchronous trusted counter? And what stability guarantees it has to provide to be compatible with modern KV stores? (§5.5)
- What is the I/O amplification overhead? (§5.6)

5.1 Experimental Setup

Testbed. We used a machine with Intel Xeon E3-1270 v5 (3.60 GHz, 4 cores, 8 hyper-threads) with 64 GiB RAM running Linux kernel 4.9. Each core has private 32 KiB L1 and 256 KiB L2 caches, and all cores share a 8 MiB L3 cache. For the storage device our testbed uses a Intel DC P3700 SSD. The SSD has a capacity of 400 GB and is connected over PCIe x4.

Workload	Pattern	Read/Write ratio
A *(default)*	Read-write	90R—10W
B	Read-write	80R—20W
C	Read only	100R—0W

Table 1: RocksDB benchmark workloads.

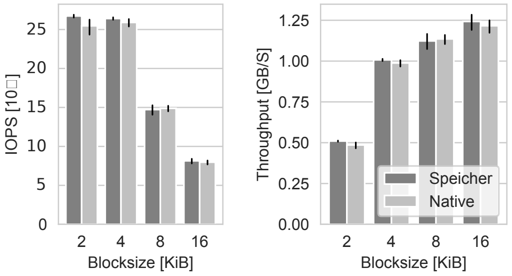

Figure 5: Performance of direct I/O library for shielded execution vs native SPDK.

Methodology for measurements. We compare the performance of SPEICHER with an unmodified version of RocksDB. The native version of RocksDB does not provide any security guarantees, i.e., it provides no support for confidentiality, integrity and freshness of the data and query operations.

Importantly, we stress-test the system by running a client on the same machine as the KV store. This is the worst-case scenario for SPEICHER since the client is not communicating over the network. Usually, the network slows down client's requests, and therefore, such an experimental setup is unable to stress-test the KV store. We avoid this scenario by running the client as part of the same process on the same host. This eliminates further the need for enclave enters and exits, which would add a high overhead, making a stress-test impossible.

Compiler and software versions. We used the RocksDB version with SPDK support (git commit `3c30815`). We used SPDK version 18.01.1-pre (git commit `73fee9c`), which we compiled with DPDK version 18.02 (commit `92924b2`). The native version of SPDK/ DPDK and RocksDB was compiled with `gcc 6.3.0` and the default release flags. The SPEICHER version of SPDK/DPDK and RocksDB was compiled with the same release flags but `gcc version 7.3.0` of the SCONE project.

RocksDB benchmark suite. We use the RocksDB benchmark suite for the evaluation. In particular, we used the `db_bench` benchmarking tool which is shipped with RocksDB [5] and Fex [52]. The benchmark consists of three workloads as shown in Table 1. Workload A is the default workload.

5.2 Performance of the Direct I/O Library

We first evaluate the performance of SPEICHER's I/O library for shielded execution. The I/O library is designed to have fast access to the persistent storage for accessing the KV pair stored

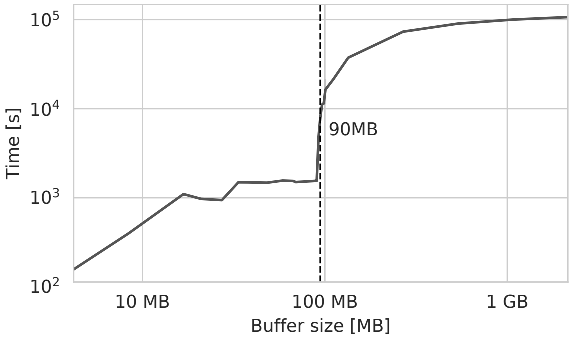

Figure 6: Impact of the EPC paging on the MemTable.

on the SSD (§3.2). We run the performance measurement 20 times for every configuration of block size for the native execution and SPEICHER. Figure 5 shows the mean throughput and IOPS with our I/O library and those with the native RocksDB-SPDK with a confidence interval of 95%. We use Workload B (80%R—20%W). Since the communication between SPDK and the device is handled completely over DMA, our direct I/O library does not suffer from context switches. Additionally, due to storing the buffers outside of the enclave, we also do not require expensive EPC paging, which would drastically reduce the performance of the I/O library. Our performance evaluation of the direct I/O library shows that it does not suffer from any performance deprecation compared to the native SPDK setup.

5.3 Impact of the EPC paging on MemTable

We next study the impact of EPC paging on MemTable(s). Note that a naive solution of storing a large or many MemTables in the EPC memory would incur high performance overheads due to the EPC paging. Therefore, we adopted a split MemTable approach, where we store only the keys along with metadata (hashes and pointers to value) inside the EPC, but the values are stored in the untrusted host memory (§3.3). To confirm the overheads of the EPC paging on accessing a large MemTable which are incurred in our rejected design, we measure the overheads of accessing random nodes in a MemTable completely resident in the enclave memory.

Figure 6 shows the performance overhead of accessing memory within the SGX enclave. The result shows that as soon as SGX has to page out MemTable memory from the EPC, which happens at 96 MiB, the performance drops dramatically. This is due to the en-/decryption and integrity checks employed by the MEE in Intel SGX. Therefore, it is important for our system design to keep the data values in the untrusted host memory to avoid the expensive EPC paging. Our approach of only keeping the key path of the MemTable inside the EPC requires a small EPC memory footprint. Therefore, our MemTable does not incur the EPC paging overhead.

5.4 Throughput and Latency Measurements

We next present the end-to-end performance of SPEICHER with different workloads, value sizes and thread counts. We measured the average throughput and latency for each of our benchmarks. Figure 7 shows the measurement results as a ratio of slowdown to the native SPDK-based RocksDB.

Effect of varying workloads. In the first experiment, we used different workloads listed in Table 1. The workloads were evaluated with 5 million KV pairs each. Each key was 16 B and value was 1024 B. The benchmarks were run single threaded.

We get a throughput of 34.2k request/second (rps) for Workload A down to 20.8k rps for Workload C, while RocksDB archived 512.8k rps or 676.8k rps respectively. The results show that SPEICHER overheads 15×—32.5× for different workloads. The overheads in Workloads A and B are mainly due to the operations performed in the MemTable, since SPEICHER has to encrypt the value and generate a cryptographic hash for every write to the MemTable. Furthermore, for each read operation the data has to be decrypted and the hash has to be recalculated and compared to one in the Skip list. However, even with AES-NI instructions, this decryption operation takes at least 1.3 cycles/byte for encryption, limiting the maximal reachable performance. The overhead in Workload C is due to reading a very high percentile of the KV pairs from the SSTable files, which uses currently an un-optimized code path for en-/decryption and hash calculations. We expect performance improvement by further optimizing the code path.

Effect of varying byte sizes. In the second experiment, we investigate the overheads with varying value sizes, since it changes the amount of data SPEICHER has to en-/decrypt and hash for each request. We used the default Workload A, and changed the value size from 64 B up to 4 KiB.

SPEICHER incurs an overhead of 6.7× for small value size, i.e. 64 B, up to an overhead of 16.9× for values of size 4 KiB. As in the previous experiment, the overhead is mainly dominated by the en-/decryption and hash calculation for the values in the MemTable. The benchmark shows a higher overhead for larger value sizes, since the amount of data SPEICHER has to en-/decrypt increases with the size of the values.

Effect of varying threads. We also investigated the scaling capabilities of SPEICHER. For that we increased the number of threads up to 8 and compared the overhead to native RocksDB with the default Workload A. Note that the current SGX server machine has 4 physical cores / 8 hyperthread cores.

In the test the overhead increased from around 13.6× for two threads to 17.5× for 8 threads. This implies SPEICHER scales slightly worse than RocksDB. This is due to less optimal caching for random memory access in SPEICHER's memory allocator. SPEICHER has to manage two different memory regions (host and EPC) for the MemTable, which leads to sub-optimal caching. We plan to optimize our memory allocator and data structures to exploit the cache locality.

Latency measurements. In the benchmarks, SPEICHER has an average latency ranging from 16 µs for single threaded and 64 B value size up to 256 µs for 8 threads and 1024 B value size, native RocksDB had for the same benchmark a latency of 1.6 µs or 14 µs respectively. However, RocksDB's best latencies were in Workload C with an average of 1.5 µs.

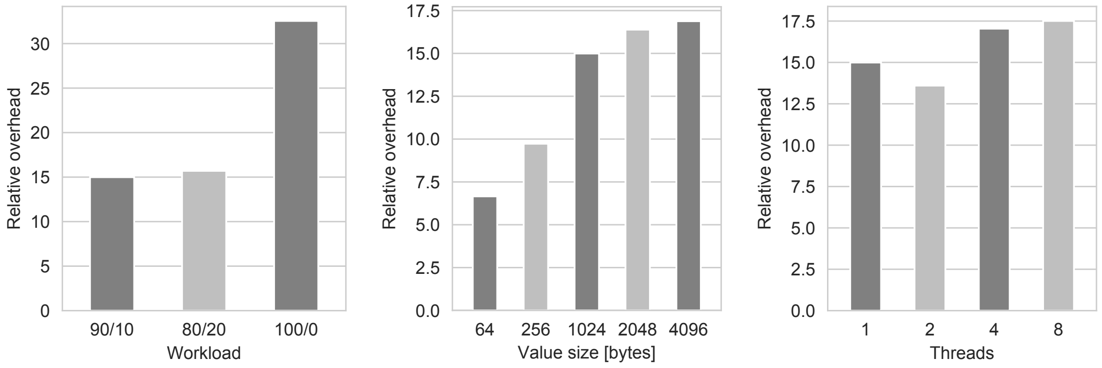

Figure 7: SPEICHER performance normalized to the native RocksDB (with no security): (a) different workloads with constant value size of 1024 and constant number of threads, (b) varying value sizes, and (c) increasing number of threads.

KV store	Default time for persistence (ms)	Configurable
RocksDB	0 (flushing)	yes
LevelDB	0 (non-flushing)	yes
Cassandra	1000	yes
HBase	10000	yes

Table 2: Default time for data persistence in KV stores.

5.5 Performance of the Trusted Counter

The synchronous trusted counter rate of SGX is limited to one increment at every 60 ms. This would limit our approach to only 20 `Put` operations per second since each `Put` has to be appended to the WAL, which requires a counter increment. However, our latency suggest that we have a lot more put operations to deal with. Even in our worse latency case with 256 µs per request we would expect 234.4 request per 60 ms, with a write rate of 10% this would amount to 23.4 required counter increases every possible sequential counter increase. In practice SPEICHER should reach far higher update rates as this calculation used worst case values from our benchmarks.

Table 2 shows the time before different KV stores guarantee that the values are persisted. We argue that these times can be used to hide the stability time of our asynchronous counters, which is a maximum of 60 ms. This is far less than the maximum time to persist the data in the default configuration of Cassandra and HBase. If the client expects the value is persisted only after a specific period of time, we can relax our freshness guarantees to match to the same time window.

5.6 I/O Amplification

We measured the relative I/O amplification increase in data for SPEICHER compared to the native RocksDB. We report the I/O amplification results using the default workload (A) with the key size of 16 B and value size of 4 KiB. We observed an overhead of 30% for read and write in the I/O amplification. This overhead mainly comes from the footer we have to add to each SSTable as well as from the hashes and counter values we have to add to the log files. This overhead is not only present in the write case but also in the read, as the additional data has also to be read to be able to verify the files.

6 Related Work

Shielded execution. Shielded execution frameworks provide strong security guarantees for legacy applications running on an untrusted infrastructure. Prominent examples include Haven [10], SCONE [8], Graphene-SGX [75], Panoply [69], and Eleos [55]. Recently, there has been a significant interest in designing secure systems based on shielded execution, such as VC3 [68], Opaque [82], Ryoan [27], Ohrimenko et al. [51], SGXBounds [36], etc. However, these systems are primarily designed to secure stateless computation and data. (Pesos [34] is an exception, see the policy-based storage systems section for the details.) In contrast, we present the first secure persistent LSM-based KV storage system based on shielded execution.

I/O for shielded execution. To mitigate the I/O overheads in SGX, shielded execution frameworks, such as Eleos [55] and SCONE [8], proposed the usage of an asynchronous system call interface [70]. While the asynchronous interface is sufficient for the low I/O rate applications—it can not sustain the performance requirements of modern storage/networked systems. To mitigate the I/O bottleneck, ShieldBox [73] proposed a direct I/O library based on Intel DPDK [2] for building a secure middlebox framework. Our direct I/O library is motivated by this advancement in the networking domain. However, we propose the first direct I/O library for shielded execution based on Intel SPDK [28] for the I/O acceleration in storage systems.

Trusted counters. A trusted monotonic counter is one of the important ingredients to protect against rollback and equivocation attacks. In this respect, Memoir [57] and TrInc [40] proposed the usage of TPM-based [74] trusted counters. However, TPM-based solutions are quite impractical because of the architectural limitations of TPMs. For instance, they are rate-limited (only one increment every 5 seconds) to prevent wear out. Therefore, they are mainly used for secure data access in the offline settings, e.g., Pasture [33].

Intel SGX has recently added support for monotonic counters [3]. However, SGX counters are also quite slow, and they wear out quickly (§3). To overcome the limitations, ROTE [45] proposed a distributed trusted counter service based on a consensus protocol. Likewise, Ariadne [71] proposed an optimized technique to increment the counter by a single bit flip. Our asynchronous trusted counter interface is complimentary to these synchronous counter implementations. In particular, we take advantage of the properties of modern storage systems, where we can use these synchronous counters to support our asynchronous interface.

Policy-based storage systems. Policy-based storage systems allow clients to express fine-grained security policies for data management. In this context, a wide range of storage systems have been proposed to express client capabilities [22], enforce confidentiality and integrity [21], or enable new features that include data sharing [44], database interface [46], policy-based storage [19, 77], or policy-based data seal/unseal operations [67]. Amongst all, Pesos [34] is the most relevant system since it targets a similar threat model. In particular, Pesos proposes a policy-based secure storage system based on Intel SGX and Kinetic disks [31]. However, Pesos relies on trusted Kinetic disks to achieve its security properties, whereas SPEICHER targets an untrusted storage, such as an untrusted SSD. Secondly, Pesos is designed for slow trusted HDDs, where the additional overheads of the SGX-related operations are eclipsed by slow disk operations. In contrast, SPEICHER is designed for high-performance SSDs.

Secure databases/datastores. Encrypted databases, such as CryptDB [60], Seabed [56], Monomi [76], and DJoin [50], are designed to ensure the confidentiality of computation in untrusted environments. However, they are primarily for preserving confidentiality. In contrast, SPEICHER preserves all three security properties: confidentiality, integrity, and freshness.

EnclaveDB [61] and CloudProof [59] target a threat model and security properties similar to SPEICHER. In particular, EnclaveDB [61] is a shielded in-memory SQL database. However, it uses the secondary storage only for checkpoint and logging unlike SPEICHER. Hence, it does not solve the problem of freshness guarantee for the data stored in the secondary storage. Furthermore, the system implementation does not consider the architectural limitations of SGX. Secondly, CloudProof [59] is a key-value store designed for untrusted cloud environment. Unlike SPEICHER, it requires the clients to encrypt or decrypt data to ensure confidentiality, as well as to perform attestation procedures with the server, introducing a significant deployment barrier.

TDB [43] proposed a secure database on untrusted storage. It provides confidentiality, integrity, and freshness using a log-structured data store. However, TBD is based on a hypothetical TCB, and it does not address many practical problems addressed in our system design.

Obladi [17] is a KV store supporting transactions while hiding the access patterns. While it can effectively hide the values and their access pattern against the cloud provider, it needs a trusted proxy. In contrast, SPEICHER does not rely on a trusted proxy. Furthermore, Obladi does not consider rollback attacks.

Lastly, in parallel with our work, ShieldStore [30] uses a Merkle tree to build a secure in-memory KV store using Intel SGX. Since ShieldStore is an in-memory KV Store, it does not persist the data using the LSM data structure unlike SPEICHER.

Authenticated data structures. Authenticated data structures (ADS) [47] enable efficient verification of the integrity of operations carried out by an untrusted entity. The most relevant ADS for our work is mLSM [63], a recent proposal to provide integrity guarantee for LSM. In contrast to mLSM, our system provides stronger security properties, i.e., we ensure not only integrity, but also confidentiality and freshness. Furthermore, our system targets a stronger threat model, where we have to design a secure storage system leveraging Intel SGX.

Robust storage systems. Robust storage systems provide strong safety and liveness guarantees in the untrusted cloud environment [14, 42, 79]. In particular, Depot [42] protects data from faulty infrastructure in terms of durability, consistency, availability, and integrity. Likewise, Salus [79] proposed a block store robust storage system while ensuring data integrity in the presence of commission failures. A2M [14] is also a robust system against Byzantine faults, and provides consistent, attested memory abstraction to thwart equivocation. In contrast to SPEICHER, this line of work neither provides confidentiality nor freshness guarantees.

Secure file systems. There is a large body of work on software-based secure storage systems. SUNDR [41], Plutus [29], jVPFS [80], SiRiUS [23], SNAD [48], Maat [38] and PCFS [21] employ cryptography to provide secure storage in untrusted environments. None of them protect the system from rollback attacks, and our challenges to overcome overheads of shielded execution are irrelevant for them. Among all, StrongBox [18] provides file system encryption with rollback protection; however, it does not consider untrusted hosts.

7 Conclusion

In this paper, we presented SPEICHER, a secure persistent LSM-based KV storage system for untrusted hosts. SPEICHER targets all the three important security properties: strong confidentiality and integrity guarantees, and also protection against rollback attacks to ensure data freshness. We base the design of SPEICHER on hardware-assisted shielded execution leveraging Intel SGX. However, the design of SPEICHER extends the trust in shielded execution beyond the secure enclave memory region to ensure that the security properties are also preserved in the stateful setting of an untrusted storage medium.

To achieve these security properties while overcoming the architectural limitations of Intel SGX, we have designed a direct I/O library for shielded execution, a trusted monotonic counter, a secure LSM data structure, and associated algorithms for storage operations. We implemented a fully-functional prototype of SPEICHER based on RocksDB, and evaluated the system using the RocksDB benchmark. Our experimental evaluation shows that SPEICHER achieves reasonable performance overheads while providing strong security guarantees.

Acknowledgement. We thank our shepherd Umesh Maheshwari for the helpful comments.

8 Appendix

In this appendix, we present the pseudocode for all data storage and query operations in SPEICHER.

Algorithm 1: Put algorithm of SPEICHER

```
Input: KV-pair which should be inserted into the store.
Result: Freshness of MemTable
/* Generating a block with the trusted counter */
$hash_{Block} \leftarrow$ hash($KV$,$counter_{WAL}+1$);
$block \leftarrow$ encrypt($KV$,$counter_{WAL}+1$,$hash_{Block}$);
/* Writing the block to the persistent storage, before the trusted counter gets incremented */
write$_{WAL}$($block$);
$counter_{WAL} \leftarrow counter_{WAL}+1$;
/* Generating hash over the KV-pair for the Memtable */
$hash_{KV} \leftarrow$ hash($KV$);
/* Trying to insert into the memtable, if the memtable is corrupted return a failure */
$freshness \leftarrow$ putIntoMemtable($KV$,$hash_{KV}$);
return $freshness$
```

Algorithm 2: Get algorithm of SPEICHER

```
Input: Key in the format of the KV-store
Result: Freshness of the KV-pair and Value
for $level = 0$ to $numberoflevels$ do /* Check in each level if key-value is existend, from highest to lowest */
    if $level = Level_0$ then /* First level lookup therefore lookup in MemTable */
        $path$,$value \leftarrow$ lookupMemtable($key$) /* It is possible that the value is empty, however we still have to do a proof of non-existence */
        foreach $node \in path$ do /* Validate hash values of the trace to the leaf node */
            if hash($node.left$,$node.right$) $\neq node.hash$ then /* check that the hash value of the child nodes is equal to the stored hash value */
                /* The integrity and freshness proof failed */
                return $stale_{MemTable}$,$value$
            end
        end
        return $fresh$,$value$
    end
    else /* Lookup in a level backup by SST files */
        $SST \leftarrow$ findSSTFile($level$,$key$) /* Lookup over authentication structures similar to MemTable */
        $block$,$value \leftarrow$ lookup($SSTs_{level}$,$key$);
        if hash($block$) $\neq$ $SST$.hashBlock($block$) or !freshness($SST$) then
            return $stale_{SST}$,$value$
        end
        return $fresh$,$value$
    end
end
```

Algorithm 3: Range query algorithm of SPEICHER

```
Input: KV-pair with the lowest key and callback method to the client
/* Build an iterator pointing to the first KV-pair */
$iterator \leftarrow$ constructIterator($key_{min}$);
$next \leftarrow True$;
/* Call the provided function until the iterator is not valid anymore or a freshness proof failed or the client request to end */
while isValid($iterator$) and $state = fresh$ and $next$ do
    $state$,$value \leftarrow Iterator.key\_value$;
    $next \leftarrow$ callback($state$,$value$);
    $Iterator \leftarrow Iterator.next$;
end
```

Algorithm 4: Iterator functions of SPEICHER

```
Input: Start key
Result: Result of freshness proof or iterator
Function constructIterator($key_{min}$)
    /* Build an iterator for each level of the LSM pointing to the KV-pair or the next pair in the level */
    foreach $level \in Level$ do
        $iterator_{level} \leftarrow$ lowerBound($level$,$key$);
        if $iterator_{level}.state \neq fresh$ then
            return $state$
        end
        iterator.add($iterator_{level}$);
    end
end
Input: iterator
Result: Iterator points to the next KV-pair and freshness of the iterator
Function next($iterator$)
    /* Forward all iterators pointing to the current key */
    foreach $iterator_{level} \in iterator$ where $iterator_{level}.key = iterator.key$ do
        next($iterator_{level}$);
        if $iterator_{level}.state \neq fresh$ then
            return $iterator_{level}.state$
        end
    end
    /* Find the level iterator pointing to the lowest key */
    for $i = 0$ to $number\_levels$ do
        $iter \leftarrow iterator[i]$;
        if $iter.state \neq fresh$ then
            return $iter.state$
        end
        if $key_{lowest} > iter.key$ then
            $key_{lowest} \leftarrow iter.key$;
            $level \leftarrow i$
        end
    end
    iterator.currentLevel($i$);
    return $fresh$
end
```

Algorithm 5: Restore algorithm of SPEICHER

```
Input: Manifest File
Result: Restored KV-store
/* Get the counter value of the first record in the
   manifest and check that the first record is an inital
   record                                               */
$counter \leftarrow Manifest.firstCounterValue$;
/* Iterate over all records in the Manifest             */
foreach $record_{encrypted} \in Manifest$ do
    $record \leftarrow decrypt$;
    $hash \leftarrow$ hash($record$);
    /* Check the records hash and counter value, if they
       do not match, report an error to the client      */
    if $hash \neq record.hash$ then
        return $Hash\ does\ not\ match$
    end
    if $counter \neq record.counter$ then
        return $Counter\ does\ not\ match$
    end
    /* If hash and counter match apply the change to the
       KV-store                                         */
    apply($record$);
    inc($counter$);
end
/* Check if the last counter in the Manifest matches the
   trusted counter, if not report an error to the client
   */
if $counter \neq tusted\_counter_{Manifest}$ then
    return $Counter\ does\ not\ match$
end
/* Get the current WAL and its initial counter value from
   the Manifest                                         */
$counter \leftarrow Manifest.firstWALCounter$;
/* Apply each record of the WAL to the KV if the counter
   and hash are correct, similar to the Manifest        */
foreach $record_{encrypted} \in WAL$ do
    $record \leftarrow decrypt$;
    $hash \leftarrow$ hash($record$);
    if $hash \neq record.hash$ then
        return $Hash\ does\ notmatch$
    end
    if $counter \neq record.counter$ then
        return $Counter\ does\ not\ match$
    end
    apply($record$);
    inc($counter$);
end
/* Check if the last counter value is the same as the
   trusted counter                                      */
if $counter \neq trusted\_counter_{WAL}$ then
    return $Counter\ does\ not\ match$
end
/* KV-store was successfully restored and no integrity
   or rollbacks problem were found                      */
return $Success$
```

Algorithm 6: Compaction algorithm of SPEICHER

```
Input: SSTable file to be compacted one from level$_n$
Result: Multiple SSTable files for level$_{n+1}$
// Creating an Iterator over the higher level SSTable file create a new file
and a new data block
$iterator_n \leftarrow$ createIterator($SSTable_n$);
$NewSSTable \leftarrow$ createNewSST();
$block \leftarrow$ createNewBlock();
$last\_key \leftarrow$ iterator$_n$.key $-$ 1;
// As long as their are KV-pairs remaining in the SSTable open the
SSTable file in the next level which has the range of the smallest possible
next key based on the last key compacted. while
has_next($iterator_n$) do
    $SSTable_{n+1} \leftarrow$ findSSTFile($n+1, last\_key+1$);
    $iterator_{n+1} \leftarrow$ createIterator($SSTable_{n+1}$);
    // As long as the currently open SST$_{n+1}$ file has KV-pairs find the
    smaller next key of SST$_n$ and SST$_{n+1}$ file. If both have the same next
    key choose from SST$_n$ file.
    while has_next($iterator_{n+1}i$) do
        $iterator_{min} \leftarrow$ min($iterator_n, iterator_{n+1}$);
        // test if the key value is still fresh, that is check the hash of the
        block compare in the SSTable file hash footer and check
        against the Manifest
        if $iterator_{min} \neq fresh$ then
            // If the key value is not fresh return error to client
            return $iterator_{min}.state$
        end
        // Add key to block, if the block is then over the size limit for
        blocks calculate a hash add the hash to the footer of the new
        file and write the block to persistent storage, and create a new
        block
        block.add($iterator_{min}.kv$);
        if size($block$) $> block\_size\_limit$ then
            $hash \leftarrow$ hash($block$);
            $encrypted\_block \leftarrow$ encrypt($block$);
            NewSSTable.write($encrypted_block$);
            NewSSTable.addHash($hash$);
            // If the file reaches the size limit after an append, write
            the footer to the storage and create a new SSTable
            if size($NewSSTable$) $> SSTable\_size\_limit$ then
                NewSSTable.writeFooter();
                $NewSSTable \leftarrow$ createNewSST();
            end
            $block \leftarrow createNewBlock$;
        end
        $last\_key = iterator_{min}.key$;
        next($iterator_{min}$);
    end
end
// After compaction, flush the block & write the footer.
$hash \leftarrow$ hash($block$);
$encrypted\_block \leftarrow$ encrypt($block$);
NewSSTtable.write($encrypted\_block$);
NewSSTable.addHash($hash$);
NewSSTable.writeFooter();
// Write the changes to the Manifest file.
Manifest.remove(SST$_n$,SST$_{n+1}$inrangeofSST$_n$);
Manifest.add($\forall$NewSSTfile);
```

References

[1] Botan Library. https://botan.randombit.net/. Last accessed: Jan, 2019.

[2] Intel DPDK. http://dpdk.org/. Last accessed: Jan, 2019.

[3] Intel, "SGX documentation: sgx create monotonic counter". https://software.intel.com/en-us/sgx-sdk-dev-reference-sgx-create-monotonic-counter/. Last accessed: Jan, 2019.

[4] Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx. Last accessed: Jan, 2019.

[5] RocksDB Benchmarking Tool. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. Last accessed: Jan, 2019.

[6] I. Anati, S. Gueron, P. S. Johnson, and R. V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[7] ARM. Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009. Last accessed: Jan, 2019.

[8] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[9] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[10] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[11] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[12] P. Bhatotia, A. Wieder, R. Rodrigues, F. Junqueira, and B. Reed. Reliable data-center scale computations. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

[13] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[14] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[15] V. Costan and S. Devadas. Intel SGX Explained, 2016.

[16] CRN. The ten biggest cloud outages of 2013. https://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm, 2013. Last accessed: Jan, 2019.

[17] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[18] B. Dickens III, H. S. Gunawi, A. J. Feldman, and H. Hoffmann. Strongbox: Confidentiality, integrity, and performance using stream ciphers for full drive encryption. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[19] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 2016.

[20] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[21] D. Garg and F. Pfenning. A proof-carrying file system. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[22] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[23] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proceed-*

ings of the Network and Distributed System Security Symposium (NDSS), 2003.

[24] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[25] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[26] M. Honda, G. Lettieri, L. Eggert, and D. Santry. PASTE: A network programming interface for non-volatile main memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[27] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[28] Intel Storage Performance Development Kit. http://www.spdk.io. Last accessed: Jan, 2019.

[29] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[30] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)*, 2019.

[31] Kinetic Data Center Comparison. https://www.openkinetic.org/technology/data-center-comparison. Last accessed: Jan, 2019.

[32] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij. Integrating Remote Attestation with Transport Layer Security. 2018.

[33] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: Secure offline data access using commodity trusted hardware. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[34] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[35] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer. Haft: Hardware-assisted fault tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[36] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, 2017.

[37] D. Kuvaiskii, O. Oleksenko, P. Bhatotia, P. Felber, and C. Fetzer. Elzar: Triple modular redundancy using intel avx. In *proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[38] A. W. Leung, E. L. Miller, and S. Jones. Scalable security for petascale parallel file systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2007.

[39] LevelDB. http://leveldb.org/. Last accessed: Jan, 2019.

[40] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[41] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[42] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. *Depot: Cloud Storage with Minimal Trust*. 2011.

[43] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI)*, 2000.

[44] K. Mast, L. Chen, and E. Gün Sirer. Enabling Strong Database Integrity using Trusted Execution Environments. 2018.

[45] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security)*, 2017.

[46] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel. Qapla: Policy compliance for database-backed systems. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[47] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[48] E. L. Miller, D. D. Long, W. E. Freeman, and B. Reed.

Strong Security for Network-Attached Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[49] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[50] A. Narayan and A. Haeberlen. DJoin: differentially private join queries over distributed databases. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[51] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security)*, 2016.

[52] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, and C. Fetzer. Fex: A software systems evaluator. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[53] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[54] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). In *Acta Inf.*, 1996.

[55] M. Orenbach, M. Minkin, P. Lifshits, and M. Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the 12th ACM European ACM Conference in Computer Systems (EuroSys)*, 2017.

[56] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[57] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*, 2011.

[58] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Crash consistency. *ACM Queue*, 2015.

[59] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2011.

[60] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[61] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A Secure Database using SGX. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[62] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communication of ACM (CACM)*, 1990.

[63] P. Raju, S. Ponnapalli, E. Kaminsky, G. Oved, Z. Keener, V. Chidambaram, and I. Abraham. mlsm: Making authenticated storage faster in ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[64] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[65] RocksDB | A persistent key-value store. https://rocksdb.org/. Last accessed: Jan, 2019.

[66] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[67] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security)*, 2012.

[68] F. Schuster, M. Costa, C. Gkantsidis, M. Peinado, G. Mainar-ruiz, and M. Russinovich. VC3 : Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[69] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[70] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[71] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium (USENIX Security)*, 2016.

[72] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[73] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.

[74] Trusted Computing Group. TPM Main Specification. https://trustedcomputinggroup.org/tpm-main-specification, 2011. Last accessed: Jan, 2019.

[75] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[76] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases (VLDB)*, 2013.

[77] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*, 2015.

[78] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[79] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the salus scalable block store. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[80] C. Weinhold and H. Härtig. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[81] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[82] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.