# ShieldStore: Shielded In-memory Key-value Storage with SGX

Taehoon Kim
School of Computing, KAIST
thkim@calab.kaist.ac.kr

Joongun Park
School of Computing, KAIST
jupark@calab.kaist.ac.kr

Jaewook Woo
School of Computing, KAIST
jwwoo@calab.kaist.ac.kr

Seungheun Jeon
School of Computing, KAIST
shjeon@calab.kaist.ac.kr

Jaehyuk Huh
School of Computing, KAIST
jhhuh@kaist.ac.kr

## Abstract

The shielded computation of hardware-based trusted execution environments such as Intel Software Guard Extensions (SGX) can provide secure cloud computing on remote systems under untrusted privileged system software. However, hardware overheads for securing protected memory restrict its capacity to a modest size of several tens of megabytes, and more demands for protected memory beyond the limit cause costly demand paging. Although one of the widely used applications benefiting from the enhanced security of SGX, is the in-memory key-value store, its memory requirements are far larger than the protected memory limit. Furthermore, the main data structures commonly use fine-grained data items such as pointers and keys, which do not match well with the coarse-grained paging of the SGX memory extension technique. To overcome the memory restriction, this paper proposes a new in-memory key-value store designed for SGX with application-specific data security management. The proposed key-value store, called ShieldStore, maintains the main data structures in unprotected memory with each key-value pair individually encrypted and integrity-protected by its secure component running inside an enclave. Based on the enclave protection by SGX, ShieldStore provides secure data operations much more efficiently than the baseline SGX key-value store, achieving 8–11 times higher throughput with 1 thread, and 24–30 times higher throughput with 4 threads.

***CCS Concepts*** • **Security and privacy → Database and storage security**; **Software and application security**; *Trusted computing*;

*EuroSys '19, March 25–28, 2019, Dresden, Germany*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303951

***Keywords*** Trusted Execution, Intel SGX, Key-value Storage

**ACM Reference Format:**
Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302424.3303951

## 1 Introduction

The recent advent of hardware-based trusted execution environments such as Intel Software Guard Extensions (SGX) provides the isolated execution even from an untrusted operating system and hardware-based attacks [22, 32]. As the processor provides such shielded execution environments, SGX allows cloud users to run their applications securely on the remote cloud servers whose operating systems and hardware are exposed to potentially malicious remote attackers and staff.

Among numerous important cloud applications, one of the most widely used is the key-value store. Key-value stores such as memcached [3] and Redis [1], provide value storage indexed by keys. These simple data storage systems are popular for a variety of online service applications. Common key-value stores maintain data in main memory for fast access, and can also support durability by writing to persistent storage. Hardware shielded execution environments can potentially allow such an in-memory store to be protected from untrusted cloud systems and providers.

However, one of the critical limitations of SGX is the capacity of the protected memory region, called *enclave page cache (EPC).* Due to the hardware and performance overheads of securing the memory, EPC is limited to 128MB in current SGX implementations [5, 6, 35]. The EPC region allows efficient fine-grained accesses with cacheline-granularity encryption and integrity protection by a memory encryption engine [19]. If enclave memory usage is larger than the EPC limit, some pages are evicted from EPC, and remapped to the non-EPC memory region with page granularity encryption and integrity protection. Accessing data in the enclave memory pages not mapped in EPC involves a costly demand paging step which maps the page back to the secure region

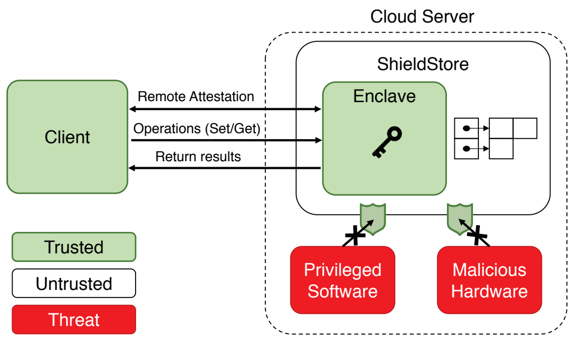

**Figure 1.** Overview of ShieldStore

along with the eviction of another victim page. Such demand paging causes a significant performance penalty [5, 15].

The EPC capacity limitation and coarse-grained paging can severely restrict the performance of secure key-value stores. A naive adaptation of current key-value stores to SGX can place the main data structure in enclave memory, without considering the implication of the EPC limit. However, with 10s to 100s of gigabytes of memory, most key-value data will not fit in EPC. Therefore, a majority of data accesses require costly demand paging to EPC.

To overcome the limit of SGX, this paper proposes a new secure key-value store design for shielded execution environments. Instead of relying on the page-oriented enclave memory extension provided by SGX, the proposed key-value store, called ShieldStore, provides fine-grained application-specific data protection, which maintains the majority of data structures in the non-enclave memory region. ShieldStore encrypts each key-value pair inside the enclave, and only the encrypted and integrity-protected data are placed in the non-enclave memory.

Fine-grained application-specific security management provides several benefits. First, it can minimize the unnecessary memory encryption overhead of demand paging. By understanding the data semantics, fine-grained data such as pointers in linked lists are not protected, as long as the security of key and value is provided. In addition, the encryption granularity always matches the size of a data item, reducing the cost of retrieving scattered small data items. Second, user space data management eliminates the costly EPC page faults which require enclave exits for demand paging. Without any enclave exit for memory extension, it improves the execution performance significantly. Finally, the semantic-aware co-optimization of index structures for fast access and protection of keys and values leads to an efficient key-value store running on an enclave.

This study is one of the first to apply SGX shielded execution to key-value stores, which effectively supports a large data capacity exceeding the EPC limit. Figure 1 shows the interaction of a remote cloud client and secure ShieldStore server running on SGX. *Trusted computing base (TCB)* in the cloud server is reduced to the enclave protected by the processor hardware.

We designed and implemented ShieldStore on a real system running Intel Processor i7-7700 supporting SGX. Our experimental results show that the proposed ShieldStore can improve throughput by 8–11 times with 1 thread, and 24–30 times with 4 threads, compared to a naive key-value store implementation with SGX. Its multi-threading design can provide scalable performance improvement compared to the single-threaded one, while the baseline key-value store on SGX does not scale beyond two cores.

The rest of this paper is organized as follows. Section 2 discusses the memory limitation of Intel SGX, and Section 3 presents the baseline key-value design. Section 4 presents the overall design of ShieldStore with fine-grained data protection, and Section 5 proposes optimizations. Section 6 presents the experimental results, and Section 7 discusses the current limitations of ShieldStore. Section 8 discusses the related work, and Section 9 concludes the paper.

## 2 Background

### 2.1 Memory Limitation in SGX

Intel SGX provides hardware-based isolated execution environments protected from malicious operating systems and physical attacks. Applications can create their own isolated execution environments called *enclaves*, and each enclave is backed by encrypted and integrity-protected memory, known as EPC. Enclave pages can be accessed only by instructions executing in the corresponding enclave. The data in EPC are in plaintext only in on-chip caches, and they are encrypted and integrity-protected when they are in the external memory. The memory pages in the enclave are accessible only from the owner enclave as the virtual-to-physical mapping is protected by the hardware address translation logic [15].

Memory access to EPC from the owner enclave is efficiently processed by hardware en/decryption logics at cacheline granularity. When a cacheline is brought from EPC to the processor, the cacheline is decrypted. In addition, the hardware logic calculates the keyed hash value of the cacheline, and verifies it against the stored hash value of the address. The integrity hash values are organized as a variant of Merkle Trees to allow sub-trees to be evicted from the on-chip storage securely [18, 19, 43]. However, in the current SGX, EPC size is limited to relatively small 128MB, considering the large main memory capacities of server systems. Due to the space and time overhead of storing and processing security meta-data at fine-grained cacheline granularity for EPC, the EPC capacity is unlikely to increase significantly. For example, creating a huge Merkle tree for tens or hundreds of gigabytes of main memory at cacheline granularity will increase the integrity verification latency intolerably [44].

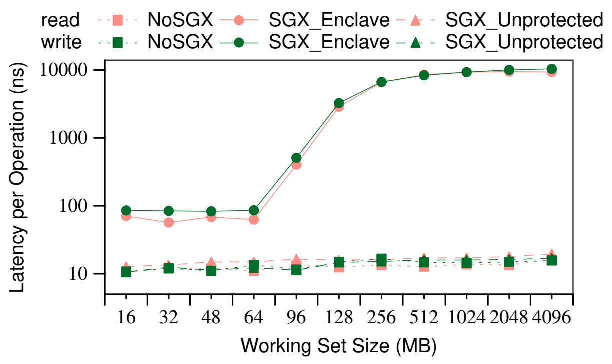

**Figure 2.** Memory access latencies w/ and w/o SGX

To mitigate the limitation, Linux support for SGX allows demand paging for EPC, extending secure memory pages for an enclave. If the memory pages used in an enclave exceeds the EPC limit, pages are evicted from the EPC region, and stored in an untrusted region. Although the evicted pages are still protected by encryption and integrity verification, the next access to an address in the evicted page will require fetching the page back to EPC by page swap, causing significant extra latencies. Furthermore, Microsoft Windows is yet to support such demand paging, limiting the enclave memory only to the EPC size. This limitation will affect the performance and feasibility of secure key-value stores on an enclave.

To quantify the performance degradation for accessing enclave memory pages when the enclave memory size exceeds the EPC limit, we evaluate memory access latencies with increasing working set sizes. Figure 2 presents the average memory read and write latencies for the enclave memory, with increasing the working set stored in the enclave. The microbenchmark issues a random memory read or write operation for each 4K page within the specified working set. For the measurement, the working set is much larger than the on-chip caches, and thus most of the accesses cause cache misses, getting the data from the memory.

The figure shows three curves for each of read and write operations. `NoSGX` presents the latencies with SGX disabled. `SGX_Enclave` presents the latencies when the entire data resides in the enclave. If the data size exceeds the EPC size, accesses require demand paging. The third curve, `SGX_Unprotected`, shows the latencies for accessing data in the unprotected memory region from an enclave. The access instructions are running inside the enclave, but those operations access the unprotected memory pages. The data in the unprotected region are not encrypted by SGX. As shown in Figure 2, if the working set size is less than 64MB, the read latencies of the enclave memory (`SGX_Enclave`) increase by 5.7 times compared to the execution without SGX. Note that the effective EPC is smaller than the 128MB reserved region

| Worker threads | Throughput (Kop/s) | |
|---|---|---|
| | memcached [3] | baseline |
| 1 | 313.5 | 311.6 |
| 4 | 876.6 | 845.8 |

**Table 1.** Throughput comparison for key-value stores w/o SGX: memcached vs. our baseline

(in practice around 90MB) due to security meta-data [19]. Moreover, if the working set size is greater than 64MB, the latency increases significantly, due to the paging overheads. With 4096MB working set, the latency of reading the enclave memory is increased by 578 times and write latency is increased by 685 times. These latency gaps are widening as the workload size gets bigger. On the other hand, reading or writing unprotected data from an enclave shows similar latencies to `NoSGX`.

### 2.2 Cost of Crossing Enclave Boundary

In addition to the memory limitation, since an enclave cannot issue a system call, any system services require exiting the enclave. The cost of crossing the enclave boundary is expensive. Recent studies show that the overhead is about 8000 cycles [35, 47]. Entering and exiting an enclave needs hardware operations such as flushing TLB entries [5]. Prior studies reduce the frequencies of enclave exits for system calls by running threads in untrusted execution for the interaction with the operating system [35, 47]. The untrusted threads for system calls communicate with the enclave thread by sharing memory. Furthermore, EPC fault handling due to the EPC limit also requires exiting the enclave, exacerbating the cost of paging. Eleos reduces the enclave exit events for EPC page faults, by supporting user space paging [35]. A recent update of Intel SGX SDK supports switchless calls for efficient processing of cross-enclave function calls [26, 45]. In this study, we employ HotCalls to reduce the overhead of crossing the enclave boundary for network evaluation (see Section 6.4), although Intel SGX SDK can provide a similar busy-wait switchless mechanism.

## 3 Baseline Key-value Store

### 3.1 Key-value Store on SGX

An in-memory key-value store stores key-value pairs in main memory, supporting durability through a logging mechanism to persistent storage. The most primitive operations for a key-value store are `set <key, value>`, and `get <key>`, which store and read a key-value pair respectively. In this paper, we employ a hash-based index structure. To resolve collisions in the hash-based index, we use *chaining* to create a linked list of key-value pairs for the same hash value. The hash-based index provides a simple and fast index structure, with small overheads for maintaining the index. However, a ranged

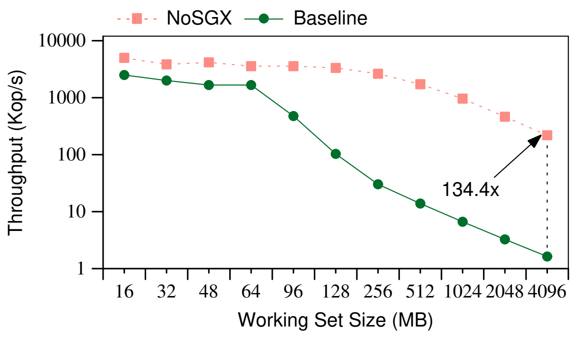

**Figure 3.** Baseline performance w/ and w/o SGX

search for a set of matching keys would be difficult. On the other hand, tree-based indices can provide ranged retrievals, although maintaining and searching the tree indices require higher complexities and overheads.

To validate the maturity of our baseline key-value store design, we evaluate the performance of our baseline key-value store and popular `memcached` [3]. We run the experiments with the networked setting as described in Section 6.1. Table 1 shows the performance of memcached and our baseline with 512B value size. As shown in the table, the performance of our baseline is similar to `memcached` with the default configuration of memcached for 1 and 4 threads.

To evaluate the performance impact of the memory extension overheads of SGX on key-value stores, we implemented a naive hash-based key-value store which stores the entire hash table in the enclave memory. The main hash table is stored in the enclave memory, far larger than the EPC limitation. As the EPC region can cover only a small portion of the total database size, a data access can cause page eviction and demand paging between an EPC page and non-EPC page. More details of experimental setup are explained in Section 6.1.

Figure 3 presents the performance of the key-value store with increasing database capacity. For comparison, it also shows the performance of the same key-value store without SGX. As shown in the figure, when the database size is smaller than 64MB, the performance is close to the insecure key-value store. Although the performance of the secure version is degraded by 60% compared to the insecure one, their difference is still relatively small. However, as the database working set increases to 96MB, the overall throughput decreases significantly. When the size is 4GB, it is 134 times slower than insecure runs. As shown in the result, a naive adoption of key-value store to SGX performs very poorly, due to limited enclave memory. To resolve the limitation of SGX, this paper proposes a fine-grained SGX-based data protection for securing the key-value store.

### 3.2 Server-side Computation for Key-value

This paper proposes a secure key-value store which protects the user data by *server-side* encryption with SGX. An alternative secure database is *client-side* encryption. For such a case, the remote key-value store just passively stores the encrypted key-value pair sent by the client. However, the *server-side* encryption model we investigate can provide richer semantics beyond simple `set` and `get` operations, supporting operations on the stored data. For example, `increment` or `append` are commonly used operations, which modify the value for a given key based on the prior value. Second, the client-side model allows only one fixed client to use the data. To allow multiple clients to decrypt the data, multiple clients need to be coordinated to exchange required keys and other security meta-data for integrity verification. In addition, the client-side model requires that data integrity for a huge data set is verified by the client, which must be able to validate the message authentication codes (MAC)s of the keys and values to avoid any types of replay attacks.

In the server-side encryption model we use, the client and server interact with the following steps. First, the client remote-attests the storage server system, verifying the SGX support of the processor, the code, and other critical memory state of an enclave. Second, the client and storage server running in the enclave generate session keys and exchange them. Intel SGX libraries provide essential implementations to establish this initial secure connection between the client and server in the enclave. Third, the client sends operations through the secure channel using the exchanged session key. The server deciphers and verifies the request, and accesses its key-value store. Clients do not directly access the ciphertexts on the server side, and they do not need to know secret keys the server uses. The server in the enclave will decrypt the retrieved data, encrypt them again with the session key used for the client, and send the response to the client.

### 3.3 Threat Model

The TCB of ShieldStore is the processor chip and the ShieldStore code running in an enclave, and thus it is resilient to malicious privileged attacks and certain physical attacks. With the reduced TCB, ShieldStore does not rely on the security of operating system managed by the cloud provider. Furthermore, it is also resilient to direct conventional physical attacks such as cold boot attacks, which attempt to retain the DRAM data by freezing the memory chip [20], or bus probing which reads the exposed memory channel between the processor and memory chips [36]. ShieldStore can protect cloud user data from such compromised operating systems and physical attacks by malicious staff. However, the weakness of the current SGX is the lack of protection for side-channel attacks [13, 24, 40]. In particular, several studies have demonstrated that SGX has vulnerabilities such as Foreshadow [10]. We will discuss the

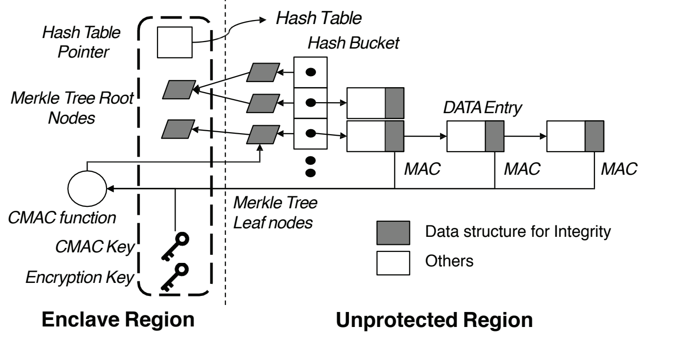

**Figure 4.** The overall data organization of ShieldStore

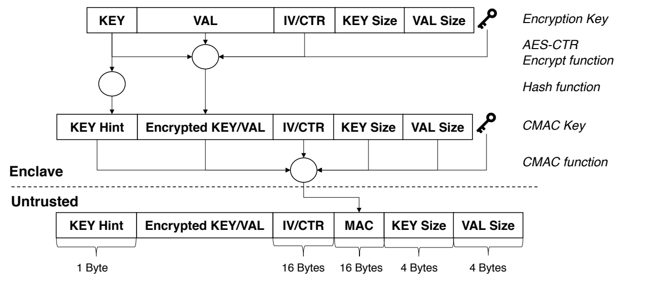

**Figure 5.** Data entry with encryption and integrity verification support

vulnerabilities of SGX that affect ShieldStore in Section 7. SGX does not provide direct protection for such side channels, and separate software or hardware techniques have been proposed to provide protection [13, 23, 34, 39–41, 49]. In addition, ShieldStore does not consider availability attacks. We assume that communication between clients and ShieldStore is protected through encryption.

## 4 Design

### 4.1 Overall Architecture

ShieldStore uses a hashed index structure with chaining. Unlike the naive key-value store with SGX discussed in Section 3.1, the main hash table storing key and data pairs is placed in the unprotected memory region. When an access to a key occurs, the corresponding key-value pair is searched from the unprotected memory region. As the main data structure is not protected by the SGX hardware, each data entry must be encrypted by ShieldStore in an enclave, and written to the main hash table for `set` operations. For a `get` operation, the encrypted data entry is retrieved from the main hash table in the unprotected memory region. The data entry is decrypted and its integrity is verified by comparing MAC.

Instead of relying on the paging to EPC, our design provides fine-grained key-value protection supported by code running in the enclave. Figure 4 presents the overall data structure of ShieldStore. Only main secret keys and integrity meta-data are stored in the EPC region. The main hash table data are stored in the unprotected memory region.

### 4.2 Fine-grained Key-value Encryption

To encrypt each key-value pair, ShieldStore uses an AES-CTR counter mode encryption with a per-key counter incremented for each encryption cycle. Figure 5 shows the data structure for a single data entry. The entry is composed of key-index, encrypted key-value, IV/counter, MAC, and key/value sizes. To hide both key and value for each entry, the key and value fields are encrypted. Although the key and value in the data entries are encrypted and not revealed directly, it is possible that the attacker can extract the distributions of keys per hash-bucket by investigating chained data entries with linked lists. By using a keyed-hash function for the hash index, the information leaks are minimized from the hashed key distribution.

**Key and value encryption:** If a new key is inserted, a data entry is allocated outside of EPC, but the enclave code fills the data entry with an encrypted value. The key and value are concatenated and encrypted by the counter mode encryption with the 128-bit global secret key and a per-key IV and counter. The critical performance problem of encrypting keys is that a search operation requires decrypting all the entries in the bucket to find the requested key. To achieve better performance, this paper proposes an optimization which attaches a 1B key-index as a hint to the data entry. The details are covered in Section 5.4.

**IV/counter management:** For the counter-mode encryption and CMAC hash computation, *sgx_aes_ctr_encrypt* and *sgx _rijndael128_cmac* in the Intel SGX SDK are used [24]. Since IV and counter are managed in a combined manner in the *sgx_aes_ctr_encrypt* function, we also store IV and counter as a combined field. Once the data entry is updated, IV/counter stored in the data entry is incremented. The IV/counter value is set to a randomly generated value by *sgx_read_rand* when a new entry is created, and stored in the data entry in plaintext.

**MAC Hashing:** The last field in the data item is a keyed hash value for integrity (MAC). ShieldStore attaches a 128-bit hash value created from encrypted key/value, key/value sizes, key-index, and IV/counter, to the data entry. This field is used to verify the integrity of the data entry for potential unauthorized modifications in unprotected memory pages.

**Operations:** For a `get` operation, the baseline ShieldStore fetches and decrypts all the entries which are in the same hash table bucket. By comparing a requested key and each decrypted key, the presence and position of the key are identified. Note that reading unprotected memory from an enclave is as fast as no-SGX reads. For a `set` operation, the corresponding data entry is searched in the same way as a `get` operation. If the same key exists, the key value field is updated with an encrypted new key and value. During the update, the IV/counter value of the key is incremented. Otherwise, the new entry is created and inserted to the head of the bucket. All of these calculations are done inside the enclave, and the updated data entry is written to the unprotected region.

### 4.3 Integrity Verification

To ensure the integrity of data, our integrity protection mechanism is derived from the Merkle tree design, supporting resiliency to replay attacks [18, 43]. However, applying the Merkle tree structure directly to all key-value pairs can be complicated and hard to manage, since the number of key-value pairs changes quickly. Furthermore, the height of the Merkle tree can be increased excessively for a large number of the key-value pairs.

To flatten the Merkle tree structure, instead of maintaining a single full Merkle tree, our design creates many in-enclave hashes, and the in-enclave MAC hashes always stay in the enclave memory region. This can be safely evicted by the paging mechanism of SGX. However, note that even if the pages containing in-enclave MAC hashes are evicted by SGX, their confidentiality and integrity are protected by SGX, although it can incur performance degradation. Each MAC hash value covers one or more buckets (bucket set), depending on the current numbers of MAC hashes and buckets. When the number of buckets is less than 1 million, the number of MAC hashes is equal to the number of buckets, and each MAC hash covers only one bucket. Each MAC hash maintains the keyed hash value of the combined MACs in the corresponding bucket set.

For a `get` operation, the MACs of all the key-value pairs in the bucket set covered by the MAC hash are read, and a hash value for the bucket set is computed. The computed hash value is compared to the correct MAC hash in the enclave. A set operation also requires reading the MACs of the other key-value pairs in the same bucket set to update the hash value. Although the bucket-based flattened hashes requires accessing all the entries in the same bucket set, it mitigates the overheads of an excessively tall Merkle tree.

The MAC hashes consume the majority of EPC memory used by ShieldStore, as most of the other major data structures are stored in unprotected memory. Therefore, deciding the number of MAC hashes and buckets is important to maximize the performance. If the number of buckets is small compared to the number of keys, the average length of chains is increased. However, as the number of buckets increases, the bucket set size covered by one hash value gets larger, increasing the overhead of MAC verification. The trade-offs will be presented in Section 6.2.

### 4.4 Persistency Support

For persistency, ShieldStore employs the current Intel SGX sealing mechanism [4, 25]. It uses a set of monotonic counters stored in non-volatile memory for resiliency against unexpected power failures. ShieldStore provides a snapshot mechanism similar to the one supported by Redis [1]. The snapshot mechanism of Redis forks the key-value store process to exploit the copy-on-write property between child and parent processes. The parent process handles incoming requests continuously while the child process stores the key-value entries on a file. The persistency support of ShieldStore is initiated every 60 seconds, which is equal to the default configuration of the snapshot mechanism in Redis.

**Algorithm 1** Optimized persistency support

```
1: procedure Persistent
2:     stop process of incoming requests
3:     seal meta-data on EPC
4:     fork the key-value store process
5:     if isParent? then
6:         write the sealed meta-data to storage
7:         store incoming requests in a temporary table
8:     else
9:         write encrypted key-value entries to storage
10:    if isChildDone? then
11:        update the main table with the temporary table
```

An advantage for ShieldStore snapshots is that the majority of the key-value data are already encrypted and integrity-protected in unprotected memory. Thanks to the design, the key-value data do not need to be re-encrypted to write a snapshot. During the snapshot process, the encrypted key-value entries in unprotected memory are directly written to storage. Only a small amount of security meta-data inside the enclave must be encrypted and sealed to storage.

When a periodic snapshot is initiated, ShieldStore seals secure meta-data in the enclave and forks a child process. The parent process stores the sealed meta-data on non-volatile storage as a file and waits for the child process which stores the encrypted key-value entries. In this way, consistency between sealed meta-data and key-value entries is maintained.

To parallelize processing requests and storing key-value entries, ShieldStore maintains another temporary table to keep incoming key-value entries, while the child process stores the original hash table. The temporary table is also stored in unprotected memory and the meta-data are updated concurrently with the temporary hash table. After the child process is done, the original hash table is updated with the temporary hash table. Algorithm 1 presents the design of the optimized snapshot mechanism.

ShieldStore uses the monotonic counter support from SGX to prevent rollback attacks. Alternatively, ROTE and LCM [8, 31] proposed an efficient protection mechanism against rollback attacks to address the vulnerability and inefficiency of the current sealing mechanism of Intel SGX.

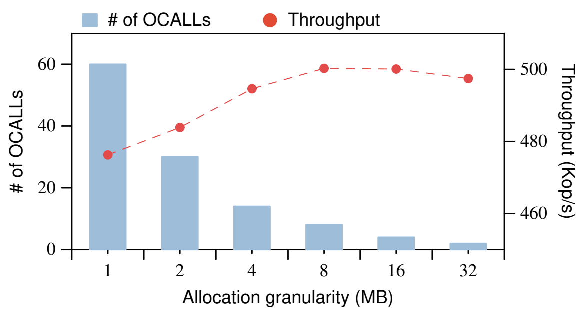

Figure 6. The number of OCALLs and throughput with the extra heap allocator

## 5 Optimizations

### 5.1 Extra Heap Allocator

ShieldStore constructs a key-value store in untrusted memory, and data entries are allocated by a heap allocator in the untrusted region. However, in the current SGX SDK, there are two heap allocators: one running inside an enclave allocates memory only in the enclave memory region, and the conventional one running outside an enclave allocates untrusted memory. However, each call to the outside heap allocator requires a costly exit from the enclave. To reduce the heap allocator overhead, we add a custom memory allocator which runs inside the enclave, but allocates unprotected memory. The customized allocator attempts to use free memory blocks from its own free memory pool. However, if the allocator runs out of free memory, it calls out of the enclave (OCALL) to execute a system call for mmap or sbrk obtaining a chunk of memory in the unprotected region. We modified a Intel SGX SDK *tcmalloc* library to create the custom allocator, and integrated it with the trusted ShieldStore code statically.

Figure 6 depicts the performance of ShieldStore with the RD50_Z workload of a small data set. This figure shows the number of OCALL operations and the throughput of ShieldStore with increasing granularity of memory allocation for each sbrk call. With the heap allocator, the occurrence of OCALL is drastically reduced as the chunk size of each sbrk is increased. In the rest of paper, we use the 16MB chunk size, which reduces the OCALL rates significantly with a small internal fragmentation. Although we use a fixed chunk size, the size of allocation can be tuned for various memory allocation behaviors of different scenarios, as ShieldStore can flexibly control the size of allocation.

### 5.2 MAC Bucketing

To check the integrity of data entries, ShieldStore collects all the MACs of the data entries in the bucket set for the corresponding MAC hash, recomputes the MAC hash value from the collected MACs, and compares it with the MAC hash stored in the enclave. Even if a requested key is found

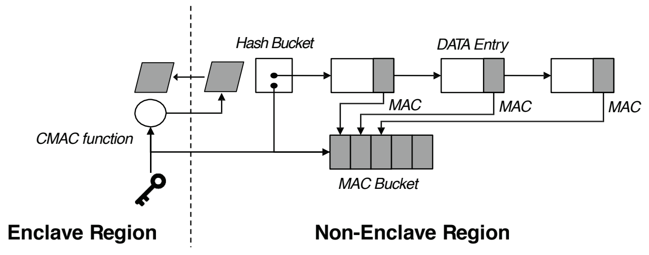

Figure 7. MAC bucket created for each hash bucket

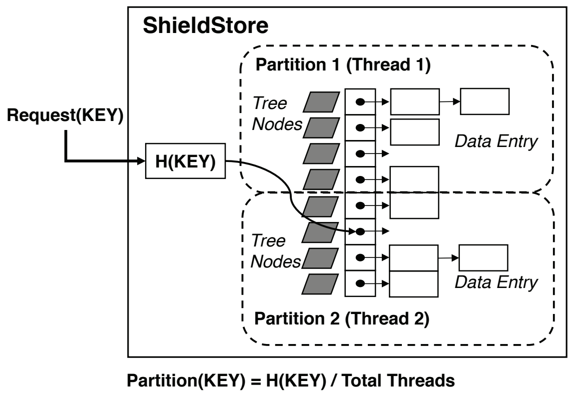

Figure 8. Multi-threading by hash key partitioning

early in the chain of data entries, the rest of data entries in the same bucket must be accessed to retrieve the MAC values.

To reduce the overhead of the MAC accesses through pointer chasing, each hash chain has its own MAC bucket. A MAC bucket contains only the MAC fields of data entries of the hash bucket. In our design, the MAC bucket contains 30 MACs, and if the length of chain exceeds the MAC bucket size, the MAC bucket itself must be chained through another MAC bucket. Figure 7 presents the data structure for MAC bucketing. Since MAC bucket exists for each hash bucket, reading all the MACs of data entries takes much less time by avoiding pointer traversal of data entries.

### 5.3 Multi-threading

As a performance optimization for the proposed key-value store, we adopt multi-threading. To exploit the parallelism without the overhead of frequent synchronization across multiple threads, we assign each thread to an exclusive partition of hash keys. To avoid synchronization, for a given key request, the serving thread is determined by the hash key, and thus multiple threads never update the same buckets. As each thread covers a disjoint part of the hash table, synchronization across threads for the table update is not necessary. Figure 8 illustrates task partitioning based on the hash key to avoid thread synchronization.

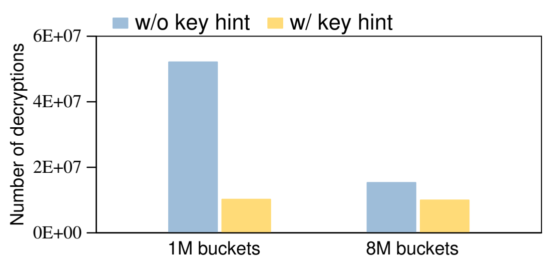

**Figure 9.** Average number of decryptions to find the matching entry w/ and w/o key hint on 1M and 8M buckets

One restriction of current SGX is that dynamic changes in the number of enclave threads are not supported. Therefore, ShieldStore does not provide dynamic threading which can increase or decrease the number of threads during runtime, although a prior study discussed a dynamic thread allocation support with SGX [42]. We leave supporting dynamic parallelism adjustment for future work.

### 5.4 Searching Encrypted Key

With the encryption for hiding a key field, the cost of searching an entry increases when the length of a hash chain increases, since a naive search requires decrypting all the entries in the same hash bucket. Note that each data entry has a different IV/counter, and thus it is not possible to know the encrypted value of a requested key before accessing the data entry. To reduce the cost of searching encrypted keys, ShieldStore uses a small 1 byte key hint in the data entry. The key hint is a hash of the plaintext key stored in each data entry. The hint is used to find candidates in the hash bucket for the requested key. The key hint trades the leakage of 1 byte of hashed key information in return for substantially better search performance.

Although using the key hint does not find the exact requested data entry, it can reduce the candidates for decryption. Figure 9 shows the number of decryptions to find the matching data entry w/ and w/o key hints for 1 and 8 million buckets, from the experiment with the small size data set described in Section 6.1. As shown in the figure, the number of decrypt operations is significantly decreased with a 1B key hint. With the small overhead of untrusted memory usage, the key hint can improve the performance of secure key-value store. For 8 million buckets, the reduction is decreased, as the chain length is reduced significantly with less collision, and the unnecessary decryptions are also reduced.

A potential problem with the key hint optimization is that an attacker can disrupt ShieldStore operations, although it does not infringe the confidentiality of data. When an attacker modifies key hints maliciously, ShieldStore may not be able to find matched entries. Such attacks can disrupt the data availability of ShieldStore operations. A remedy for such a key hint attack is to employ a two-step search. The key hints are used only for the first step search. If the first step search does not find potentially matched entries by the key hints, a full search by decrypting all the keys chained in the same bucket is conducted. In normal runs without the attacks, for get operations or set operations to existing keys, the first step search will find a matched entry, although a new key insertion will conduct both steps.

| Workload | Description (R:W ratio) | Distribution |
|---|---|---|
| RD50_U | Update heavy workload (50:50) | Uniform |
| RD95_U | Read mostly workload (95:5) | Uniform |
| RD100_U | Read only workload (100:0) | Uniform |
| RD50_Z | Update heavy workload (50:50) | Zipfian |
| RD95_Z | Read mostly workload (95:5) | Zipfian |
| RD100_Z | Read only workload (100:0) | Zipfian |
| RD95_L | Read latest workload (95:5) | Latest |
| RMW50_Z | Read modify write workload (50:50) | Zipfian |

**Table 2.** Workload configurations

| Data Set | Key Size(B) | Value Size(B) |
|---|---|---|
| Small | 16 | 16 |
| Medium | 16 | 128 |
| Large | 16 | 512 |

**Table 3.** Data size configurations

## 6 Results

### 6.1 Experimental Setup

We evaluate ShieldStore on Intel i7-7700 processors with 4 physical cores (8 logical threads). Due to the current lack of SGX support in server-class multi-socket systems, this paper can provide only modest 4-core evaluations. The processor has 32KB instruction and data caches, 256KB L2 cache, and 8MB shared L3 cache. It has two memory channels with four 16GB DIMM modules. The machine uses Ubuntu 16.04.5 LTS 64bit with linux kernel 4.13.0-36. We run the SGX driver, SDK, and platform software version 1.8 [24]. The Intel microcode and operating system update for mitigating Foreshadow/L1TF bugs was not applied in the evaluation [10].

In our first experiment, to focus on the performance of key-value structures with SGX, the requests are generated by the storage server without network components (standalone evaluation). This standalone setup enables a focused investigation of the CPU-memory performance impact of enclave execution with the proposed fine-grained data encryption. In Section 6.4, we evaluate the networked client-server scenario with a separate client machine connected through networks via a 10Gb NIC. The client machine simulates 256 concurrent users that generate requests for the key-value store.

**Workloads:** The workload scenarios used in this experiment are based on the two workload patterns used in a prior

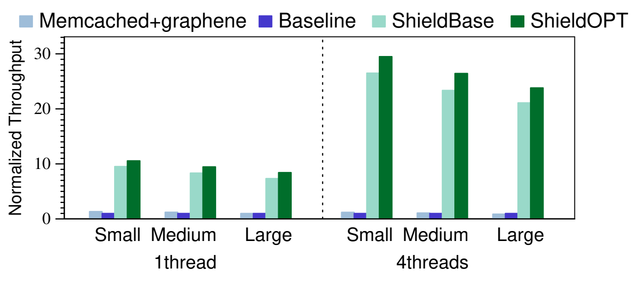

**Figure 10.** Overall performance with 1 and 4 threads

study [30]. The `uniform` workload generates keys with a uniform distribution, and the skewed workload generates keys with a zipf distribution of skewness 0.99, as used in YCSB [14]. To investigate the difference between `set` and `get` operations, we evaluate three configurations, 50:50 between `get` and `set` operations, 95:5 as get-intensive scenario, and 100:0 as get-only scenario. Table 2 summarizes the workload configurations for our experiments. The last two configurations (`RD95_L` and `RMW50_Z`) exhibit tight locality. We also evaluate three different data sizes, small, medium and large as described in Table 3. The small data set uses 16B key and 16B values, the medium data set uses 16B key and 128B values, and the large data set uses 16B key and 512B values. We initially set 10 million key-value pairs for each data set in the key-value store and evaluate the performance of workloads. The working set sizes of small, medium, and large data sets, are 320MB, 1.3GB, and 5.2GB respectively. All of the working sets in our experiments exceed the limitation of EPC size (128MB).

The baseline (`Baseline`) puts the entire hash table in enclave memory as discussed in Section 3.1. In addition to the baseline, to compare ShieldStore with a common key-value store, we also evaluate memcached with GrapheneSGX (`Memcached+graphene`) which allows an unmodified application to run in an enclave [11]. `ShieldBase` is the proposed key-value store without optimizations except for multi-threading from Section 5. `ShieldOpt` denotes the final version with all of the optimizations.

## 6.2 Standalone Evaluation

To isolate the effect of ShieldStore design from other factors, this section evaluates ShieldStore with the standalone setup where requests are generated within the same system.

**Overall Performance:** Figure 10 shows the average throughputs with three data sizes normalized to those of the baseline. The throughputs with 1 or 4 threads are normalized to those of the baseline with 1 or 4 threads respectively.

As shown in Figure 10, `ShieldBase` improves throughput from the baseline by 7–10 times with 1 thread configuration. As the number of threads increases to 4, the performance of `ShieldBase` is 21–26 times higher than the baseline, due to the scalability limitation of the baseline design. With several optimizations, `ShieldOpt` can further increase throughput over the baseline by 8–11 times and 24–30 times with 1 thread and 4 threads respectively. It provides an order of magnitude performance improvement through the design which considers the memory limitation of SGX. `Memcached+graphene` shows mixed performance results compared to `Baseline`, with a range of performance change over the baseline by $-1 \sim 34\%$ and $-12 \sim 20\%$ with 1 thread and 4 threads respectively. As our baseline uses a naive memory allocator instead of the customized slab memory allocator of memcached, `Memcached+graphene` provides a better memory allocation performance over `Baseline`.

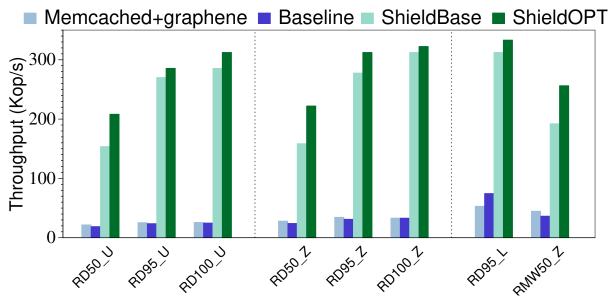

**Figure 11.** Throughput results of `Memcached+graphene`, `Baseline`, `ShieldBase`, and `ShieldOpt` with the large data set

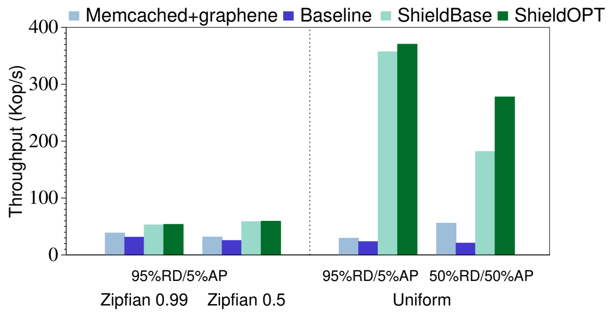

**Figure 12.** Performance of ShieldStore with append operation (RD:Read/AP:Append)

Figure 11 presents the throughput for each workload with the large data set. In `RD50_U` and `RD50_Z` with 50% sets, the throughput of `ShieldBase` achieves about 7.3 times improvements from the baseline. As the ratio of get operations increases in RD95 and RD100, the performance improvements by ShieldStore further increase by 11 times over the baseline.

**Append Operations:** One of the benefits of server-side encryption is to support data updates based on the prior value, such as `increment` or `append`. As an example of such updates, we also evaluate our key-value store with append operations. Figure 12 presents the throughput with append operations. The two workload mix scenarios are used, one with

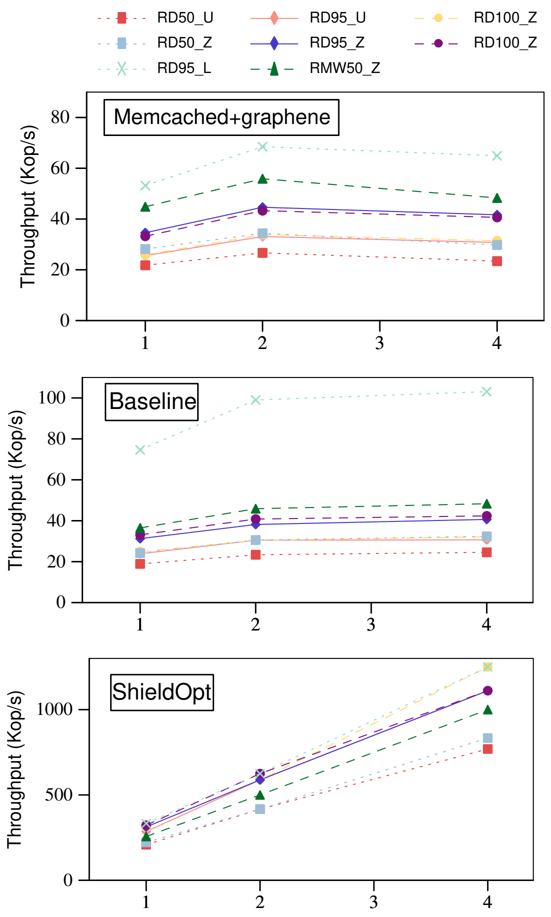

**Figure 13.** Performance scalability of `Memcached+graphene`, `Baseline` and `ShieldOpt` from 1 to 4 threads

95% reads and 5% appends, and the other with 50% reads and 50% appends. The same zipfian and uniform distributions are used for the evaluation. As shown in the figure, ShieldStore can improve throughput from the baseline for the append scenarios by 1.7–16 times. The performance improvements are lower in the zipfian distribution. In such skewed access cases, repeated append operations significantly increase the size of a small subset of data, and thus the decryption and encryption costs for the large frequently updated and accessed data dominate the performance, reducing the gap between the baseline and ShieldStore.

**Multi-core Scalability:** The next aspect of ShieldStore performance is its scalability with more physical cores. With hash key-based partitioning, ShieldStore mostly eliminates synchronization across threads. Figure 13 presents the throughput scalability for `Memcached+graphene`, `Baseline`, and ShieldStore. The `Memcached+graphene` and `Baseline` results show the scalability problem of the naive adoption of SGX for a key-value store. With frequent page faults, even

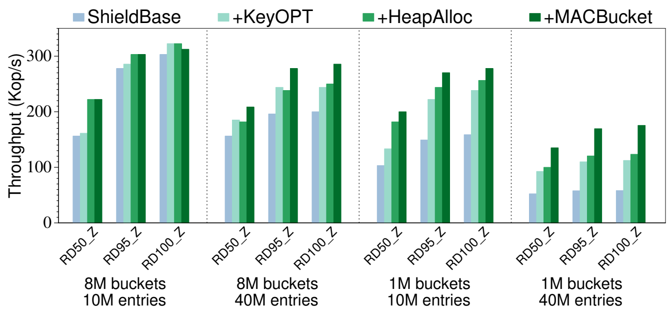

**Figure 14.** Effects of several optimizations with two bucket sizes (1M and 8M) and two key counts (10M and 40M entries)

if the number of threads increases, demand paging causes significant serialization of thread execution. Even if the number of threads increases from 2 to 4, no further throughput is gained. Especially, `Memcached+graphene` performance degrades with 4 threads compared to the performance with 2 threads since the background maintainer thread of memcached continually adjusts the hash table while holding locks.

Unlike the `Memcached+graphene` and `Baseline` results, ShieldStore provides scalable performance with increasing numbers of threads. As long as physical cores are available, the throughput is improved linearly from about 330 Kop/s with 1 thread to 1250 Kop/s with 4 threads. One of the critical issues for the thread implementation is the selection of memory allocator. We use *tcmalloc* instead of the default malloc in libc as dynamic memory allocation is frequently used for SGX-supporting libraries [24].

**Effect of Optimizations:** Figure 14 presents the effect of various optimization techniques with large data set size. It shows the operation throughputs with two bucket sizes (1M and 8M) and two key counts (10M and 40M entries). The four different configurations show different average chain lengths per bucket of 1.25, 5, 10, and 40. The figure shows the base ShieldStore throughput without any optimizations (`ShieldBase`), ShieldStore only with key-index (`+KeyOPT`), cumulatively, with heap allocation (`+HeapAlloc`), and the final ShieldStore with MAC bucketing (`+MACBucket`).

When the average chain length is small (1.25 with 8M buckets + 10M entries), the overall improvements by the optimizations is low. However, even in such cases, frequent set operations in RD50 benefit from the custom heap allocator. As the average chain length increases, the optimizations such as the key-index and MAC bucketing further improve performance. These optimizations will make the design more resilient under potential increases of collision.

**Trade-offs in MAC hashes:** As discussed in Section 4.3, the number of MAC hashes and buckets can have trade-offs. Figure 15 presents the throughput changes by four different numbers of MAC hashes. In the experiment, the number

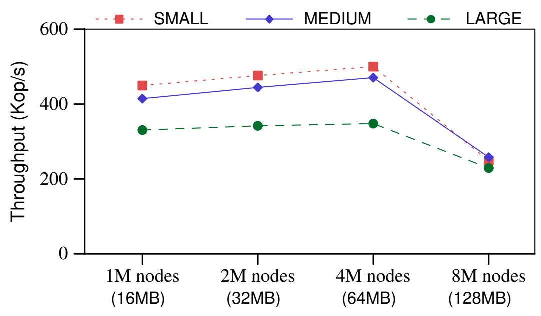

**Figure 15.** Performance of ShieldStore with various numbers of MAC hashes

of buckets is 8 million. As the number of MAC hashes increases from 1 to 4 million, the throughputs are increased by 11.25%, 13.53%, and 5.22% for the small, medium, and large data sets respectively. However, when the number of MAC hashes is 8M, the MAC hashes no longer fit in EPC, causing demand paging. It reduces throughput significantly. In our other experiments, the default number of MAC hashes is 4 million.

## 6.3 Comparison to Eleos

To improve the efficiency of memory extension for SGX, Eleos [35] proposed user space memory paging and exit-less RPC mechanisms inside an enclave to alleviate the limitation. Eleos can reduce the enclave exit events with the user space memory paging. It provides a generalized memory extension library to applications, although it requires certain modifications of memory related code in the applications. However, Eleos is still based on coarse-grained page encryption and integrity protection. This section compares our scheme with the baseline key-value store ported to Eleos.

Figure 16 presents the throughput comparison between ShieldStore and the baseline with Eleos. The data set size is fixed to 500MB, but the value sizes are varied from 16B to 4KB. We run 100% `get` operations in key-value store for testing Eleos. The figure shows that with the small 500MB data set, Eleos can perform efficiently for 1KB and 4KB values. Eleos uses 4KB as the default granularity, but it also supports 1KB sub-pages. However, when the value sizes decrease to 512B and 16B, the coarse-grained management of Eleos becomes ineffective. In those two fine-grained value sizes, ShieldStore performs 7 and 40 times better than Eleos.

Figure 17 presents the throughput comparison with various data set sizes from 32MB to 8GB. The experiments use 4KB value size, which is the best case scenario for Eleos. In ShieldStore, we employ a simple cache design to use the remaining memory of EPC efficiently at small working set sizes. The performance of ShieldStore with the cache shows that ShieldStore can leverage the enclave memory as much as possible, almost matching the performance of Eleos with

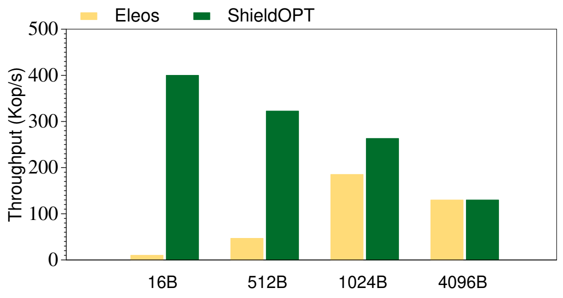

**Figure 16.** Performance comparison with Eleos on various value sizes (500MB working-set)

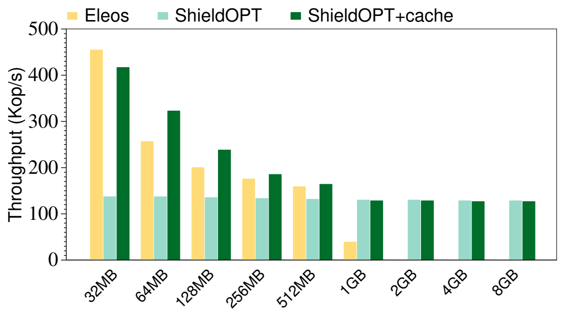

**Figure 17.** Performance comparison with Eleos on various working-set sizes (4KB value size)

up to 512MB size, where Eleos outperforms ShieldStore without the cache. ShieldStore without the cache provides consistent throughput even if the data set size is increased to 8GB in the result. However, even with the 4KB data size, Eleos does not support the data set larger than 2GB. Eleos uses the memsys5 slab memory allocator from SQLite [2] and pre-allocates a memory pool for using a backing store. Since the memory allocator can manage up to 2GB per pool, it needs several pools to increase the size of backing store over the 2GB, further increasing the overheads. The Eleos paper also reports memory sizes up to 1.5GB, and our results are consistent with that. In addition, as the data size increases from 200MB to 1GB, the performance decreases significantly for Eleos.

## 6.4 Networked Evaluation

This section evaluates the performance of ShieldStore with two machines for client and server processes. In the networked evaluation, a new extra overhead is added to ShieldStore: the cost of receiving requests and sending responses through the socket interfaces. In the front-end design of the secure key value storage server, a critical overhead is the enclave mode switch cost, as the network communication must be done with system calls. A recent study, HotCalls [47], proposed a new SGX interface to reduce the overhead between enclave and non-enclave communication. HotCalls uses a

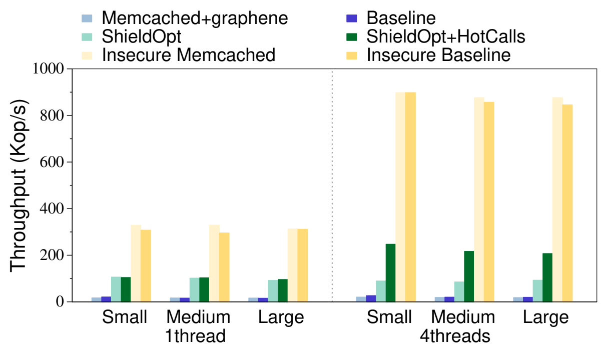

**Figure 18.** Networked evaluation with 1 and 4 threads

shared memory region to communicate with the trusted enclave code and eliminates the `EEXIT` instruction with its own security support. In the networked evaluation, we applied HotCalls to our design to reduce the overhead of network communication. Note that HotCalls is applied to the baseline too for fair comparison. The performance overheads of en/decryption of requests and responses between the client and server are included in the result.

Figure 18 shows the average throughput of ShieldStore for all the workload configurations with the networked client-server runs with one and four threads. The results also include the evaluation of memcached. First, the first two bars and the last two bars (memcached and baseline with/without security) compare the performance of the baseline key-value store implementation against memcached. The comparison shows that our baseline design almost matches the performance of memcached with and without SGX.

The two bars in the middle shows the performance of ShieldStore and ShieldStore with HotCalls. For the single thread evaluation, ShieldStore with HotCalls performs 4.9, 6.4 and 6.2 times better than `Baseline` with HotCalls in the networked runs, for the small, medium, and large data sets respectively. For the four-threads evaluation, ShieldStore with HotCalls performs 9.2, 10.7, and 10.6 times better than `Baseline` with HotCalls.

Comparing to insecure runs, ShieldStore still has a performance gap over `Insecure Baseline` due to the fact that ShieldStore requires mechanisms for protecting data and copying data back and forth from an enclave. ShieldStore with HotCalls performs 3.0 and 3.9 times slower than `Insecure Baseline` on average with 1 thread and 4 threads respectively. However, the secure `Baseline` is 17.7 and 39.8 times slower than `Insecure Baseline` with 1 thread and 4 threads respectively.

To support secure communication between the client and server, requests and responses are all encrypted. In addition to the encrypted network evaluation, we additionally evaluated ShieldStore without network security. The performance impact of such encryption is not high, with only up to 7% and 5% in ShieldStore with HotCalls in the single-thread and four-threads evaluations respectively. In `Baseline`, the performance overhead of en/decryption for network traffic is up to 9% and 27% with single-thread and four-threads evaluations respectively.

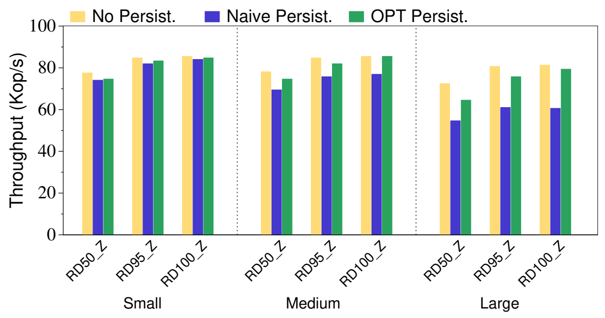

**Figure 19.** Performance of ShieldStore with persistent support

## 6.5 Persistency Evaluation

As discussed in Section 4.4, ShieldStore supports persistence via periodic snapshots. Figure 19 shows the performance of ShieldStore with persistency support. We evaluate performance in the networked environment with a subset of the workloads which have different ratios of reads and writes in the zipfian distribution. In the figure, the naive persistency bar shows the performance when the execution is blocked during snapshot generation.

With naive persistency support, the performance degrades with increasing data set size, since storing all the key-value entries incurs long stall times. It degrades the performance up to 25% with the large data set. However, the optimized version of persistence support reduces the performance degradation significantly since it parallelizes the processing of incoming requests and storing key-value entries. In particular, the optimized version with 100% read requests has a performance to that with no persistency support. With only read requests, since ShieldStore with optimized persistency does not write entries while writing a snapshot, it does not need to update the original hash table and keep the temporal hash table. With optimized persistency support, ShieldStore only degrades 2.1%, 2.6% and 6.5% of performance on average with small, medium, and large data set size.

# 7 Limitations

**Vulnerability of SGX:** Recently, several vulnerabilities of SGX have been announced such as Foreshadow attacks [10] and variants of Spectre attacks [12, 33]. These attacks extract secrets through speculative execution even if the data is stored in EPC. Protecting side-channel attacks is out of scope of this paper, but the current patch-based solution can potentially reduce the performance of any SGX applications. We expect that future hardware fixes will remedy the problem in a more efficient way.

**Limitation of hash-based key-value store with server-side encryption:** The hash-based index structure of ShieldStore does not support range query operations. For such range query operations, alternative designs using a balanced tree or skiplist can be adopted. However, incorporating the new index structure in ShieldStore requires substantial changes to the overall system such as the re-designing of integrity verification meta-data structure. Due to the lack of range query support, ShieldStore evaluated only the server-side update command (append). Modifying the index structure of ShieldStore for range operations is part of our future work.

**Weak persistency support:** ShieldStore supports persistency with periodic snapshots with the hardware monotonic counter as a defense for rollback attacks. However, ShieldStore loses any changes after the last snapshot, if a crash occurs. The alternative fine-grained design is to store a log entry for each operation. However, using the current hardware monotonic counters for such frequent sealing has performance issues as discussed in Section 4.4. To mitigate the problem, ShieldStore can be integrated with approaches which mitigate the limitation of the monotonic counters [8, 31].

**Untrusted meta-data:** ShieldStore stores the meta-data and pointers of a key-value table in untrusted memory. Although untrusted pointers may permit corruption and thus compromise the availability of ShieldStore, the integrity and confidentiality of the data can be protected. However, if an attacker changes a hash chain pointer to an enclave address, ShieldStore may overwrite critical data in the enclave region when writing new entries or data fields. To prevent this problem, before dereferencing pointers, ShieldStore must check whether the value of an untrusted pointer are outside the enclave's virtual address range. Since the virtual address range is contiguous, the checking mechanism is simple and would add minimum overhead to ShieldStore.

For its custom heap allocator, ShieldStore assumes the existence of a heap implementation that strictly partitions metadata (object sizes, free lists, etc.) from data (allocated objects), and permits keeping all metadata securely in enclave memory. Our current prototype uses a traditional heap implementation, and thus remains vulnerable to corruption of heap metadata in untrusted memory.

## 8 Related Work

**Trusted execution based on SGX:** With the introduction of Intel SGX [22, 32], hardware-based trusted execution environments have become available for general commercial applications. Haven [7], Graphene-SGX [11], and PANOPLY [42] used isolated execution to run unmodified applications in the enclave by adding a library OS in the enclave. As SGX does not allow any system calls inside the enclave, the library OS serves the system call APIs from unmodified application with its own memory management and file system. VC3 [38] and M2R [16] employs SGX on MapReduce computation to preserve integrity and confidentiality of distributed codes and data. Opaque employs SGX on distributed data analytics platform with obliviousness support [50]. Scone applies SGX support to secure linux containers [5]. It provides a secure container layer running in the enclave, protecting user applications from malicious privileged execution. Ryoan addresses a different aspect of remote cloud execution [21]. It uses SGX to protect the application, but in addition, by adding a sandbox layer on the enclave execution, it also protects user data processed by the untrusted 3rd party application in the enclave.

**Alleviate the limitation of SGX:** Several systems attempted to alleviate the limitation of SGX such as the limited trusted memory and the cost of crossing enclave boundary including HotCalls [47] and Eleos [35]. SGXBOUND reduces the memory-safe application's memory usage to fit in the enclave memory limitation [29]. SecureKeeper proposed a secure ZooKeeper with SGX storing the coordination data in untrusted region [9]. ShieldBox proposed a high performance secure middlebox for network functions with copying only the security sensitive packets into the enclave, while leaving other packets in untrusted memory [46]. HardIDX [17] proposed a secure B-Tree based index constructed in untrusted region to mitigate the memory limitation of enclave like ShieldStore.

**Secure Database:** EnclaveDB [37] and Pesos [28] proposed a secure database and object storage using SGX. However, EnclaveDB assumes that SGX supports large enclaves whose size is an order of several hundred GBs, and Pesos restricts the size of data structure to the size of EPC. Unlike the these systems, ShieldStore proposes a secure key-value store alleviating the memory limitation of Intel SGX.

A recent study conducted in parallel to ShieldStore proposed a secure key-value store for persistent storage [6]. The proposed key-value system, SPEICHER, hardens the Log-Structured Merge tree with trusted execution support from SGX, securing key-value data and provides rollback protection on untrusted storage. Compared to SPEICHER, which is primarily designed for the persist storage, ShieldStore is focused on a fast in-memory key-value store backed by coarse-grained periodic snapshots on persistent storage.

**Hardware-based trusted execution:** There have been extensive studies to improve the efficiency and security of hardware-based trusted execution. AEGIS proposed memory protection techniques with the counter-mode encryption and Merkle tree to provide both confidentiality and integrity for secure processors [43]. Recent studies investigate how to protect guest virtual machines. H-SVM proposed a low cost isolation mechanism across VMs protected from the malicious hypervisor [27]. Xia et al. proposed VM isolation resilient to physical attacks [48].

## 9 Conclusion

Although the trusted execution of SGX can improve the security of cloud-based data storage significantly, the limitation of securing memory severely restricts the performance of the naive adoption of SGX for a key-value store. This paper proposed a hash-based key-value store designed for SGX, with fine-grained data encryption and integrity protection. Our prototype shows that even with a very limited memory capacity in EPC, a large scale data store can be protected by placing encrypted data entries in untrusted memory. The evaluation on a real system shows 8–11 times improvements for 1 thread, and 24–30 times improvements for 4 threads, compared to the baseline. Using SGX, this study proves the feasibility of cloud-based key-value store resilient to privileged accesses from operating systems and hardware attacks. The source code is available at `https://github.com/cocoppang/ShieldStore`.

## Acknowledgments

The authors thank our shepherd, Andrew Baumann for his insightful comments and feedbacks on early revisions of this work. We also thank the anonymous reviewers for helpful comments. This work was supported by the National Research Foundation of Korea (NRF-2016R1A2B4013352) and by the Institute for Information & communications Technology Promotion (IITP-2017-0-00466). Both grants are funded by the Ministry of Science and ICT, Korea.

## References

[1] Accessed: 2017. Redis. http://www.redis.io/.

[2] Accessed: 2018. SQlite zero-malloc allocation system. https://www.sqlite.org/malloc.html.

[3] Accessed: 2019. Memcached: A high performance, distributed memory object caching system. http://memcached.org/.

[4] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 2013. 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)*, Vol. 13.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 689–703.

[6] Maurice Bailleu, Jörg Thalehim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *17th USENIX Conference on File and Storage Technologies (FAST)*.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI)*. 267–283.

[8] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. 2017. 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)*. 157–168. https://doi.org/10.1109/DSN.2017.45

[9] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Proceedings of the 17th International Middleware Conference (Middleware)*. Article 14, 13 pages. https://doi.org/10.1145/2988336.2988350

[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security)*. 991–1008.

[11] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC)*. 645–658.

[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *CoRR* abs/1802.09085 (2018). arXiv:1802.09085

[13] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (Asia CCS)*. 7–18. https://doi.org/10.1145/3052973.3053007

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. 143–154. https://doi.org/10.1145/1807128.1807152

[15] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained.. In *IACR Cryptology ePrint Archive*.

[16] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. 2015. M2R: Enabling Stronger Privacy in Mapreduce Computation. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC)*. 447–462. http://dl.acm.org/citation.cfm?id=2831143.2831172

[17] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2017. HardIDX: Practical and Secure Index with SGX. *arXiv preprint arXiv:1703.04583* (2017).

[18] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. 2003. Caches and hash trees for efficient memory integrity verification. In *The Ninth International Symposium on High-Performance Computer Architecture (HPCA)*. 295–306. https://doi.org/10.1109/HPCA.2003.1183547

[19] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *Cryptology ePrint Archive* (2016).

[20] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (2009), 91–98.

[21] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: a distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI)*. 533–549.

[22] Intel Corporation. 2014. Intel(R) Software Guard Extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[23] Intel Corporation. 2015. SGX Tutorial, ISCA 2015. http://sgxisca.weebly.com

[24] Intel Corporation. 2017. Intel(R) Software Guard Extensions SDK for Linux* OS. https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf.

[25] Intel Corporation. 2017. Trusted Time and Monotonic Counters with Intel(R) Software Guard Extensions Platform Services. https://software.intel.com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services.pdf.

[26] Intel Corporation. 2018. Intel(R) Software Guard Extensions SDK for Linux* OS. https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Reference_Linux_2.2_Open_Source.pdf.

[27] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. 2011. Architectural Support for Secure Virtualization under a Vulnerable Hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 272–283.

[28] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*. Article 25, 17 pages. https://doi.org/10.1145/3190508.3190518

[29] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*. 205–221. https://doi.org/10.1145/3064176.3064192

[30] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 429–444.

[31] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium (USENIX Security)*. 1289–1306.

[32] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. Article 10, 1 pages.

[33] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. 2018. SGXSpectre. https://github.com/lsds/spectre-attack-sgx.

[34] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, André Martin, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC)*. 227–240.

[35] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*. 238–253. https://doi.org/10.1145/3064176.3064219

[36] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security)*. 565–581.

[37] C. Priebe, K. Vaswani, and M. Costa. [n. d.]. EnclaveDB: A Secure Database using SGX. In *2018 IEEE Symposium on Security and Privacy (S&P)*, Vol. 00. 405–419. https://doi.org/10.1109/SP.2018.00025

[38] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*. 38–54. https://doi.org/10.1109/SP.2015.10

[39] Jaebaek Seo, Byoungyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*.

[40] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*.

[41] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (Asia CCS)*. 317–328. https://doi.org/10.1145/2897845.2897885

[42] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*.

[43] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*. 160–171. https://doi.org/10.1145/782814.782838

[44] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 665–678. https://doi.org/10.1145/3173162.3177155

[45] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. 2018. Switchless Calls Made Practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX)*. 22–27. https://doi.org/10.1145/3268935.3268942

[46] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2018. ShieldBox: Secure Middleboxes Using Shielded Execution. In *Proceedings of the Symposium on SDN Research (SOSR)*. Article 2, 14 pages. https://doi.org/10.1145/3185467.3185469

[47] Ofir Weisse, Valeria Bertacco, and Todd M. Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*.

[48] Yubin Xia, Yutao Liu, and Haibo Chen. 2013. Architecture Support for Guest-transparent VM Protection from Untrusted Hypervisor and Physical Attacks. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 246–257.

[49] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy (S&P)*. 640–656. https://doi.org/10.1109/SP.2015.45

[50] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 283–298.