TwinStore: Secure Key-Value Stores Made Faster with Hybrid Trusted/Untrusted Storage

Xiang Li[1,3][0000−0001−8906−0832],
Huanchen Zhang[1,2][0009−0001−4821−1558], and
Mingyu Gao[1,2][0000−0001−8433−7281]✉

[1] Tsinghua University, Beijing, China
[2] Shanghai Qi Zhi Institute, Shanghai, China
[3] China Telecom eSurfing Cloud (State Cloud), Beijing, China

`lixiang20@mails.tsinghua.edu.cn, {huanchen,gaomy}@tsinghua.edu.cn`

Abstract. Key-value (KV) stores are one of the most prominent data storage engines in many cloud services, including those that keep sensitive user information and must be protected against privileged attackers in public clouds. While modern hardware processors could offer isolated execution environments during processing, existing secure KV stores still need to use software-based protection to ensure confidentiality, integrity, and freshness for external data I/O to disks. The recent development of hardware-based trusted I/O and disks provides a new way to implement secure KV stores and obviate software encryption. However, we find that although trusted I/O simplifies freshness enforcement, directly putting all data to the trusted disk is not optimal due to the repetitive encryption agnostic to the application behaviors. We thus propose TwinStore to combine the benefits of the software and hardware, by only storing the metadata associated with freshness in the trusted disks to minimize the performance overheads. With our prototypes on two KV store structures, TwinStore outperforms the software-only and hardware-only designs, by 18.5× and 1.1×, respectively.

Keywords: KV store · trusted execution environment · trusted storage.

1 Introduction

As cloud computing advances rapidly, commercial workloads have been moving from on-premise servers to public clouds. Many cloud services leverage persistent key-value (KV) stores as their underlying data storage engines. Besides performance and cost, a new design goal of security emerges for such KV stores as they may keep sensitive information such as user credentials and medical records in the untrusted and unsafe cloud environment.

Trusted execution environments (TEEs) offered in modern Intel, AMD, and ARM processors [1,4,13,14,17] provide a promising way to protect sensitive data. Prior research [5,18–20,40,41,45] has built secure KV stores with TEEs like Intel

SGX [13], incorporating *software-based* protection when data move outside the TEE protection regime, e.g., disk I/O. Recently, the scope of TEEs has enlarged to the virtual machine (VM) level [4, 14, 17], which can naturally include the I/O devices, particularly with the new TEE-IO specifications [2,15] and trusted storage (Section 2.2) that can transparently perform *hardware-based* protection on all I/O data transfers. This motivates us to rethink how to implement efficient and secure KV stores with the best choices of data protection methods.

To our surprise, we find that *the hardware-based approach is not necessarily always better than the software one.* Each option has its own drawbacks (Section 3). If we directly put all KV store data into the TEE-IO-capable trusted storage, although it facilitates efficient freshness protection, *repetitive encryption and decryption* occur along the I/O path when transferring data from the memory along the PCIe bus to the disk. This is because different hardware components use different encryption keys that must be managed in different ways to match their specific requirements. The software-based method might be more efficient and flexible here, as it can leverage the multiple CPU cores and the AES-NI instruction extension [10], and can easily saturate the I/O bandwidth. Nevertheless, to ensure data freshness and prevent replay and rollback attacks, software methods need to rely on some kinds of *trusted counters*, which are not available in many TEEs or have significant performance overheads.

In this work, we propose an efficient and secure KV store design, TwinStore, which combines the advantages of both hardware- and software-based implementations. *TwinStore identifies the minimal components for freshness guarantee in the secure KV store, and only puts their data into the hardware-based trusted disks, while using normal untrusted disks with software-based authenticated encryption for the rest majority of data.* This approach allows us to minimize the amount of data suffering the repetitive encryption overheads, while also relieving the design from the difficulty of maintaining trusted counters for freshness enforcement. Note that such identification and separation are straightforward given an existing secure KV store design, and in most cases even simplify the original design as we can now rely on hardware for freshness protection.

We have demonstrated the feasibility and benefits of TwinStore on two KV stores, RocksDB [30] and LotusDB [24], based on log-structured merge (LSM) trees [34] and B+ trees, respectively. The software-only implementation suffers from significant overheads for trusted counter management due to the limitation of current TEEs, while leveraging trusted disks alleviates this issue. Overall, by combining the two approaches, TwinStore achieves 3.4× and 1.1× speedups for RocksDB, and 18.5× and 1.06× for LotusDB, over the two baselines. TwinStore is open sourced at `https://github.com/tsinghua-ideal/twinstore`.

2 Background and Related Work

2.1 Trusted Execution Environments

Trusted execution environments (TEEs) provide isolated execution regions inside untrusted servers [1, 4, 14, 17, 27]. A remote client can connect to the TEE and verify its status through remote attestation [3] to establish trust, and then securely offload computations to it. The confidentiality and integrity of data and code in the TEE are guaranteed even against a privileged attacker on the server.

The CPU is the root of trust in building the isolated hardware region. Besides isolation against the concurrently running contexts inside the CPU, the content in the TEE is transparently encrypted by a memory encryption engine when leaving the CPU to the off-chip memory. The logical scope of the trusted region varies in different TEEs. While Intel SGX provides in-process isolated regions, AMD SEV [17], Intel TDX [14], and ARM CCA [4] choose to protect a whole virtual machine (VM). Nowadays, the industry prefers TEEs with VM-level isolation, known as *confidential VMs* (CVMs), because they have a clear security boundary consistent with the management scope of cloud servers. Even if the hypervisor is compromised, the security of CVMs is still guaranteed.

Previously, data should be encrypted and authenticated by software before leaving TEEs for persistence. With the prosperity of CVMs, it is natural to consider also covering the I/O devices in the trusted domain of the VM. The PCIe-5 specification introduces integrity and data encryption (IDE) [35], which guarantees confidentiality, integrity, and freshness for PCIe packets against attackers with physical access to the I/O bus. Some device manufacturers have released hardware security modules for IDE [46]. However, IDE is not originally designed for a threat model with untrusted hypervisors [42]. Therefore, TEE-IO specifications, such as SEV-TIO [2] and TDX-TEE-IO [15], are being proposed. TEE-IO formulates the device attestation model and describes the necessary capabilities that trusted devices should have to establish secure packet channels between PCIe stubs. When transferring data across the I/O bus and devices, each hardware component adopts hand-over-hand authenticated encryption, to ensure data confidentiality, integrity, and freshness even with adversarial hypervisors (Section 3). After the attestation between the device and an attested TEE instance, the client knows it is communicating with the correct instance of the device. Although there are not yet commercial storage devices supporting TEE-IO, TEE-IO is promising and expected to appear soon.

2.2 Hardware-Based Trusted Storage

Although TEE-IO protects the physical channel (i.e., PCIe) between the host processor and the storage device, the storage device still requires additional protection mechanisms against both internal (software) attacks and external physical (hardware) attacks. Therefore, trusted storage devices should feature self-encrypting disk drives [31, 36] for fine-grained access control. Concretely, after connecting to the device, the user should set the password originally set by the device manufacturer [43]. Then only the user can unlock and access the device with the password. The TCG Opal specification [44] allows multiple passwords to be set, and hence a storage device can be divided into multiple parts. Each part is encrypted with a different disk encryption key (DEK) and can be erased independently of the others. The DEKs should be derived from the passwords [28]. When combined with TEEs, it is the TEE instance that sets the password for its part, after which even a privileged attacker cannot reset the password. The device then encrypts each part with the corresponding DEK [44]. However, self-encrypting is insufficient to defend against physical attacks that undermine data freshness. Hence, trusted storage devices should also incorporate hardware defenses for end-to-end security [48].

2.3 Secure Key-Value Stores

Key-value (KV) store is an important data service that has been integrated with TEEs [5, 18–20, 40, 41, 45]. Previous designs mainly targeted the defects in conventional TEEs like Intel SGX, including limited trusted memory capacity [18, 19, 40, 45], high domain switch overheads [19], slow Merkle-tree verification [29], and fragile trusted counters (if existing) for data freshness [6, 18]. In this work, we focus on secure KV stores on persistent storage, whose structures rely on log-structured merge (LSM) trees [34], B+ trees, etc.

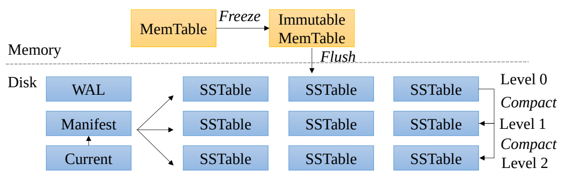

Fig. 1: An LSM-tree-based KV store (RocksDB [30]).

A normal (insecure) LSM-tree-based KV store [9, 30] contains in-memory tables (MemTables), sorted strings tables (SSTables) on disks, write-ahead log (WALs) files, and a Manifest, as shown in Figure 1. For `put` requests, the items are sequentially logged to the WAL for crash recovery and temporarily stored in memory using the MemTable. As the MemTable fills up, the content is first made immutable (to avoid blocking future requests), then converted into a SSTable, and dumped to the disks. SSTables are hierarchically organized in multiple levels. When the size of a level grows over a threshold, background *compaction* merges several files into a larger one to the next level. For `get` requests, the KV store first searches the MemTable, and then the SSTables level by level, for the key. The version changes of SSTables are recorded in the Manifest.

TWEEZER [18] and Speicher [5] are representative secure KV stores based on LSM trees. Take TWEEZER as an example. It encrypts and authenticates the KV pairs in the SSTables for confidentiality and integrity. It further uses different cryptographic keys (held in TEE memory) for each SSTable to defend against rollback attacks, which is secure because SSTables are immutable and each index key is unique in each SSTable. To protect WAL files, TWEEZER utilizes classic hash chains [37, 38] and lets the client keep the latest cryptographic key that is the same as the one in the trusted memory.

Apart from LSM trees, KV stores can be built with other structures like B+ trees [24,49]. B+ trees have more stable read performance, while LSM trees have higher write throughput. To combine the best of both, similar to [16], LotusDB [24] replaces SSTables with value log files, and uses B+ trees to index values in the files. If we put a B+tree-based KV store into a TEE, besides encryption and authentication when data leave the TEE, Merkle trees are typically needed for data freshness when doing in-place update [41].

2.4 Threat Model

We assume a group of clients outsourcing their data to a remote KV store service in a CVM on an untrusted server. The TEE-capable hardware processor and trusted storage on the server are trusted, which means they contain no internal bugs and defend against physical attacks (e.g., physically rolling back the data). The clients share a password, which is used by the TEE instance to access the trusted storage. The guest OS and the KV store inside the CVM are also trusted. We do not need the clients to manage metadata for storage encryption (e.g., counters for defending against rollback attacks). We do not assume the availability of distributed trusted counter services. We assume a powerful malicious attacker, who controls all privileged software on the server including the hypervisor, and wants to steal sensitive information or fool the clients with incorrect or stale results. The attacker can arbitrarily observe and tamper with the code and data in untrusted devices, including memories, disks, and networking. However, any curiously or maliciously adversarial behaviors to the trusted CVM will fail. Moreover, the attacker cannot directly query the KV store due to the lack of access permission (e.g., passwords). Alleviation of side channels [8,11,21,22,25] for CVMs is considered orthogonal. Although our system supports secure crash recovery, denial-of-service attacks are also out of scope.

3 Motivation

The development of TEE-IO-capable trusted storage offers an alternative way of building secure KV stores. We could put a standard KV store system into CVMs and trusted storage, instead of relying on software-based encryption, authentication, and freshness enforcement. It becomes a natural question to consider which approach is better, adopting TEE-IO *hardware* or using pure *software* implementations. Our study reveals that *neither one is ideal*, but there is an opportunity to *combine the two methods to achieve the best of both worlds.*

Issue of trusted storage: repetitive encryption and decryption. Intuitively, the CVM, TEE-IO, and trusted storage respectively protect the data in the processor and memory, along the PCIe links, and inside the disks. As the data flow from the CVM memory to the I/O storage, data confidentiality and integrity are naturally guaranteed through hardware-based encryption and authentication. Data freshness protection is ensured by the use of temporary session keys and counters during I/O transfers, as well as by the access control and physical protection when data are stored in trusted storage (Section 2.2). However, there would be repetitive encryption and decryption as shown in Figure 2. Data are first decrypted by the encryption engine in the memory controller, and then encrypted before arriving at one PCIe endpoint. After transferred over PCIe, they are decrypted again at the other endpoint, and finally encrypted by the disk controller encryption engine to store in the disk.

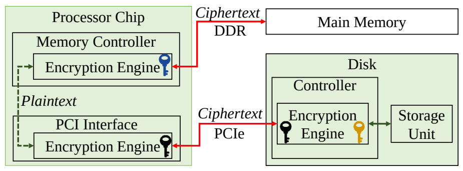

Fig. 2: Encryption/decryption as data flow between the memory and the disk. Green denotes the trusted parts.

The root cause of such repetitive encryption and decryption is the different keys used in different phases and managed by different components. We believe this is a more fundamental limitation rather than suboptimal implementation, because of *the inconsistent characteristics across the phases that hinder the unification of the keys.* According to the TEE-IO specification, PCIe keys are negotiated *per session*, which means that as the device boots, a new pair of keys are used and they cannot be programmed by users. However, for disk encryption, the same key should be used *per user* without changing across sessions. As for memory encryption, the key is generated by the CPU through a hardware digital random number generator (HW-DRNG), and is lost once the CPU resets [14]; thus it is not user-programmable, either. Unifying these keys is hence non-trivial without significantly changing the entire system architecture.

Issue of software: difficulty of freshness enforcement. If we fully rely on software-based security techniques without utilizing TEE-IO and trusted storage, guaranteeing freshness is the most tricky thing. Previous designs rely on several methods. Some use in-TEE trusted counters [5], which take 60 to 250 ms for every increment and are easy to wear out [7, 26]. Others implement distributed secure counter services, which are complicated and also have high overheads [26]. Another way is to let the remote client manage the cryptographic keys used to encrypt and authenticate the WAL (Section 2.3). This is impractical considering the synchronization needed when log truncation occurs or multiple clients collaboratively access the KV store. If Merkle trees are used to protect freshness [5, 41], there would be even more significant performance overheads.

Opportunity: software is not necessarily worse than hardware if freshness is not considered. We observe that, in terms of encryption and authentication, software-based solutions are not definitely worse than hardware-based ones. First, software can also utilize hardware acceleration instructions like AES-NI [12], reaching throughput of several GB/s per core on AES-GCM-

256 [10]. Second, modern servers are typically equipped with tens to hundreds of cores, so encryption and authentication can be fully parallelized at the data-block granularity. In contrast, the number of hardware encryption engines is typically limited. A PCIe 5.0 device with 32 lanes, even not considering the encryption engine limit and repetitive encryption overheads, has 128 GB/s bandwidth [33]. The encryption and authentication throughput required to saturate this bandwidth can be easily satisfied by parallel software with multiple cores. Third, generally speaking, the higher in the system stack we apply encryption, the more flexibility we have. With application-level encryption, the application developers and users can apply any encryption code (e.g., choosing a specific cryptographic algorithm) to any particular data they need. Therefore, the software approach may be combined with the hardware method to achieve both ones' advantages.

4 TwinStore Design

To fix the thorny freshness issue in the software method and mitigate the repetitive encryption issue in the hardware method, we propose TwinStore to combine the best of both worlds. The key idea is that, *with a KV store, we identify the minimal components associated with freshness enforcement and put them to the hardware-based trusted storage, while the other components are directed to the normal untrusted storage protected by software encryption and authentication.* The minimization reduces the amount of data that suffer from repetitive encryption and decryption in hardware, while the difficulty of ensuring freshness in software is avoided. We illustrate how we instantiate the idea on an LSM-tree-based KV (i.e., RocksDB [30]) in Section 4.1 and a B+tree-based KV (i.e., LotusDB [24]) in Section 4.2.

4.1 LSM-Tree-Based TwinStore

To secure an LSM-tree-based KV store, we follow the design of TWEEZER [18] (Section 2.3). The minimal part for freshness enforcement includes the WAL and the Manifest files, as well as the Current file that points to the Manifest of the latest version of SSTables. Hence, as shown in Figure 3, we assign the above components to the trusted disks, while the other components, including the SSTables, are stored in the untrusted storage with software-based protection. Like TWEEZER, we use a KeyMatrix to hold the cryptographic keys corresponding to each SSTable. All volatile states including the KeyMatrix are stored in the CVM trusted memory. The Manifest files are responsible for persisting the cryptographic keys in the KeyMatrix, delegating protection to trusted storage.

For a `put` request, the WAL sequentially saves the log entry for the request and the MemTable stores the KV pair. As we flush the log entry to the WAL, there is no need for software-based encryption, authentication, or using hash chains [18]. Instead, the log entry stays in plaintext from the software perspective, and the hardware, i.e., trusted storage, would transparently protect the entry when transferred along the PCIe bus and stored in the trusted disk. On the

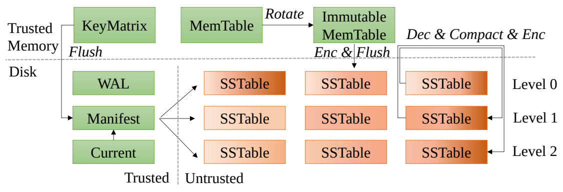

Fig. 3: Applying TwinStore to RocksDB [30].

other hand, when the MemTable is flushed to the untrusted disk as an SSTable, it needs software-based protection of encryption and authentication, associated with a unique key in the KeyMatrix. A `get` request follows the same process that first searches the MemTable and then the encrypted SSTables. Given that the keys in the KeyMatrix are organized in the same way as the SSTables, we can easily get the corresponding key to decrypt each SSTable during search.

As shown in Figure 4, an SSTable contains various types of blocks. The footer is used to locate the index block and the meta index block that points to the filter block (for bloom filters). Each entry in the index block refers to a data block by the offset. The encryption granularity in the SSTable is a block rather than a KV pair [18]. We use AES-GCM-256 to encrypt each block except for the footer, where the MACs are stored. In particular, the MAC computation needs to involve the data block offset, to prevent attackers from substituting another (MAC validated) data block in the same SSTable.

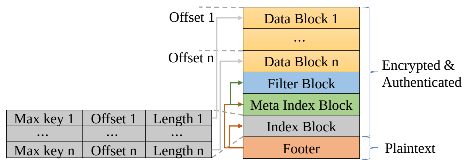

Fig. 4: The structure of an encrypted SSTable.

When SSTables undergo compaction, they are first loaded into the CVM trusted memory and decrypted. The compaction is done on plaintext KV pairs as usual. The compacted output SSTables are associated with new cryptographic keys, which requires updates of the KeyMatrix, Manifests, and Current.

For crash recovery, the trusted disks first perform mutual attestation with the CVM to build a secure channel. After that, only the CVM having the correct password can access the (part of) trusted storage. Given that the freshness of WAL, Manifests, and Current is well protected, the rest of recovery is similar to the normal one, except that a KeyMatrix should be built from the Manifest.

Security analysis. The states in the trusted storage are protected by the CVM, TEE-IO, and trusted disks, with data confidentiality, integrity, and freshness (i.e., no replay or rollback) ensured by the end-to-end flow of TEE-IO (Section 2.1). The only attack point is the SSTables in the untrusted storage outside the CVM and TEE-IO region. Any snooping and tampering is prevented by authenticated encryption (i.e., AES-GCM-256), so the attacker can only try to (1) replace one SSTable as a whole with another, (2) deceive TwinStore into believing the SSTable does not exist, (3) replace one block with another, or (4) deceive TwinStore into believing the SSTable does not contain a certain block. To defend against (1) and (2), TwinStore detects the attacks using the KeyMatrix when it obtains an SSTable and verifies its block through MAC computation. Since the code in the CVM knows which SSTable to search on, it uses the right key to decrypt and verify the block. If the attacker provides a block from another SSTable, the verification will fail. For (3) and (4), although we do not conduct an HMAC computation on the entire SSTable as in TWEEZER [18], our block-level protection is sufficient. In an SSTable, there is only one block of each type except for data blocks. The attacker cannot, for example, replace an index block with a meta index block. Otherwise, the CVM code will fail to decode the block. For data blocks, since we involve the offset in the MAC computation, the attacker can neither replace a data block with another nor delete the block without modifying the index block, which is also impossible.

4.2 B+Tree-Based TwinStore

Instead of searching on multiple SSTables, LotusDB [24] supports using the B+ tree to index the values in the files called value logs, and thus improves the read performance. We identify that the minimal part for freshness enforcement in LotusDB includes the WAL and the B+ tree indexes. Hence, as in Figure 5, we assign these components to the trusted TEE-IO disks, and the value logs are stored in the untrusted storage. All volatile states are in the CVM memory.

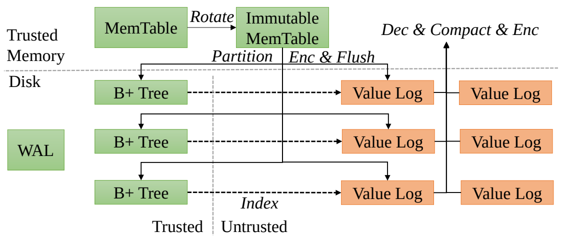

Fig. 5: Applying TwinStore to LotusDB [24].

For a `put` request, similar to Section 4.1, the WAL persists a log entry to the trusted storage without software encryption and authentication. LotusDB partitions the filled-up MemTable by hashing the keys, so each partition can easily be indexed using a B+ tree and written to disks in parallel. The position in the B+ tree indicates which value log and at what offset in this log the KV pair resides, since there are multiple value logs in one partition. In TwinStore, we use AES-GCM-256 to encrypt the value logs at the granularity of each KV pair. The computed MAC tags are stored along with the positions in the B+ trees. For a `get` request, TwinStore first searches the MemTable, and then the value logs through the B+ tree indexes. Similar to SSTables, value logs may also contain duplicated keys and need compaction, which happens similarly to Section 4.1 in plaintext in the CVM trusted memory. Crash recovery is simple based on the fully protected WAL and B+ tree indexes.

Security analysis. Similar to Section 4.1, the only possible attack is replacing the requested block with another block. This attempt can be easily detected as the MAC tags are stored in the trusted storage.

Discussion. We provide guidelines for identifying critical components in other KV stores. First, the WAL, which is commonly used, should be stored in trusted disks, as it is typically the ground truth for crash recovery. Second, the structures holding the data should be assigned to untrusted disks due to the large volume. Finally, the persistent index should go to trusted disks, such as the Manifests (and Current) in RocksDB and the B+ trees in LotusDB.

5 Evaluation

5.1 Experimental Setup

Platform. We evaluate TwinStore on a server equipped with an AMD EPYC Milan processor (64 cores) which supports AMD SEV-SNP CVMs [39], 256 GB RAM, and a 1.6 TB NVMe SSD.

Workloads. We evaluate both RocksDB and LotusDB. For RocksDB, we use `db_bench`, a benchmark tool with rich configurations. Unless otherwise specified, the default configurations are: number of client threads = 48, (random) read-write ratio = 9:1, block size = 32 kB, key size = 16 B, value size = 1 kB, block cache size = 512 MB, and number of requests per client = 5×10^6. For LotusDB, we directly evaluate sequential/random `put`/`get` since it does not provide an off-the-shelf benchmark tool. The default configurations are: key size = 16 B, value size = 32 B, read-write ratio = 1:1, number of partitions = 92, and number of requests per client = 4.8×10^6.

Baselines. We compare TwinStore to the software-only (SW-only) and hardware-only (HW-only) approaches. SW-only is similar to TWEEZER [18], but now puts the KV store into a CVM instead of the legacy Intel SGX TEE, therefore many of the optimizations in TWEEZER can be simplified due to the larger trusted memory capacity. However, this approach suffers from the difficulty of managing freshness counters. SEV does not have non-volatile trusted counters. Based on [26], we simulate a conservative 1 ms latency of counter update when persisting each log entry. We do not compare with previous secure KV designs (Section 2.3) due to their different target platform, i.e., Intel SGX. HW-only directly puts the KV store into a CVM with TEE-IO-capable trusted disks. However, to our best knowledge, no commercial storage currently supports trusted disks. Therefore, we conservatively simulate this system. We utilize `dm-crypt` in the `cryptsetup` tool [47] in the Linux kernel to transparently encrypt and authenticate persistent data, configuring it with AES-GCM-256. This is reasonable because: (1) authenticated encryption of `cryptsetup` is also transparent to applications; (2) the granularity of encryption can be set to 4 kB as the disk block size; (3) decrypted data blocks also utilize OS-level caching (i.e., Linux page cache), avoiding redundant decryption; (4) the transparent authenticated encryption runs in parallel and fully utilizes multi-core [32] and AES-NI acceleration, thus matching or even exceeding the speed of the hardware encryption engines. We remark that if using an ASIC/FPGA to simulate the encrypted PCIe bus, the latency would be higher since in this case both PCIe endpoints involve encryption/decryption. The client does not manage cryptographic keys.

5.2 Results

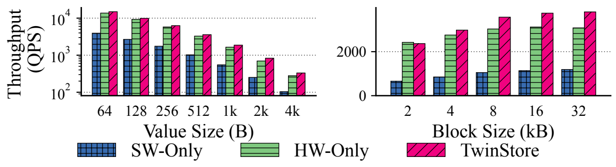

Fig. 6: Performance of RocksDB as the value/block sizes vary.

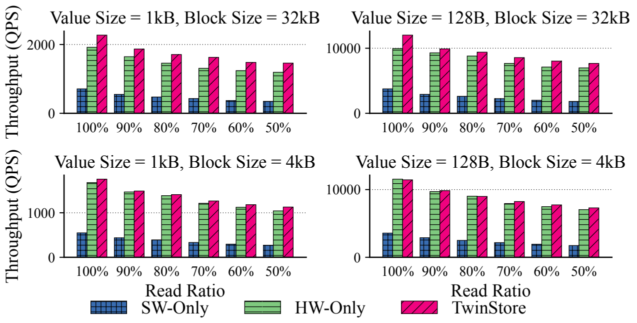

Fig. 7: Performance of RocksDB as the read-write ratio varies.

RocksDB result analysis. Figure 6 shows the comparison among SW-only, HW-only, and TwinStore, with various value sizes and SSTable block sizes. When the value size increases, the improvement of TwinStore over HW-only ranges from 6% to 16%. Smaller value sizes have less improvement, because the page cache in the baseline can keep more KV pairs, while we encrypt at the block granularity without optimizing for block-level caching. On the other hand, as the block size increases, the improvement grows from −2% (worse) to 19%. For software encryption/decryption in TwinStore, larger blocks can amortize the overheads of each cryptographic routine invocation and result in better cache utilization [23]. Besides, because the SW-only design increments the trusted counter for each new log entry, it incurs significant slowdown from 3.8× to 3.2× compared to TwinStore, as the value size grows. The slowdown decreases due to the reduced frequency of writing logs when the value size becomes larger. However, as the block size increases, the slowdown of SW-only over TwinStore is relatively stable around 3.4×.

Besides, different numbers of client threads from 1 to 64 affect the speedup of TwinStore over HW-only from 10% worse to 18% better. Our current implementation of software encryption/decryption has less parallelism than `dm-crypt` in HW-only. More request-level parallelism helps fully utilize the multiple cores for TwinStore, while `dm-crypt` suffers from more parallel scheduling cost [32].

Figure 7 shows the comparison as the read-write ratio varies. With more writes, the speedups of TwinStore over both HW-only and SW-only become slightly better. This is because TwinStore does not optimize for caching decrypted blocks, while HW-only benefits from the Linux page cache when serving reads. On the other hand, SW-only will write logs more frequently if there are more writes, exacerbating the slowdown over TwinStore.

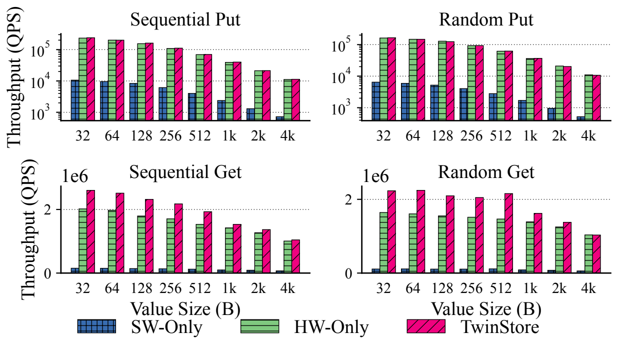

Fig. 8: Performance of LotusDB as the value size varies.

LotusDB result analysis. The encryption granularity is a KV pair now, so for both sequential and random `get`, we see significant speedups of TwinStore over HW-only as the value size shrinks in Figure 8, which are 3% to 22% for sequential `get` and up to 28% for random `get`, showing the necessity of flexibility in the encryption granularity. However, for both sequential and random `put`, TwinStore and HW-only have comparable performance, because `put` is performed in a batched manner, and `dm-crypt` can encrypt a batch at once.

The speedup of TwinStore over SW-only is larger for LotusDB because SW-only additionally uses Merkle trees to protect the freshness of the B+ trees, while for RocksDB SW-only uses the KeyMatrix to store the latest keys in the trusted memory. For sequential/random `put`, TwinStore achieves about 20× speedup over SW-only. For sequential/random `get`, the speedup slightly decreases to about 18× because writing logs is not needed for SW-only.

6 Conclusions

In this paper, we comprehensively compare two approaches to implementing secure KV stores, through using software encryption and authentication, or relying on hardware-based protection offered by recent trusted I/O and disks. The software methods are inefficient to guarantee data freshness, while the hardware methods suffer from repetitive encryption and decryption cost. We thus propose TwinStore to combine the best of both worlds. We evaluate TwinStore in various configurations and show it improves over the two baseline designs.

Acknowledgments. This work is partially supported by the Shanghai Qi Zhi Institute Innovation Program (SQZ202314 & SQZ202406).

References

1. Alves, T.: TrustZone: Integrated Hardware and Software Security. White paper (2004)
2. AMD: AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization. `https://www.amd.com/system/files/documents/sev-tio-whitepaper.pdf` (2023)
3. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative Technology for CPU Based Attestation and Sealing. In: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy. vol. 13, p. 7 (2013)
4. ARM: Arm Confidential Compute Architecture. `https://developer.arm.com/documentation/den0125/0200/?lang=en` (2021)
5. Bailleu, M., Thalheim, J., Bhatotia, P., Fetzer, C., Honda, M., Vaswani, K.: SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In: 17th USENIX Conference on File and Storage Technologies (FAST 19). pp. 173–190 (2019)
6. Brandenburger, M., Cachin, C., Lorenz, M., Kapitza, R.: Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 157–168 (2017)
7. Cen, S., Zhang, B.: Trusted Time and Monotonic Counters with Intel Software Guard Extensions Platform Services. misc at: https://software. intel. com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services. pdf (2017)
8. Gast, S., Juffinger, J., Schwarzl, M., Saileshwar, G., Kogler, A., Franza, S., Köstl, M., Gruss, D.: SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 2256–2272 (2023)
9. Google: LevelDB. `https://rocksdb.org/` (2011)

10. Gueron, S.: Constructions based on the AES Round and Polynomial Multiplication That Are Efficient on Modern Processor Architectures. In: The Third NIST Workshop on Block Cipher Modes of Operation (2023)
11. Hetzelt, F., Buhren, R.: Security Analysis of Encrypted Virtual Machines. ACM SIGPLAN Notices **52**(7), 129–142 (2017)
12. Hofemeier, G., Chesebrough, R.: Introduction to Intel AES-NI and Intel Secure Key Instructions. Intel, White Paper **62**, 6 (2012)
13. Intel: Intel Software Guard Extensions (Intel SGX) Developer Guide (2018)
14. Intel: Intel TDX. `https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html` (2020)
15. Intel: Intel TDX Connect TEE-IO Device Guide. `https://cdrdv2-public.intel.com/772642/whitepaper-tee-io-device-guide-v0-6-5.pdf` (2023)
16. Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S.H., Choi, Y.r.: SLM-DB:Single-Level Key-Value Store with Persistent Memory. In: 17th USENIX Conference on File and Storage Technologies (FAST 19). pp. 191–205 (2019)
17. Kaplan, D.: Protecting VM Register State with SEV-ES. White paper, Feb (2017)
18. Kim, I., Kim, J.H., Chung, M., Moon, H., Noh, S.H.: A Log-Structured Merge Tree-aware Message Authentication Scheme for Persistent Key-Value Stores. In: 20th USENIX Conference on File and Storage Technologies (FAST 22). pp. 363–380 (2022)
19. Kim, T., Park, J., Woo, J., Jeon, S., Huh, J.: ShieldStore: Shielded In-Memory Key-Value Storage with SGX. In: Proceedings of the 14th EuroSys Conference 2019. pp. 1–15 (2019)
20. Li, K., Tang, Y., Zhang, Q., Xu, J., Chen, J.: Authenticated Key-Value Stores with Hardware Enclaves. In: Proceedings of the 22nd International Middleware Conference: Industrial Track. pp. 1–8 (2021)
21. Li, M., Wilke, L., Wichelmann, J., Eisenbarth, T., Teodorescu, R., Zhang, Y.: A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 337–351 (2022)
22. Li, M., Zhang, Y., Lin, Z., Solihin, Y.: Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1257–1272 (2019)
23. Li, X., Li, F., Gao, M.: Flare: A Fast, Secure, and Memory-Efficient Distributed Analytics Framework. Proceedings of the VLDB Endowment **16**(6), 1439–1452 (2023)
24. LotusDB team: LotusDB. `https://lotusdblabs.github.io` (2022)
25. Lou, X., Chen, K., Xu, G., Qiu, H., Guo, S., Zhang, T.: Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. In: 2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 195–208 (2024)
26. Matetic, S., Ahmed, M., Kostiainen, K., Dhar, A., Sommer, D., Gervais, A., Juels, A., Capkun, S.: ROTE: Rollback Protection for Trusted Execution. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 1289–1306 (2017)
27. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative Instructions and Software Model for Isolated Execution. HASP@ ISCA **10**(1) (2013)
28. Meijer, C., Van Gastel, B.: Self-Encrypting Deception: Weaknesses in the Encryption of Solid State Drives. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 72–87 (2019)

29. Merkle, R.C.: A Digital Signature Based on a Conventional Encryption Function. In: Proceedings of the Conference on the Theory and Application of Cryptographic Techniques. pp. 369–378 (1987)
30. Meta Platforms: RocksDB. `https://rocksdb.org/` (2012)
31. MICRON: Data Security Features for SSDs. `https://www.micron.com/content/dam/micron/global/public/products/white-paper/self-encrypting-drives-white-paper.pdf` (2016)
32. Mikulas Patocka: dm-crypt Parallelization Patches. `https://lkml.iu.edu/hypermail/linux/kernel/1304.1/01458.html` (2013)
33. Nag, S.N.: Technical Analysis of PCIe to PCIe 6: A Next-Generation Interface Evolution. World Journal of Engineering and Technology **11**(3), 504–525 (2023)
34. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica **33**, 351–385 (1996)
35. PCI-SIG: Integrity and Data Encryption (IDE) ECN Deep Dive. `https://pcisig.com/sites/default/files/files/PCIe%20Security%20Webinar_Aug%202020_PDF.pdf` (2020)
36. Samsung: Samsung SED Security in Collaboration with Wave Systems. `https://download.semiconductor.samsung.com/resources/brochure/Samsung_SSD_Security_Encryption_Brochure.pdf` (2014)
37. Schneier, B., Kelsey, J.: Cryptographic Support for Secure Logs on Untrusted Machines. In: 7th USENIX Security Symposium (USENIX Security 98). vol. 98, pp. 53–62 (1998)
38. Schneier, B., Kelsey, J.: Secure Audit Logs to Support Computer Forensics. ACM Transactions on Information and System Security (TISSEC) **2**(2), 159–176 (1999)
39. Sev-Snp, A.: Strengthening VM Isolation with Integrity Protection and More. White Paper, January **53**, 1450–1465 (2020)
40. Shen, C., Fan, L.: Pldb: Protecting LSM-based Key-Value Store using Trusted Execution Environment. In: 2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). pp. 762–771 (2023)
41. Sinha, R., Christodorescu, M.: VeritasDB: High Throughput Key-Value Store with Integrity. Cryptology ePrint Archive (2018)
42. Sridhara, S., Bertschi, A., Schlüter, B., Kuhne, M., Aliberti, F., Shinde, S.: ACAI: Protecting Accelerator Execution with Arm Confidential Computing Architecture. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 3423–3440 (2024)
43. Stevens, C.: AT Attachment 8-ATA/ATAPI Command Set–4 (ACS-4). Working Draft, American National Standard, Revision **14** (2016)
44. Storage, T., Set, O.S.F., Tables, A.D.: TCG PUBLISHED (2012)
45. Sun, Y., Wang, S., Li, H., Li, F.: Building Enclave-Native Storage Engines for Practical Encrypted Databases. Proceedings of the VLDB Endowment **14**(6), 1019–1032 (2021)
46. Synopsys, Inc: Synopsys IDE Security IP Module for PCI Express 5.0. `https://www.synopsys.com/dw/ipdir.php?ds=security-pcie5-ide` (2021)
47. Turnbull, J.: Securing Files and File Systems. Hardening Linux pp. 187–231 (2005)
48. Weingart, S.H.: Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 302–317 (2000)
49. WiredTiger Inc.: WiredTiger. `https://github.com/wiredtiger` (2012)