Tee-based key-value stores: a survey

Aghiles Ait Messaoud, Sonia Ben Mokhtar, Anthony Simonet-Boulogne

Noname manuscript No.
(will be inserted by the editor)

TEE-based Key-Value Stores : a Survey

Aghiles Ait Messaoud · Sonia Ben Mokhtar · Anthony Simonet-Boulogne

the date of receipt and acceptance should be inserted later

Abstract Key-Value Stores (KVSs) are NoSQL databases that store data as key-value pairs and have gained popularity due to their simplicity, scalability, and fast retrieval capabilities. However, storing sensitive data in KVSs requires strong security properties to prevent data leakage and unauthorized tampering. While software (SW)-based encryption techniques are commonly used to maintain data confidentiality and integrity, they suffer from several drawbacks. They strongly assume trust in the hosting system stack and do not secure data during processing unless using performance-heavy techniques (*e.g.*, homomorphic encryption). Alternatively, Trusted Execution Environments (TEEs) provide a solution that enforces the confidentiality and integrity of code and data at the CPU level, allowing users to build trusted applications in an untrusted environment. They also secure data in use by providing an encapsulated processing environment called *enclave*. Nevertheless, TEEs come with their own set of drawbacks, including performance issues due to memory size limitations and CPU context switching. This paper examines the state of the art in TEE-based confidential KVSs and highlights common design strategies used in KVSs to leverage TEE security features while overcoming their inherent limitations. This work aims to provide a comprehensive understanding of the use of TEEs in KVSs and to identify research directions for future work.

Keywords Key-Value-Store · TEE · SGX · Privacy

A. Ait Messaoud
LIRIS, iExec Blockchain Tech, Lyon, France
E-mail: aghiles.ait-messaoud@insa-lyon.fr

S. Ben Mokhtar
LIRIS, CNRS, Lyon, France
E-mail: sonia.benmokhtar@insa-lyon.fr

A. Simonet-Boulogne
iExec Blockchain Tech, Lyon, France
E-mail: anthony.simonet-boulogne@iex.ec

1 Introduction

Key-value stores (KVSs) like Redis [29], DynamoDB [38], and RocksDB [82] are NoSQL databases that have gained popularity since the cloud shift post-2010. KVSs provide a simple, fast, and scalable storage solution for unstructured or semi-structured data by mapping each key to a unique value. They store various types of data, including sensitive and confidential information such as session details [38], private cryptographic keys [17],

and wallet data [61], necessitating strong confidentiality and integrity protections.

Traditional solutions for securing KVSs include software-based encrypted databases like [122,90,93,112], which keep data encrypted along with verification metadata. Clear-text data is accessible only to those with the decryption key, and data integrity is ensured through digests or MACs that are compared to known references. However, these methods merely shift the problem to securing cryptographic keys. If the cryptographic keys are kept by the data owner, only the latter can use the outsourced data, complicating cloud-based data sharing unless keys are shared with authorized clients, risking leakage. If the cryptographic keys are managed by the database server or a Key Management Service (KMS), trust must be placed in their system administrators and software stack to not misuse them. Building secure KVSs for confidential data under such conditions implies strong trust assumptions. Moreover, software-based encryption techniques protect data at rest and in transit but require expensive methods like homomorphic encryption [4] or verifiable computation [12,88,102,26] to protect data during processing.

An alternative to software-based encryption is using Trusted Execution Environments (TEEs)[97]. TEEs provide confidentiality and integrity at the hardware level by creating isolated secure environments, or enclaves, ensuring more robust isolation than virtualization-based techniques[60,67]. Therefore, TEEs shift the trust in computation from the server software stack to the enclave code and its underlying CPU chip. TEEs are utilized, among other things, in secure mobile platforms [13], secure payment systems [98], or even privacy-preserving federated learning [23,83,7,123], becoming central to securing sensitive data across various fields.

Despite several TEE implementations [34, 9,91], SGX [34] is the most widely deployed due to Intel's market share and investment in SGX development and tools. For instance, compared to ARM TrustZone [91], SGX offers built-in support for remote attestation [3], allowing verification of enclave authenticity and code integrity. However, SGX suffers from performance issues due to CPU context switching and its limited secure memory, which leads to in-enclave page faults and, thus, costly page swapping. To balance security and performance, SGX-based KVSs adopt novel designs and optimization techniques.

In this study, we examine TEE-based No-SQL KVSs and how they leverage the benefits of TEEs while addressing their limitations, especially those of SGX. All the existing TEE-based KVSs we could survey leverage Intel SGX. We believe this is due to the maturity and specific features of the Intel implementation (*e.g.*, resilience against physical attacks). We exclude general-purpose frameworks such as Scone [11], Gramine [111], Haven [21], Panoply [106], and SGX-LKL [94], which port existing codebases to enclaves with minimal effort and no optimization. Instead, we focus on KVSs with novel designs that address SGX limitations. We also target NoSQL KVSs and exclude SQL databases such as [126,95,20,125], as they serve different purposes, such as handling relational data (*e.g.*, for data analytics), and do not prioritize performance (including scalability). Moreover, these systems require a deeper literature review to address their specificities (parsers, optimizers, *etc.*).

The flow of this survey is depicted in Figure 1. Section 2 delves into the TEE concept, SGX implementation, limitations, and versions. Section 3 formalizes the adopted threat model for TEE-based KVSs. Section 4 synthesizes a modular architecture for the surveyed SGX-based KVSs and explains the roles and instantiations of each module. Section 5 classifies SGX-based KVSs based on relevant criteria and implementation strategies. Section 6 overviews the problem of side-channel attacks (SCAs) in SGX applications. Section 7 discusses open challenges related to SGX. Finally, Section 8 concludes the work.

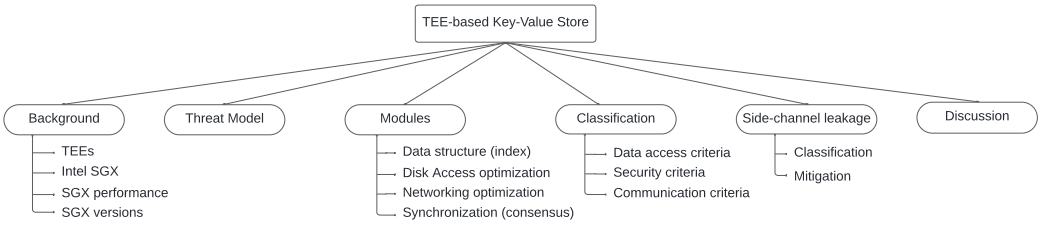


Fig. 1: Survey flow

2 Background

2.1 Trusted Execution Environments

In this section, we define TEEs and provide a comparison between major TEE implementations with respect to their security features.

2.1.1 Definition

TEEs are specialized environments implemented on commodity processors to protect applications (or portions of applications) in a secure environment isolated from other processes and the — potentially malicious — operating system (OS). They are turnkey solutions for securing applications, with early implementations appearing in the mid-2000s [120,65,28]. In 2012, GlobalPlatform and the Trusted Computing Group created standard specifications for TEEs. TEEs allow applications to use a secure region of memory (RAM) to store sensitive data and execute sensitive code without the risk of leakage or modification by unauthorized applications, including those with high privileges such as the OS and hypervisor. Today, TEEs are available from all major CPU vendors (e.g., ARM TrustZone [91], Intel SGX (Software Guard Extensions) [34], AMD SEV (Secure Encrypted Virtualization) [9]).

2.1.2 Implementations comparison

Ménétrey et al [79] compared the TEE implementations of major vendors according to several features. We extract relevant comparison criteria in Table 1. It is noteworthy that other academic, non-commercial frameworks for TEEs exist, such as Keystone [71], Sanctum [35], TIMBER-V [117], and LIRA-V [104], among others.

Features	**SGX**	**TrustZone**	**SEV-SNP**
Encryption	✓	✗	✓
Integrity	✓	✗	✓
Freshness	✓	✗	✓
Local attestation	✓	✗	✗
Remote attestation	✓	✓	✓
Open source	✓	✓	✗
Isolation level	Intra-address space	Secure World	Virtual Machine

Table 1: Comparison of major TEEs implementations

The considered criteria are: *(1)* Encryption: DRAM of TEE instances is encrypted to ensure that no unauthorized access or memory snooping of the enclave occurs; *(2)* Integrity: an active mechanism prevents DRAM of TEE instances from being tampered with; *(3)* Freshness: protects DRAM of TEE instances against replay and rollback attacks; *(4)* Local attestation: a TEE instance attests to another instance running on the same system; *(5)* Remote attestation: a TEE instance attests to its genuineness to remote parties; *(6)* Open-source: indicates whether the solution is

publicly available; *(7)* Isolation level: the level of granularity at which the TEE operates to provide isolation. (✓) means full support of the feature; (✓) means partial support; and (✗) means no support.

TrustZone. TrustZone (TZ) [91] does not provide built-in remote attestation to establish the trustworthiness of the code loaded into the enclave. Instead, it relies on academic work [6,74,103,124] to add remote attestation support. Nevertheless, after enclave loading, TZ does not offer reliable mechanisms to protect against integrity and freshness attacks. It also logically isolates its enclave from the OS environment but does not encrypt its main memory, leaving the TZ enclave vulnerable to memory snooping. Additionally, TZ requires a side-OS to support the secure environment (*i.e.*, Secure World).

SEV-SNP. SEV-SNP [9] protects entire virtual machines (VMs), including their OS. However, it does not guarantee integrity or freshness against physical attacks. While AMD has provided documentation and specifications for SEV to assist software developers in understanding and implementing support for the technology, the actual implementation and firmware of AMD SEV are not open source.

SGX. SGX [34] is the commercial TEE that supports the widest range of security features, particularly protection against physical attacks and encryption of secure memory. It also offers fine-grained security by enabling users to isolate specific parts of an application. SGX is further supported by extensive documentation and regular updates from Intel.

Based on the previous comparison, SGX may be favored by developers and researchers for building TEE-based applications, including KVSs.

2.2 Intel SGX

Intel SGX [34] is a set of extensions to the Intel x86 CPU architecture for implementing a TEE. It aims to provide confidentiality and integrity for user-level sensitive data and computations, even when the underlying execution platform (including privileged software) is potentially malicious. SGX allows developers to split applications into untrusted (non-sensitive) parts that do not require security properties and trusted (sensitive) parts that do; the trusted parts run in a protected memory region called an *enclave*.

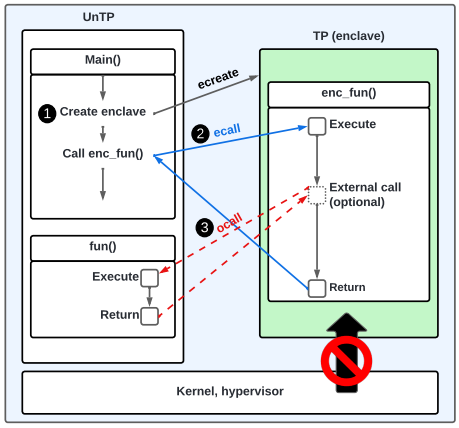


Fig. 2: High level SGX execution workflow

2.2.1 Execution workflow.

The execution workflow of SGX-based applications is described in Figure 2. An SGX application is subdivided into two parts: the Untrusted Part (UnTP), where the main program and routine functions are executed, and the Trusted Part (TP or enclave), where sensitive functions are executed. First, the `Main()` function of the UnTP runs until it reaches ❶ the `ecreate` instruction, which loads the enclave environment. The `Main()` function continues executing until it reaches ❷ an enclave call (*i.e.*, `ecall`) to an enclave function (`enc_fun()`). At this point, the CPU changes its context and executes `enc_fun()` until it returns. Optionally, during the execution of `enc_fun()`, the CPU can ❸ issue a call to a function in the UnTP environment (`ocall`), such as a syscall. When the enclave

function returns, the CPU restores its context and resumes the execution of `Main()`.

2.2.2 Memory layout.

The memory organization of SGX is shown in Figure 3. At boot time, a secure memory area called *PRM* (Processor Reserved Memory), whose size is determined by the BIOS, is allocated as a subset of DRAM (Dynamic Random Access Memory). The PRM itself consists of two main parts:

- *Enclave Page Cache*: The *EPC* is a reserved portion of system memory (DRAM) used to store the encrypted memory pages of the enclaves. These pages can only be decrypted at execution time by a key stored in the CPU, when they are in the physical processor core cache. When a trusted application exceeds the EPC size, the OS may evict some pages from the EPC to swap in other pages, following these steps: The CPU (1) reads the EPC page to be swapped out, (2) encrypts the contents of that page (second encryption), and (3) writes the encrypted page to regular, unprotected system memory. Since this process has inherent overhead, The more pages that are swapped out, the more performance drops.
- *Merkle Hash Tree*: The *MHT* [80] maintains the Message Authentication Code (MAC) tags of the EPC pages in the form of a tree, where the leaves represent the individual MAC of each page in the enclave, and the root represents the summary MAC of all enclave content.

To allow the creation of multiple enclaves, the EPC is divided into 4KB pages, each of which can be assigned to a different enclave instance. Since the system software responsible for allocating EPC pages (*e.g.*, OS kernel, hypervisor) is not fully trusted, SGX must verify the correctness of the allocation decisions. Therefore, SGX records the EPC allocation information in the *EPCM* (Enclave Page Cache Map), a lookup table inside the CPU, where each entry corresponds to an allocated page.

2.2.3 Remote Attestation.

Remote attestation [3] (RA) allows a remote client to verify that a particular application is running within an authentic (not forged) SGX enclave using an SGX-generated certificate. Remote attestation in SGX involves an architectural enclave called the Quoting Enclave (QE), provided by Intel, which generates a verifiable QUOTE to prove the authenticity of an enclave. Two RAs models are used :

- **EPID-based RA:** Enhanced Privacy ID (EPID)-based attestation [3] is a cryptographic protocol that provides anonymous attestation. This method allows an enclave to generate a QUOTE while preserving privacy through the use of digital signatures and zero-knowledge proofs [27]. The QUOTE can then be verified online by the Intel Attestation Service (IAS). EPID-based RA ensures the authenticity and integrity of an enclave without revealing the identity of its signer. It is primarily used in scenarios where anonymity is crucial, such as peer-to-peer networks, distributed ledger technologies (*e.g.*, blockchain), or any situation where the identity of the attesting device needs protection. Despite its advantages, support for EPID-based RA is deprecated, and Intel is now focusing on the newer DCAP scheme.
- ***DCAP*-based RA:** *DCAP* (Data Center Attestation Primitives)-based attestation [3,99] is a remote attestation scheme provided by Intel, specifically designed for data center environments. It enables the creation of on-premises attestation services, reducing the need for interactions with Intel's online services. This approach is particularly beneficial for cloud service providers (CSPs) who prefer to manage the attestation process in-house. DCAP allows CSPs to cache essential SGX certificates from Intel within their local infrastructure, thereby authenticating enclaves without relying on the IAS. Essentially, DCAP

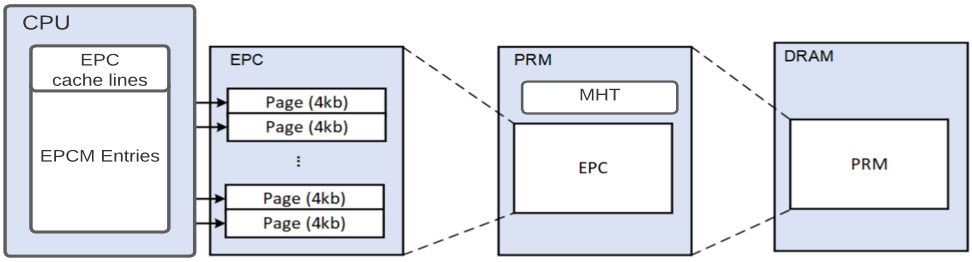


Fig. 3: SGX Memory layout

decentralizes the attestation protocol, facilitating faster enclave attestation.

2.2.4 Sealing.

Sealing [58] allows in-enclave data to be securely stored on persistent storage. The data is encrypted within the enclave using a secret sealing key derived from the *Root Sealing Key* (a cryptographic key embedded in the CPU fuse array during manufacturing) before being stored outside the enclave. This encryption provides confidentiality and integrity assurances for the sealed data. Intel SGX offers two binding policies for sealing keys, based on two different measurements:

- *MRENCLAVE* stands for *Enclave Measurement* and refers to a hash value of the enclave code and static data at the time of enclave creation.
- *MRSIGNER* stands for *Enclave Signer Measurement* and refers to a hash value of the identity of the entity that signed the enclave (typically an independent software vendor or a certificate authority).

If the sealing key is tied to a specific MRENCLAVE, any change that affects the enclave's measurement will result in a different sealing key, making the sealed data undecipherable. In other words, the MRENCLAVE-based sealing is used to share data between enclave instances with the same MRENCLAVE. On the other hand, if the sealing key is bound to an MRSIGNER, the data can be unsealed by any enclave signed by the same signer as the one that sealed the data.

2.3 SGX performance

Some critical limitations directly impact the performance of applications running in SGX enclaves and hinder their adoption. Two primary sources of performance degradation should be considered when designing secure applications with SGX:

Page faults. If the secure portion of the application exceeds the size of the EPC, the extra pages will be encrypted and swapped out of the EPC memory region. Accessing these pages will trigger costly decryption and verification checks when they are loaded back into the EPC. Therefore, it is advisable to wisely select the components (both code and data) to be secured inside the enclave to avoid expensive page swapping.

Environment context switch. Environment context switch refers to switching between the untrusted part of an application and the trusted part in an SGX enclave. A context switch is triggered by `ecalls` and `ocalls` and results in significant performance degradation, being 5.5 times more expensive than a user-mode context switch [19,118]. Excessive environment context switches can be caused by an application design that frequently alternates between trusted and untrusted parts, or by an excessive number of in-enclave system calls (syscalls).

2.4 SGX versions

Intel SGX CPUs are available in two flavours:
Client SGX. Client SGX systems provide comprehensive protection against software and hardware (physical) confidentiality and integrity attacks. To offer integrity protection, client SGX relies on the MHT at the expense of maximum secure memory, which does not exceed 128 MB. Any additional secure pages will trigger page swapping on Linux systems. An evolution of client SGX, namely SGX2, introduces EDMM (Enclave Dynamic Memory Management) feature. It allows an enclave to add more secure memory after it has already been loaded.
Scalable SGX. Built on top of SGX2, Scalable SGX was released in 2021, focusing on server-grade processors. Scalable SGX significantly increases the amount of secure memory available to the enclave to 512GB. It also introduces support for multi-socket CPUs. Furthermore, Intel switched from Memory Encryption Engine (MEE) to Total Memory Encryption – Multi-Key (TME-MK) [55] for faster memory encryption and execution. However, the increase in secure memory and performance compromises the security guarantees of scalable SGX CPUs, which sacrifice MHT and thus protection against hardware-based integrity attacks.

3 TEE-based KVSs threat model

We surveyed many TEE-based KVSs [108, 66,18,81,64,10,73,19,45,116] to understand the threat models they use. A common factor among these systems is their reliance on Client SGX as the TEE implementation. The likely motivation behind this choice, despite the fact that some KVSs appeared before scalable SGX, is that *"among the commercially available TEEs, Client SGX is the only one that provides integrity guarantees against a physical attacker"* [16], despite its limitation of having a small secure memory (128MB). Consequently, we detail the Client SGX threat model[34] that is considered by SGX-based KVS nodes.

SGX-based KVSs are designed to mitigate the following threats inside the enclave (EPC):

- **Malicious Software:** EPC is protected against malware that may attempt to access sensitive data (KV records) or tamper with application workflow running in EPC.
- **Insider Attacks:** EPC is protected against attacks from users or processes with legitimate access to the system (including OS, hypervisors) but who may try to abuse that access to steal sensitive information (KV records).
- **Physical Attacks:** SGX helps mitigate the risk of attacks on the system's hardware components by ensuring that sensitive data (KV records) inside EPC is encrypted by hardware key and isolated, even from privileged software running on the system.
- **Alteration of code and static data:** SGX allows remote parties to ensure that the expected code and static data are loaded in enclave through RA.

Lone SGX is not designed to handle SCAs, therefore the literature provides mitigation techniques on top of SGX. The surveyed TEE-based KVSs do not employ any mitigation techniques against SCAs, as these are notoriously costly and conflict with the objectives of the studied KVSs, which aim to maintain practical performance. However, some TEE-based SQL databases [125,41,36] do leverage SCA mitigation techniques that may also be applicable to KVSs. Section 6 provides an overview of SCAs.

4 TEE-based KVSs modules

TEE-based KVSs have a typical architecture that encompasses several modules. They are highlighted with green rectangles in Figure 4. Rectangles with dashed lines represent modules that are specific to distributed KVS.
KVS' Data Structure module. The choice

of the underlying data structure to store the KV pairs directly impacts the time complexity of the KVS for processing requests and its space complexity. The former point is independent of the use of Intel SGX, while the latter directly influences performance due to the limited size of secure memory. The goal of this module is to survey the data structures used in building SGX-based KVSs and describe how they were adapted to fit the limited secure memory constraints of SGX.

Disk Access Optimization module. This module encompasses a set of optimizations to efficiently access the storage device, through the enclave memory, without experiencing SGX-typical context switching overhead. It encompasses asynchronous storage syscall and the use of Storage Performance Development Kit (SPDK) [121].

Networking Optimization module. On the same way than Disk Access Optimization module, Networking Optimzation module encompasses a set of optimizations to efficiently access the network device (NIC) and process network packets. It encompasses asynchronous networking syscall and the use of Data Plane Development Kit (DPDK) [31].

Synchronization module. Multi-node (distributed) KVSs require communication to ensure data consistency and integrity across multiple nodes, using synchronization protocols that handle various system and networking errors (packet dropping, delays, alterations, *etc.*). This module presents the synchronization protocols adopted by distributed SGX-based KVSs to ensure consistency across their nodes in a byzantine environment [30] and how SGX benefits to them.

Security module. This module is a transversal component linked to all the previous modules to ensure they are correctly secured with the appropriate confidentiality and integrity guarantees, even outside the enclave. It encompasses a set of measures taken to extend the security of the enclave to sensitive data located outside the enclave, whether

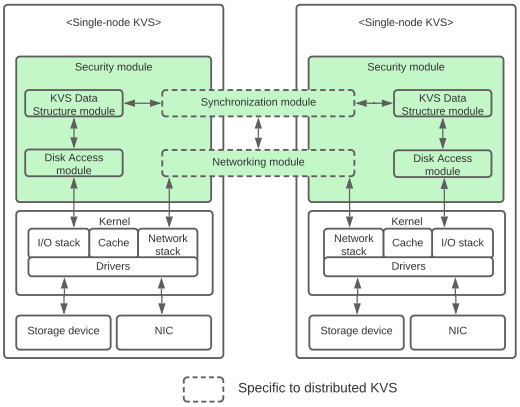


Fig. 4: TEE-based Key-value stores generic architecture

due to the lack of secure memory space or because of persistency (data exits the enclave boundaries when it is in storage disk).

The remaining of this section details the different building blocks and/or implementations of each module.

4.1 KVS' data structure module

The surveyed KVSs use the following data structures: Hash table, Skip list and Log-Structured Merge Tree (LSMT). We systematically explain the architecture of each of them, their space complexity, *get/put* operation workflow and their adaptation for SGX considering its small secure memory.

4.1.1 Hash table

A hash Table [78] is a data structure that implements an associative array abstract data type, a structure that can map keys to values. It uses a hash function to compute an index into an array of buckets (or slots), from which the desired value can be found. A hash function takes an input (or 'key') and returns an integer, which is typically the index where the KV pair is stored in the hash table. The goal is to distribute keys uniformly across the

buckets. Each bucket contains a list (or chain) of KV pairs that mapped to the same bucket.
Space complexity. Considering a hash table with M buckets and N KV pairs, the space complexity of such structures is *O(M+N)*.
Put operation. To add a KV pair to the hash table, (i) we compute the hash value of the key using the hash function, (2) we use the hash value to determine the index in the array, and (3) we insert the KV pair into the appropriate bucket. If the buckets are uniformly filled, this *put* operation results in an *O(1)* complexity. In the worst case, if all keys hash to the same index (poor hash function or high load factor), all KV pairs (N) end up chained in the same bucket. In this scenario, the insertion requires traversing a list of length N, resulting in an *O(N)* complexity.
Get operation. The *get* operation follows the same pattern as *put*, where we use the hash function on the key to locate the appropriate bucket and search for the element within it. Consequently, depending on the distribution of KV pairs across the buckets, the complexity can be *O(1)* in the best case and *O(N)* in the worst case.
Adaptation to SGX. ShieldStore [66] is an SGX-based KVS that uses a hash table to store its KV pairs. To save SGX memory, the entire data structure is kept outside the enclave. For confidentiality, each KV pair in each bucket is encrypted inside the enclave using an in-enclave encryption key before being stored outside. For integrity, a Merkle tree over the buckets is maintained outside the enclave, with its root secured inside the enclave.

4.1.2 Skip list

A skip list [96] is an in-memory probabilistic data structure that allows fast search, insertion, and deletion operations within an ordered sequence of elements (according to the key part of KV record). Skip lists are an alternative to balanced trees and provide a simpler and often more efficient implementation for certain types of data management tasks. It consists of multiple layers of linked lists. Skip list also introduces the concept of probabilistic balancing by using randomization to maintain an efficient structure (*i.e.*, nodes are promoted to higher levels based on a probabilistic criterion). Skip list leverages the following key principles:

- *Nodes:* Each node in a skip list contains a data (*i.e.*, KV pair) and a set of forward pointers. The number of forward pointers (levels) a node has is determined randomly when the node is inserted.
- *Levels:* The skip list has multiple levels, with the bottom level containing all elements. Higher levels contain fewer elements, creating shortcuts for faster traversal.

Space complexity. Every element in the skip list appears in the base level (level 0), which is a simple linked list. This contributes *O(N)* space, where N is the number of elements. Each element has a probability p of being promoted to the next level. Commonly, p is set to *0.5*, meaning each level has about half as many elements as the level below it. The expected number of levels in a skip list is *O(log(N))*. This is because, with probability p, a node appears in level 1; with probability p^2, it appears in level 2, and so on. Thus, the total number of nodes is modeled by a geometric series of ratio *p=0.5*, leading to a total number of nodes bounded by *2N*. Thus, the average space complexity of skip list, without considering the pointers, is *O(N)*.
Get operation. A skip list is composed on average of *O(log(N))* levels as we explained above. During a search, we start from the highest level and move forward until the next node's key exceeds the target key, then drop down a level. This process ensures that we are practically going through the levels, leading to an average time complexity of *O(log(N))*.
Put operation. (i) Inserting or updating an element requires finding the appropriate

position, which on average takes *O(log(N))* due to the search process explained above. (ii) Once the position is found, the new element is inserted into the multiple levels based on random level assignment. Since the expected number of levels for any node is *O(log(N))*, the insertion process also averages to *O(log(N))*.

Adaptation to SGX. Avocado [19] is a distributed SGX-based KVS where each nodes utilizes a skip list for indexing its underlying data structure. The skip list index, along with the keys, pointers to values, and their individual MACs, are stored within the enclave. The values are stored outside the enclave, in an encrypted form, using an in-enclave encryption key. Overall, this design attempts to ensure a balance between security (*i.e.*, by protecting keys, cryptographic keys and integrity checks inside the enclave) and performance (*i.e.*, by storing the bulkier values outside the enclave to avoid overloading EPC).

4.1.3 Log Structured Merge Tree.

LSMT [87] is a disk-based data structure designed for high-throughput write operations and efficient read operations in systems that handle large volumes of data and that require persistence. It is widely used in modern persistent databases and storage systems like RocksDB [82], and Speicher [18]. Behind LSMT name, two key concepts are leveraged: (1) Log-structured: Data is initially written to a log in an append-only manner to enforce sequential writes, which are faster than random writes, especially on disk storage; and (2) Merge tree: Data is organized in multiple levels or tiers, each level consisting of sorted runs of data. Periodically, data from smaller, more frequently accessed levels is merged into larger, less frequently accessed levels.

LSMTs leverage three key principles:

- *MemTable:* The LSMT begins with an in-memory table called the MemTable. All writes are first recorded here. Once the MemTable reaches a certain size, it is flushed to disk, creating a new sorted file called an SSTable (Sorted String Table).
- *SSTable:* SSTables are immutable and stored on disk. Each SSTable is organized in a set of blocks (tailored to the same block size as the disk), each block contains a set of KV pairs and the whole SSTable is sorted by key. When a MemTable is flushed, it becomes a new SSTable in Level 0.
- *Compaction:* To manage the growing number of SSTables and maintain read efficiency, compaction processes merge SSTables, invloving merging overlapping SSTables within the same level or between adjacent levels, deleting obsolete versions of keys and consolidating fragmented data. The process creates a new bigger SSTable at highest level. Compaction helps in maintaining the sorted order and removing stale data, thus optimizing read performance.

Space complexity. LSMT spans both main memory and disk storage. However, since enclave secure memory only encompasses main memory, we focus on the spatial complexity of the MemTable. The complexity depends on the data structure used to implement the MemTable. For classical architectures (*e.g.*, B-tree, skip list, etc.), the spatial complexity of the MemTable is *O(N)*, where N represents the maximum number of records a MemTable can store before flushing.

Put operation. The temporal complexity of *put* operation is amortized to *O(1)* as explained in the following: (i) Put operations are first appended in a write-ahead log (WAL) to ensure durability, which is an *O(1)* operation. (ii) Data is then written to the MemTable. Assuming that MemTable is a balanced data structure that record N KV pairs before flushing, it translates to an *O(log(N))* operation. But considering the amortized cost over multiple operations and the efficient management of writes, it's often treated as *O(1)*. (iii) When the MemTable is full, it is flushed to disk as an SSTable. Similarly to the previous operation, it is amortized to *O(1)*

considering that flushing spreads the cost over many writes. (iv) Finally, if compaction is triggered, its cost is also spread over many writes, leading to an amortized complexity of *O(1)* per write.

Get operation. (i) Read operations first check the MemTable, which takes *O(log(N))* if the MemTable is implemented as balanced tree, assuming that *N* is the number of entries of the MemTable. (ii) If the data is not found in the MemTable, the system searches through the SSTables, starting from the most recent and moving to older ones. To spead up this search process, bloom filters and other indexing structures are often used to quickly eliminating SSTables that do not contain the queried key. Assuming that the previous optimization techniques allow us to search only one SSTable per level and that *M,K* are respectively the number of entries in a single SSTable and the number of SSTable levels, the complexity of searching in SSTables is *O(K.log(M))*. The summarized complexity of *get* operation in LSMT is *O(log(N)+K.log(M))*.

Adaptation to SGX. Speicher [18] is a pioneering SGX-based KVS that leverages LSMT as its underlying data structure. For the MemTable implementation, it uses a skip list, similar to Avocado [19]. Consequently, it inherits the same partitioning design between enclave and non-enclave as described in *4.1.3: Adaptation to SGX*. Before persisting an SSTable, each block is encrypted using an in-enclave encryption key for confidentiality. Additionally, a Merkle tree is built over the SSTables of each level, where their root are kept in enclave, ensuring SSTables integrity and freshness.

4.2 Disk Access Optimization module

One of main source of overhead of SGX applications comes from context switching between the trusted and untrusted environments. In the context of SGX-based KVS, this switch is typically triggered by synchronous storage syscall triggered within an enclave, usually for persisting in-enclave data on disk. Two approaches are used to overcome this overhead:

4.2.1 Asynchronous storage syscall

Asynchronous storage syscall allows to execute storage operations asynchronously by leveraging a producer/consumer model. Syscall is issued from the enclave thread by placing a request into the producer queue (request queue). Then, a non-enclave thread, continuously probing the queue, processes these requests. When the syscall returns, the non-enclave thread places the result into the consumer (response) queue to be used by the enclave thread. Thus, enclave thread does not exit the enclave environment and consequently does not perform the expensive context switch. Shielded execution frameworks such as Scone [11] and Eleos [86] provide asynchronous storage interface.

4.2.2 Storage Performance Development Kit

SPDK [121,2] is a collection of tools and libraries designed to significantly improve the performance of storage applications by utilizing user-space, polled-mode drivers, and minimizing CPU overhead. SPDK achieves this by bypassing the traditional kernel storage stack, thus reducing latency and improving throughput. SPDK is built around several key principles:

– **User space drivers:** Virtual memory is typically divided into kernel space and user space, with storage drivers typically operating in kernel space. However, SPDK's drivers run in user space while still directly controlling storage hardware. To utilize SPDK, the OS must first release control (unbind) of the device. On Linux, this is achieved through *sysfs*. SPDK then binds the device to special dummy drivers like vfio, preventing the OS from reclaiming the

device. Subsequently, SPDK replaces the OS storage stack with its own implementations in C libraries, which include block device abstraction, block allocators, and filesystem components.

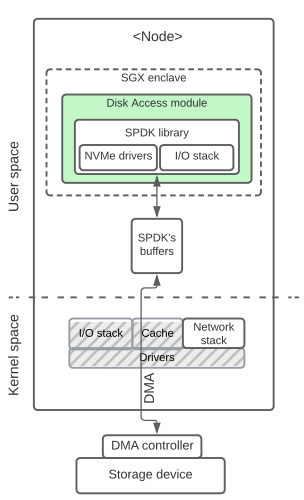
– **Polled mode drivers:** Unlike interrupt-driven drivers, SPDK employs polled-mode drivers. Specifically, when a user provides a callback function for an I/O operation, SPDK continuously polls the device to check for the completion of that I/O operation. When the I/O operation finishes, SPDK triggers the provided callback function. This polling mechanism ensures low-latency and high-performance storage operations by avoiding the overhead associated with interrupts and context switches.
– **Zero-copy data path:** In traditional I/O operations, data typically has to be copied multiple times. For instance, in order to write to storage device, data need to be copied from the user application buffers to kernel staging buffers, then from kernel staging buffers to storage device. These additional copies introduce latency and consume CPU resources. Zero-copy allows user applications to transfer data directly from the host memory to storage without using any staging buffer inside the kernel.
– **Modular Design:** SPDK is modular, enabling developers to use only the components necessary for their specific use case. This is particularly useful in SGX context where the size of secure memory is limited.

Coupling with DMA. DMA (Direct Memory Access) is a feature of computer systems that allows certain hardware subsystems (*e.g.*, storage devices) to transfer data to or from memory without involving the CPU, thus freeing up CPU resources for other tasks. The data transfer is supervised by the DMAC (DMA Controller), which only involves the CPU for initialization (*i.e.*, setting the source and destination addresses, amount of data to be transferred, and direction of transfer). DMA is often coupled with SPDK to manage data transfer from userspace between storage device and user application buffers without intermediate copying (*i.e.*, zero-copy) and without involving the CPU (*i.e.*, using DMAC).

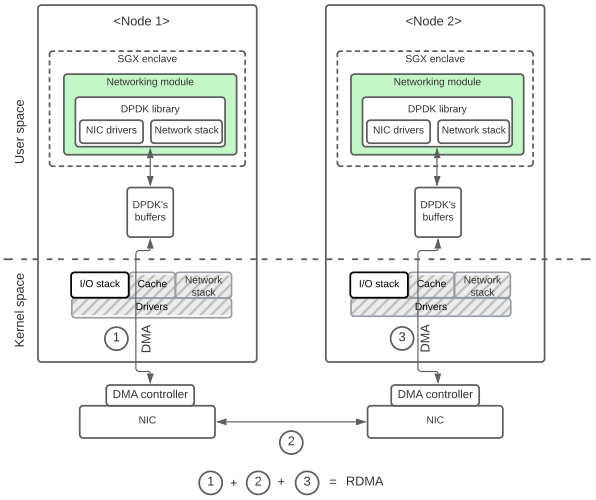
Adaptation to SGX. In the context of Intel SGX, SPDK's user space drivers can be installed within the enclave to enable enclave threads to issue I/O storage operations without exiting their enclave environment. Speicher [18] is an SGX-based KVSs that leverage in-enclave SPDK logic to improve I/O storage performance. However, to avoid expensive EPC paging, it hosts SPDK buffers outside the enclave. Figure 5(a) shows the combination between SPDK and DMA in the context of SGX applications and the bypassed kernel components. Performance measurements of Speicher's I/O throughput in GB/S (Gigabytes per second) under different block size conditions of SSTables reveal that the throughput of Speicher with adapted in-enclave SPDK is similar to that of RocksDB [82] with native -without enclave- SPDK (1.25GB/s for blocks of 16KB for instance), indicating that I/O-related overhead for enclaves are successfully mitigated.

4.3 Networking Optimization module

Another reason of context switch between the trusted and untrusted environment in SGX applications, alongside accessing to storage device, is accessing to network device (NIC, *i.e.*, Network Interface Card) to transfer packets through the network. Thus, Networking Optimization module is specific to distributed KVS where high-performance networking is essential for supporting low-latency communication between nodes. Traditional inter-node communication involves using synchronous networking syscall based on sockets to process network packets. Similar to persisting data, processing network packets force enclave thread to leave its security context, leading to a non-negligible overhead. TWo approaches are used to overcome this issue:



(a) Combining SPDK and DMA optimizes disk access within SGX enclaves by using the SPDK library to provide NVMe driver access in user space while the DMA controller handles data transfer. The combination eliminates dependence on the kernel storage driver, kernel storage stack, and kernel buffers. Nevertheless, SPDK's buffers should reside outside the enclave to be registered by DMA controller and to avoid costly EPC paging.



(b) Combining DPDK and RDMA optimizes data exchange within SGX enclaves by using the DPDK library to provide NIC driver access in user space while RDMA transfers data between two remote devices. This eliminates dependence on the kernel's NIC drivers, network stack, and cache. Nevertheless, DPDK's buffers should reside outside the enclave to be registered by DMA controller.

Fig. 5: Direct storage and direct transfer technologies in SGX-based KVS

4.3.1 Asynchronous transfer syscall

Like asynchronous storage syscall, it is an exitless syscall approach for inter-node data transfer based on sockets, where networking I/O operations do not block or prevent the execution of other code. Specifically, enclave thread delegates the syscall of packet transfer from/to NIC (Network Interface Card) to an external thread while the enclave thread continues to execute its code. The purpose is to avoid the context switch induced by in-enclave syscalls and optimize the CPU usage time.

4.3.2 Data Plane Development Kit

DPDK [31,1] is a set of libraries and drivers for fast packet processing, primarily used in network applications. It allows developers to build high-performance networking applications by bypassing the kernel network stack and accessing the hardware directly. Similar to SPDK, DPDK provides user-space drivers for various NICs and polling mode drivers. It also leverages Hugepages as described below.

Hugepages. In most OSs, memory is managed in small chunks of 4KB pages. Hugepages are much larger memory pages, typically 2 MB or even 1 GB in size, depending on the hardware and OS configuration. Using hugepages

in DPDK context allows for efficient memory allocation by providing large contiguous blocks of memory to handle high-speed packet processing while reducing fragmentation. Also, by reducing the number of pages (by increasing the size of individual pages), hugepages decrease the number of page table entries and TLB misses. This translates to lower latency and higher throughput in packet processing tasks.

Coupling with RDMA. If DMA is a feature for transferring data between local peripheral devices and main memory without involving the CPU, RDMA (Remote DMA) extends this capability to transfer data between the main memory of two remote systems over a network, without involving the CPUs of either system. RDMA is often coupled with DPDK to manage packet transfers between two remote userspace memories, while bypassing the CPU.

Adaptation to SGX. In the context of Intel SGX, DPDK's user space drivers can be installed within the enclave to enable enclave threads to issue I/O networking operations without exiting their enclave environment. Avocado [19] and Treaty [45] are both distributed SGX-based KVSs that use in-enclave DPDK logic to improve networking performance. However, to avoid costly EPC paging, Hugepages that host DPDK buffers are instantiated outside the enclave. Figure 5(b) shows the combination of DPDK and RDMA in the context of SGX applications and the bypassed kernel components. Performance measurements on Avocado show that its network stack based on DPDK is 1.66x faster in processing packets than sockets-Scone, an in-enclave and asynchronous secure socket by Scone [11] that does not leverage DPDK optimizations.

Table 2 summarizes the kernel components that are bypassed for each storage/networking optimization technology as well as their ability to bypass the CPU.

4.4 Synchronization module

The surveyed distributed SGX-based KVSs leverage three different synchronization protocols, implemented inside the enclave. This approach ensures reliable execution of synchronization code, allowing developers to avoid addressing the full spectrum of Byzantine failures (as handled by Byzantine Fault Tolerance - BFT - protocols [30]). Indeed, since enclaves are designed to prevent misbehavior, the synchronization module in SGX only needs to address crash and networking failures, which do not result from arbitrary cheating. Consequently, all employed synchronization algorithms in distributed SGX-based KVSs are CFT (Crash Fault Tolerant) rather than BFT (Byzantine Fault Tolerant). In a nutshell, using CFT logic in SGX enables systems to be resilient against the same malicious behaviors considered in BFT, while requiring only $2f+1$ nodes, as in CFT systems, instead of $3f+1$ (f being the number of faulty nodes).

Synchronization module is subdivided into two classes: Consensus protocols for non-transactional KVSs [19,116] (Raft [85] and ABD [14]) and a consensus protocol for transactional KVSs [45] (Two-phase commit [8]). We define each protocol and explain for Raft and ABD how the write and read operations are handled through the nodes. For 2PC, we explain how the transactions are distributively handled across the participants.

4.4.1 Raft

Raft [85] is a distributed consensus algorithm designed as an alternative to the Paxos [70] algorithm. It is used for managing a replicated log in a distributed system to ensure crash tolerance and consistency. Specifically, in a system of n replicated nodes, Raft can tolerate $f \leq \lceil \frac{n-1}{2} \rceil$ crashes. In Raft, the system is divided into three roles: leader, follower, and candidate. These roles are determined through an election process. The leader is

	Bypassed kernel components					CPU bypass
	Storage stack	Storage driver	Network stack	NIC driver	Cache (buffers)	
SPDK+DMA	✓	✓	✗	✗	✓	✓ (w DMA)
DPDK+RDMA	✗	✗	✓	✓	✓	✓ (w DMA)

Table 2: Bypassed kernel components w.r.t direct storage and direct transfer technologies

responsible for managing the replication of the log and handling client requests. Followers replicate the log and respond to client requests forwarded by the leader. Candidates are nodes attempting to become the leader by initiating leader election when no current leader is detected. Raft operates through a series of terms, each with its own unique leader. During each term, a leader is elected through an election process in which nodes exchange messages to determine the leader. Once a leader is elected, it maintains its position until it fails or another node with a higher term is elected.

Write operation. The core idea of Raft is log replication. The leader receives client requests, appends them to its log, and then replicates the log entries to followers. Once a majority of followers have confirmed receiving an entry, it's considered committed, and the leader can apply the entry to its state machine and respond to the client.

Read operation. In Raft, read operations are typically handled by the leader. When a client sends a read request to any node in the system, that node forwards the request to the current leader. The leader then responds to the read request by accessing its own state machine and returning the requested data to the client. This operation is performed by the leader to ensure that the client receives the most up-to-date data.

‘

4.4.2 ABD

The ABD consensus protocol [14] is another distributed consensus algorithm, specifically designed for systems where read operations are more frequent than write operations. It is a lightweight protocol that optimizes for read operations while ensuring consistency across the distributed system. Similar to Raft, in a system of n nodes, ABD can tolerate $f \leq \lceil \frac{n-1}{2} \rceil$ crashes. In the ABD protocol, each node maintains a data structure containing two timestamps: a read timestamp and a write timestamp. These timestamps are used to track the most recent read and write operations performed on a particular data item.

Write operation. When a node wants to perform a write operation on a data item, it initiates a write process by assigning a new timestamp to the data item. This timestamp is higher than any previous timestamps associated with the data item. The node then propagates the write request to other nodes in the system. Upon receiving a write request, a node compares the timestamp associated with the request with its own timestamps for the data item. If the request timestamp is greater, the node updates its own timestamp and acknowledges the write operation. If the request timestamp is lower or equal, the node ignores the request, ensuring that only the most recent write operation is accepted.

Read operation. In contrast to Raft, any node can handle read operation itself without forwarding the task to a unique leader. Still, to ensure that a read operation returns the most recent value and maintains consistency across the distributed system, the ABD protocol requires that a read operation contacts a quorum of nodes. This ensures that the read operation encounters the most recent write operation during its execution.

4.4.3 Two-phase commit

The Two-Phase Commit (2PC) [8] protocol is a distributed algorithm crucial for maintaining atomicity in distributed transactions, a key property related to both consistency and integrity in the context of transactions. It orchestrates a coordinated decision-making process among all participating nodes. 2PC leverages two sequential phases.

Voting phase. The coordinator node solicits votes from all participants regarding their readiness to commit the transaction (prepare request). Participants respond with either a "Yes" or a "No" based on their ability to commit. If all participants agree to commit, the coordinator proceeds to the second phase. Otherwise, if any participant votes "No" or fails to respond within a timeout period, the coordinator decides to abort the transaction.

Decision phase. The coordinator instructs all participants to commit the transaction (commit request). Upon receiving the commit request, each participant commits the transaction and acknowledges the coordinator. However, if any participant encounters a problem, it rolls back the transaction and informs the coordinator who broadcasts an abort message to all participants.

4.5 Security module

This module encompasses a set of measures taken by SGX-based KVSs to provide security properties to their data even outside the enclave. It includes the CIA triad (*i.e.*, "data confidentiality", "data integrity (with freshness)", "data availability") and the enclave-inherent security property: enclave attestation and secure channel establishment. We explain below how SGX-based KVSs use enclaves to enforce each of the properties above even when data are not kept inside the enclave.

4.5.1 Enclave attestation and secure channel establishment

Enclave attestation. Enclave attestation is used to verify the identity of a remote enclave (MRENCLAVE or MRSIGNER) by a client against an expected identity. This verification process assures the client that the remote enclave executes the expected code and can be relied upon for secure computation and data protection, establishing a foundation of trust in an untrusted environment. Enclave attestation is equivalent to certificate verification in PKI, where Intel acts as the main certification authority, with the additional verification of code integrity. As explained in section 2.2.3, there are two models for SGX RA: the DCAP model and the EPID model. All the surveyed SGX-based KVSs rely on the EPID model, even though it is deprecated. We believe this choice is due to its ease of use, as it only relies on IAS (Intel Attestation Service) without the need to deploy a dedicated infrastructure as required by the DCAP model to locally cache Intel certificates.

Secure channel establishment. After attesting an enclave, the client and SGX server establish a secure channel, with the server-side endpoint inside the enclave. This ensures that a client can push (or pull) its data to (or from) the server enclave without disclosing it to the surrounding non-enclave environment. Intel's SGXSSL [56] or previous academic work such as TaLoS [15] provide primitives to combine remote attestation (RA) and TLS to securely communicate with SGX servers.

4.5.2 Data confidentiality

Data confidentiality includes protecting sensitive data from disclosure. In SGX-based KVSs, inherent confidentiality offered by enclave is combined with confidentiality offered by software encryption, since the enclave alone may not be large enough to keep all

confidential data. Specifically, each SGX-based KVS develop its strategy to split its data structure across enclave and non-enclave environments, while using encryption key to cipher sensitive parts that are kept outside enclave, as explained in section 4.1. Since the encryption key does not leave the enclave, enclave confidentiality guarantees indirectly spans to non-enclave environments.

4.5.3 Data integrity and freshness

Data integrity is usually associated with freshness to ensure that data is accurate, not tampered with and in its last version. Regarding in-enclave data, their integrity and freshness is ensured by design with the SGX built-in MHT for EPC pages (see section 2.2.2). Regarding data that are kept outside the enclave, different methods are employed by SGX-based KVSs to guarantee their integrity and freshness:

Merkle Hash Tree. MHT is a cryptographic structure used to ensure the integrity of large datasets, including KVSs. It constructs a tree-like structure over a KVS where each leaf node represents the hash of a KV pair, and each non-leaf node represents the hash of its child nodes' concatenated hashes. The root of the tree, known as the Merkle root, uniquely summarizes the entire KVS. ShieldStore [66] is an SGX-based KVS that utilizes MHT to ensure the integrity of its KV pairs. Due to the typically large size of MHTs, only the Merkle root is kept inside the enclave, serving as a ground truth reference for the state of the KV pairs. Speicher [18] also leverages one MHT per SSTables of same level to ensure their integrity and freshness.

SGX counters. Built-in SGX counters are hardware-based monotonic counters provided by SGX technology to span the freshness protection provided by enclave to data located outside enclave boundaries, such as persisted data. These counters prevent replay and rollback attacks by offering a secure, tamper-resistant method for tracking the sequence of operations. By ensuring that each counter value associated with an operation or file is unique and incrementing, SGX counters help maintain data freshness. Furthermore, to detect data tampering, SGX counters are often paired with MAC computation, using an in-enclave MAC key. However, SGX counters are notoriously slow (60-250 ms) [77] because they are synchronous and wear out after only a few weeks [18]. More importantly, Intel stopped supporting SGX monotonic counters since version 2.8 of the SDK in January 2020, leading developers to propose their own methods for building trusted counters. ShieldStore [66] is an SGX-based KVS that leveraged SGX counters to ensure the integrity and freshness of its persisted (sealed) data.

Custom software counters. Since SGX monotonic counters are slow and have been deprecated by Intel, recent SGX-based KVSs have developed custom solutions for building trusted counters. For example, Speicher [18] and Treaty [45] systems designed an asynchronous trusted monotonic counter (AMC) to ensure the freshness and integrity of their log files. Each time data is modified, the counter is incremented asynchronously. This incrementation is only stabilized (persisted to a file) when the data itself is persisted. By separating the counter incrementation from its stabilization, AMC can maintain availability guarantees while optimizing performance, the incrementation being 7000x faster than the synchronous SGX counter.

Hash chain. Hash chain aims to create a cryptographic link between successive entries of a log file to detect integrity violation. For instance, Tweezer [64] leverages hash chain to maintain the integrity of its logs. Specifically, for each new log entry e_i, the system computes a MAC M_i by concatenating the encrypted data entry with the previous log entry ($M_i = MAC(M_{i-1}||Enc(e_i))$), with M_0 being a random nonce. Freshness is ensured by using a new MAC key for each log entry, invalidating a replayed log that would be linked to an older MAC key. Tiks [116], a

distributed KVS, also utilizes hash chains for its logs and additionally leverages the broadcast protocol offered by Raft to maintain the consistency and freshness of the logs across multiple nodes.

Operation ID. Operation ID (oid) is an incremental nonce attached to messages over the network to guarantee their freshness. For instance, Precursor [81] keeps track, within its enclave, of each oid provided by a client for its KV packets to ensure that the packets are not replayed by an attacker. Avocado [19] and Treaty [45], both distributed KVSs, also leverage authenticated packets with oid to track requests/responses and transactions, respectively.

4.5.4 Data availability

Data availability ensures that the data is available when needed, despite the presence of failures (*e.g.*, crashes), including enclave failures. To ensure this property, SGX-based KVSs rely on three methods :

Persistency. Persistency refers to the act of saving data from volatile memory (RAM) to non-volatile storage (disk), such as flushing in-memory MemTable to disk in form of SSTables, in the context of LSMT data structure. Persistency is natively used by persistent SGX-based KVSs such as Speicher [18], Tweezer [64] and Treaty [45]. However, to keep in-enclave data, such as software encryption keys, secure (in terms of confidentiality and integrity) even after enclave is shutdown, they rely on SGX sealing.

Logging. Logging involves recording every write operation in a KVS sequentially in log files before the operation is applied to the main data structure. The goal is to ensure immediate durability by enabling data replay in case a failure occurs before the data is pushed to the main data structure. Such mechanism is generally employed by log-based KVSs (*i.e.*, those that rely on LSMT [18, 64, 45]). Logging is usually combined with persistency to allow logs to be refreshed after persistency, by eliminating entries of persisted data from logs.

Replication. Replication is a mechanism usually leveraged by distributed KVSs that do not support data persistency (*e.g.*, Avocado [19]) to store copies of data on other machines. To ensure the consistency and integrity of replication, consensus mechanisms (see section 4.4) are employed across nodes. The main advantage of replication over persistency as a data availability strategy is that it eliminates the additional latency induced by flushing data to disk.

5 Classification of TEE-based KVSs

As shown in Table 3, we categorize the surveyed KVS into single-node KVS or multi-node (distributed) KVS. Single-node KVSs are databases that store KV pairs on a single machine while multi-node KVS store KV pairs on multiple machines through replication, sharding or both. We recall that all the surveyed SGX-based KVSs made the choice to use client SGX, which has a limited secure memory of 128MB, as their TEE implementation. We classify surveyed KVSs according to to three criteria: (1) Data access, *i.e.*, *the underlying data structure used while outlining KVSs that support range queries*; (2) security, *i.e.*, *how data are partitioned between enclave/non-enclave environments, what strategies are deployed to ensure data integrity with freshness outside the enclave, and what strategies are used to guarantee availability of data*; (3) communication, *i.e.*, *what are the employed I/O optimization techniques and the consensus algorithms used in distributed context.*

5.1 Data access criteria

We have previously enumerated the data structures commonly used in SGX-based KVSs (see section 4.1). In this section, we

Single-node KVSs	Multi-node KVSs
VeritasDB [108], ShieldStore [66], Speicher [18], Precursor [81], Tweezer [64], Concerto [10], eLSM [73]	Avocado [19], Treaty [45], Tiks [116]

Table 3: Surveyed TEE-based KVSs

classify existing SGX-based KVSs in Table 4 according to their underlying data structure. Frameworks that act as proxy atop existing KVS such as VeritasDB [108] and Concerto [10] can use any data structure to store KV pairs with no or minimal changes, so they are not classified. Table 4 also outlines KVSs that support range query operations in addition to the classical *get(K)/put(K, V)*. Below, we explain how each SGX-based KVS customized its data structure to exploit SGX features while addressing the secure memory size limitation.

Hash Table. ShieldStore [66] leverages a hash table to store its KV pairs. To avoid stressing the limited EPC, the hash table is kept outside the enclave with encrypted KV pairs. Only the bare minimum data is kept inside the enclave, such as cryptographic keys and MHT root. Precursor also leverages uses a hash table put prefer to keep the keys part in enclave and only the encrypted values part outside enclave.

Skip list. Avocado [19] is an in-memory distributed SGX-based KVS that employs skip-list on each individual node to store its KV pairs. The skip list spans across enclave and non-enclave environment. The skip list index, keys, pointers to values and their individual hashes are kept inside the enclave. The values are stored encrypted outside the enclave to alleviate the impact on the EPC.

LSMT. Speicher [18] is a pioneering persistent SGX-based KVS that leverages LSMT as its underlying data structure. It uses a skip list for its MemTable implementation, partitioned between the enclave and non-enclave environment in the same manner as Avocado [19]. Each SSTable is stored on disk (*i.e.*, outside the enclave) and authenticated using one per-block hash and a global hash over the per-block hashes. The entire SSTables are authenticated using a MHT whose root is kept inside the enclave. Tweezer [64] builds upon Speicher and proposes two design optimizations: using a unique MAC key per SSTable instead of building an MHT over SSTables to improve latency, and using finer-grained authenticated SSTables with one MAC per KV pair to save EPC memory during read operations instead of one hash per SSTable block. Treaty [45] also leverages Speicher as a building block to construct a distributed transactional SGX-based KVS by adding a distributed transaction layer (2PC) to support ACID transaction over multiple replicated nodes. Another noteworthy work is eLSM [73], which acts as an add-on (middleware) to existing non-SGX LSMT-based KVSs such as RocksDB [82] and LevelDB [44]. eLSM provides an enclave environment to store the logic of the KVS, index structure, and authentication data such as the MHT root of SSTables, with minimal changes to off-the-shelf LSMT-based KVSs.

Data structures		
Hash table	**LSMT**	**Skip list**
[66, 81]	[18, 45, 64, 73]	[19]
Range queries support		
[18, 64, 73]		

Table 4: Data Access

5.2 Security criteria

In this criterion, we classify SGX-based KVSs according to the security measures used to ensure the confidentiality, integrity (with freshness), and availability of data, by referring to Table 5. The table is subdivided into three parts:

Data separation. This part highlights the

generic partitioning of data between the enclave environment and the non-enclave environment, adopted by all SGX-based KVSs to overcome the SGX secure memory limitation. Data kept outside the enclave are generally large buffers that would consume significant secure memory. Being outside the enclave, they necessitate additional measures to secure them. Data kept inside the enclave are inherently secure regarding confidentiality and integrity (with freshness). This includes, among other elements, authentication data (hashes, MACs, etc.) and encryption keys to authenticate and encrypt/decrypt data located outside the enclave.

Integrity and freshness of outside-enclave data. This part focuses on the different methods used to provide integrity and freshness of data kept outside the enclave. We do not discuss ensuring the confidentiality of these data, as it is systematically achieved using encryption algorithms such as AES-GCM, with the encryption key kept inside the enclave to prevent unauthorized decryption.

Availability mechanisms. This part focuses on the different methods used by SGX-based KVSs to ensure that their KV pairs remain available even in the event of crash failures.

5.2.1 Data separation

Regardless of the studied SGX-based KVS, a generic framework is followed to partition data between enclave and non-enclave environments, as shown in Table 5. Specifically, we do not focus on what elements each specific KVS has, but rather provide a comprehensive overview showing the location of each element, if present, in an SGX-based KVS.

In-enclave data. Regarding in-enclave data, we find the different application's logic to ensure their execution reliability: (1) KVS operations logic (*e.g.*, *get/put*, compaction, *etc.*); (2) I/O optimization logic (SPDK, DPDK libraries) and; (3) Consensus logic employed in distributed KVSs. Index structures (including keys of KV pairs) are generally also stored inside the enclave to ensure reliable data fetching. Some exceptions, like VeritasDB [108] and Concerto [10], keep their index structure outside the enclave to save even more secure memory. If values are kept outside the enclave, their pointers, alongside their hashes, are maintained inside the enclave to detect any tampering with the values. The root of big authentication data (*e.g.*, MHT) is also kept inside the enclave as a reference for trust. Cryptographic keys used to encrypt, compute MACs for outside-enclave data, or secure sessions between clients and enclaves, are stored within the enclave to prevent unauthorized data decryption, alteration, or session spoofing. Finally, some frameworks, like VeritasDB [108], may also dedicate a secure memory portion to cache their latest accessed KV pairs to avoid re-authenticating them in case they are accessed again.

Outside-enclave data. Regarding outside-enclave data, we find the encrypted values (or the encrypted KV pairs) as their size may be prohibitive for the enclave. Large authentication data structures, such as the MHT (excluding the root), are also kept outside the enclave. Additionally, large data buffers, such as those used in SPDK and DPDK, are stored outside the enclave for the same reason. Finally, on-disk files, including persisted data and logs, are inherently located outside the enclave.

5.2.2 Integrity and freshness of outside-enclave data

Merkle tree. Methods that leverage Merkle Hash Trees (MHT) to safeguard the integrity and freshness of their data keep the MHT root inside the enclave. For instance, VeritasDB [108] and ShieldStore [66] use MHT to secure the KV pairs of their underlying data structures, while Speicher [18] and Treaty [45] use one MHT per SSTable level to secure their SSTables.

Data separation	
In-enclave data	**Outside-enclave data**
KVS operations logic, I/O optimization logic (SPDK,DPDK libraries), Consensus logic, Index structure (excluding [108, 10]), Pointers to external data (*e.g.*, in-memory values), Cryptographic keys (for encryption, MAC, session), Hashes of in-memory data (*e.g.*, values) or MHT root, Trusted counters, Cached KV pairs	Encrypted Vs (or KV pairs [108, 66, 10]), MHT, Log files and persisted data (*e.g.*, SSTables), SPDK/DPDK buffers

Integrity and freshness of outside-enclave data				
Merkle Tree	**Hash chain**	**SGX counters**	**Custom software counters**	**Operation id**
[108, 66, 18, 45]	[64, 116]	[108, 66]	[18, 45]	[81, 19, 45]

Availability mechanisms		
Persistency	**Logging**	**Replication**
[108, 66, 18, 45, 73, 64, 116]	[18, 45, 73, 64, 116]	[19, 116, 45]

Table 5: Security methods

Hash chain. Tweezer [64] and Tiks [116] both leverage hash chain to secure the integrity of their logs data. For ensuring their freshness, Tweezer utilizes a fresh in-enclave MAC key to authenticate each new log entry, while Tiks rely on its distributed consensus (Raft) for that (see section 4.4). Details are provided in *4.5.3: Hash chain*.

SGX counters. VeritasDB [108] and ShieldStore [66] both leverage SGX built-in counters to ensure the freshness of their sealed enclave data. However, as mentioned in *4.5.3: SGX counters*, Intel has deprecated the use of SGX counters. Consequently, these methods need to change their approach to ensure freshness.

Custom software counters. Speicher [18] and its distributed version, Treaty [45], leverage custom in-enclave asynchronous monotonic counters to ensure freshness of their log files. Details are given in *4.5.3: Custom software counters*.

Operation id. Precursor [81], Avocado [19] and Treaty [45] are SGX-based KVSs that explicitly mention tracking packet versions received through the network, inside enclave, to detect replayed packets.

5.2.3 Availability mechanisms

All LSMT-based KVSs [18, 45, 64, 73] leverage persistency and logging by design to write their SSTables and keep track of not-yet-persisted KV pairs. Other SGX-based KVSs [108, 66] implement their own mechanisms of persistency atop their storage. Finally, distributed KVSs [19, 45, 116] rely either on lone replication [19, 116] to keep their data available or on all the previous methods [45] to improve availability guarantees.

5.3 Communication criteria

Table 6 classifies SGX-based KVSs according to the mechanisms adopted to communicate with peripherals (storage disks, network cards) or other KVS nodes (to establish a consensus) from within the enclave.

6 Side-channel leakage

Despite SGX's security properties, it is far from perfect, as its popularity makes it a target for various attacks, especially side-channel attacks (SCAs) [110]. The latter

Disk access optimization	
Async syscall	**SPDK+DMA**
[45, 116]	[18]
Networking optimization	
Async syscall	**DPDK+RDMA**
[116]	[19, 45, 81]
Synchronization		
Raft	**ABD**	**2PC**
[116]	[19]	[45]

Table 6: Communication methods

represent a class of security threats that exploit unintended information leakage from a system's physical implementation rather than its theoretical cryptographic weaknesses. SCAs leverage various vectors of information leakage, among others: (1) page access or cache access pattern leakage by analyzing page accesses, page faults and cache hit/miss; (2) enclave interface invocation leakage by analyzing enclave function invocation delay; (3) instruction trace leakage by analyzing leaked instruction sizes; (4) volume leakage by analyzing the size (or volume) of the query result [62, 48, 50, 92, 89, 115]. By analyzing the previous leaked information, attackers can glean valuable information about the internal state of the system, including secret keys or other sensitive data.

6.1 Classification w.r.t compromised security objectives

A previous survey [43] has classified SCAs into three categories based on the SGX security objectives that may be compromised:
Confidentiality impairment: SCAs that falls under this category can obtain secrets from ISV enclave (*i.e.*, application enclave) instances, where KV pairs may be stored [107, 113, 119, 51, 63, 47, 84, 24, 100, 42, 54, 69, 52, 62, 48, 50, 92].
Attestation security impairment: SCAs that falls under this category can obtain secrets from architectural enclave instances, *i.e.*, Quoting enclaves (see section 2.2.3), compromise the integrity of the attestation mechanism and thus impersonate enclaves [37, 32, 72, 114].
Usability impairment: SCAs that falls under this category can interrupt the work of enclaves through Denial-of-Service (DoS) and render them unresponsive [59]. It is worth noting that some SCAs may fall under multiple categories, usually causing both confidentiality and attestation security impairments simultaneously. Examples include SGXPectre [32], an SGX-variant of the Spectre attack [68], and Foreshadow [114], a Meltdown-style attack [75] against SGX.

6.2 Mitigation

Existing countermeasures to SCAs can be categorized into three classes: hardware solutions, application solutions and system solutions.

6.2.1 Hardware solutions

Hardware solutions require modifications to processors and their microcode, which can only be implemented by manufacturers. For instance, architectural changes introduced by Cascade Lake CPUs enabled Intel to provide effective fixes against both SGXPectre and Foreshadow [109]. However, these solutions require a long time window before deployment in commercial CPUs, and only CPU manufacturers (*e.g.*, Intel, AMD, ARM) can propose these fixes.

6.2.2 Application solutions

Application solutions are proposed by literature to mitigate SCAs before CPU manufacturers propose hardware mitigation. They are usually hardware-agnostic and standalone security layers that may interface with existing applications. We leverage [43] and recent literature about volume leakage attacks to classify application solutions into five categories

according to their defense strategy and mitigated attacks.

Deterministic multiplexing. Deterministic multiplexing (DM) [107] protects SGX-enabled systems from page-fault driven attacks. It leverages page-fault obliviousness, ensuring input data does not affect the number of allocated pages, thus preventing sensitive information leakage through page faults. However, the software implementation of DM could lead to an average overhead of 705x, while a dedicated hardware implementation reduce overhead to 6.77%.

Anomaly detection. Similar to DM, these methods aim to mitigate page-fault driven SCAs. They are based on page anomaly detection as an indicator of on-going SCA. They rely on Intel's hardware transactional memory (TSX) [39] to handle page exceptions and interrupts, without the involvement of the OS. Specifically, T-SGX [105] redirects exceptions to a specific page, allowing a user-space fallback handler to manage errors, bypassing the underlying OS. T-SGX increased execution time by 40% and memory usage by 30%. Déjà vu [33] used TSX to create a reliable time measurement tool for anomaly detection, independent from the OS, effectively countering page-fault attacks despite increasing runtime. Regarding cache-access driven SCA, Cloak [49] used TSX to ensure that critical data stays in the CPU cache during transactions, proving effective with minimal overhead (1.2%) for low-memory tasks but significant overhead (248%) for memory-intensive ones.

Randomization. randomization is a memory protection technique that mitigates page-fault driven SCAs attacks by randomly loading data and code in the OS, making it difficult for attackers to locate targets. SGX-Shield [101], a code randomization technique, uses a secure in-enclave loader to secretly randomize memory layout with fine granularity by splitting target code into small randomization units and loading each at random addresses. DR.SGX [25], a data location randomization technique, breaks the link between an attacker's memory observations and a victim's data access patterns. It permutes data locations at a fine granularity using small-domain encryption and the CPU's AES-NI hardware acceleration. To prevent correlation attacks from repetitive access patterns, periodic re-randomization of enclave data is applied, although this results in performance overheads ranging from 4.36× to 11× depending on the re-randomization rate.

Volume leakage mitigation. In the context of KVSs, an attacker can monitor value size for simple *get* operations and/or the number of matched keys for *get-range* operations [48,50]. For multi-maps, where each key is associated with a vector of values, the adversary can also analyze the number of values associated with a specific key [89, 115]. Discussed methods to mitigate these attacks [50,92] include: (1) padding or injecting random noise (differential privacy) to query results or database to conceal the actual number of records or their size; (2) preventing query replay by uniquely identifying and recording each request; (3) preventing data injection by filtering suspicious data during *put* operations. However, these methods can introduce significant server overhead and impact user experience. Recent works aim to minimize server storage overhead caused by naive padding while still hiding volume information. For instance, dprfMM [89] uses a cuckoo hash-based method, and XorMM [115] employs an XOR filter to support volume hiding in multi-maps while minimizing server storage overhead. Similarly, Veil [53] randomly assigns each key to a set of equally sized buckets, which are padded if needed and may overlap for different keys.

ORAM. ORAM (Oblivious RAM) [46] is designed to protect from nearly all SCA vectors. It is a cryptographic construct that ensures secure access to memory regions on untrusted servers by accessing multiple locations per operation and re-shuffling/re-encrypting memory with a random seed.

OBFUSCURO [5] uses SGX for program obfuscation by enforcing code execution and data access via ORAM operations. It transforms the program layout to be ORAM-compatible and ensures fixed time intervals for program runs. Still, OBFUSCRO incurs significant run-time performance overhead (83× on average), although it is faster than most cryptographic obfuscation schemes.

6.2.3 System solutions

System solutions modify existing system software by integrating tailored SCA mitigation in their design. A notable example in databases is Opaque [125], an oblivious SGX-based distributed data analytics platform built on top of Spark SQL. At a basic level, Opaque leverages SGX to provide hardware-based data encryption, attestation, and enforce correct execution. In its oblivious mode, Opaque additionally provides oblivious execution, which protects against access pattern leakage in SQL queries. Opaque's method differs from Oblivious RAM (ORAM) as it takes into account specific SQL operations such as sorting. In its oblivious pad mode, Opaque pads the final output to prevent size leakage based SCAs. However, obliviousness is fundamentally costly, and Opaque's oblivious execution incurs an overhead of up to 46x compared to non-oblivious execution. ObliDB [41] further provides obliviousness while being 19x faster than Opaque, but still does not reach practical performance.

7 Discussion

Preference of Client SGX over Scalable SGX. All the TEE-based KVSs we surveyed use SGX implementations, specifically the client version of SGX, which offers the most extensive set of security features, including resilience against physical integrity attacks. However, this version has a small secure memory (128MB), requiring developers to adopt novel designs to avoid costly page swapping. Scalable SGX overcomes the limited secure memory size by providing an EPC size up to 512GB, reducing the effort needed to port applications to a secure environment. However, Intel achieved this by removing the built-in MHT, which protects against physical integrity attacks in the EPC. Without this protection, an attacker can mount replay attacks, replacing memory content with previous versions undetected. For instance, an attacker could revert a security hot-patch to reintroduce a vulnerability, allowing them to extract enclave secrets. To reconcile the large EPC size of Scalable SGX with the safety of Client SGX, Aublin et al. [16] propose a PoC that leverages both SGX versions: Scalable SGX for hosting and executing the main application code and Client SGX for hosting verification metadata of Scalable SGX pages. However, it is not clear whether Intel will keep supporting Client SGX alongside Scalable SGX. Otherwise, integrity protection of Scalable SGX against physcial attacks should be delegated to dedicated hardware such as on-chip hardware [71] or FPGA [40].

Continuous problem of SCAs. Intel prioritizes patching SCAs that threaten the attestation mechanism of SGX, as these attacks undermine the core security features of SGX. For example, Intel mitigated Foreshadow [114] and the most dangerous Spectre variants [32] through microcode updates [109]. However, for other SCAs that may leak end-user secrets without compromising SGX's built-in attestation secrets, Intel asserts that *"it is the enclave developer's responsibility to address side-channel attack concerns"* [57]. In the absence of Intel's mitigation, application-level solutions provide generic defenses against various SCAs. However, these solutions are expensive to implement, and thus none of the surveyed KVSs employed them, as they prioritize performance for real-world viability. Additionally, SCAs leverage unexpected channels, complicating future prevention

efforts. The Keccak Team notes, *"protection against side-channel attacks is never expected to be absolute: a determined attacker with massive resources will sooner or later break an implementation. The engineering challenge is to implement enough countermeasures to make the attack too expensive to be interesting"* [22]. Thus, future research will likely continue to focus on mitigating existing SCAs while controlling performance loss. An emerging approach may involve using specifically designed enclave oblivious memory [35,76] for building KVSs.

8 Conclusion

SGX-enabled CPUs provide security and trust assurances for various computing systems and services, including storage services like KVSs that handle sensitive data. However, integrating SGX introduces performance challenges due to limited secure memory and context switching, complicating the design of SGX-based KVSs. This survey aims to elucidate the complexities of SGX-based KVS design by identifying various modules that collectively form a fully functional KVS that attempt to address both security and performance considerations. We classify existing SGX-based KVSs according to multiple criteria reflecting these modules, providing an overview of the diverse strategies adopted in the literature to build secure and practical KVSs. Additionally, we highlight the persistent threat of SCAs to SGX systems, classify these attacks, and discuss available mitigations. Although Intel continuously patches the most dangerous SCAs affecting SGX attestation, mitigations for SCAs impacting end-user applications are left to enclave developers. Proposed mitigations effectively address access pattern leakage, a common SCA vector, but often result in impractical performance. Thus, none of the surveyed SGX-based KVSs implemented SCA mitigations, leaving them vulnerable to attacks that could leak users' KV pairs even with SGX protection. Future SGX-based KVSs will need to address this issue while preserving practical performance, suggesting the need for novel enclave architectures. Another concern is whether Intel will continue supporting Client SGX, the TEE implementation with the most comprehensive security features, or shift focus to Scalable SGX to meet the growing server demand for larger and faster secure memory, albeit at the expense of a broader security spectrum.

Acknowledgments

This work was supported by a French government grant managed by the Agence Nationale de la Recherche under the France 2030 program, reference "ANR-23-PECL-0007" as well as the ANR Labcom program, reference "ANR-21-LCV1-0012".

References

1. Dpdk: Data plane development kit (2010). URL `https://www.dpdk.org/`
2. Spdk: Storage performance development kit (2015). URL `https://spdk.io/`
3. Remote attestation (2022). URL `https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html`
4. Acar, A., et al.: A survey on homomorphic encryption schemes: Theory and implementation. ACM Comput. Surv. **51**, 1–35 (2018). DOI 10.1145/3214303
5. Ahmad, A., Joe, B., Xiao, Y., Zhang, Y., Shin, I., Lee, B.: Obfuscuro: A commodity obfuscation engine on intel sgx. In: Network and Distributed System Security Symposium 2019 (2019). DOI 10.14722/ndss.2019.23513
6. Ahn, J., et al.: Design and implementation of hardware-based remote attestation for a secure internet of things. Wireless personal communications **114**, 295–327 (2020). DOI 10.1007/s11277-020-07364-5
7. Ait Messaoud, A., et al.: Shielding federated learning systems against inference attacks with arm trustzone. In: Proceedings of the 23rd ACM/IFIP International Middleware Conference, p. 335–348 (2022). DOI 10.1145/3528535.3565255

8. Al-Houmaily, Y.J., et al.: Two-Phase Commit, pp. 1–7 (2016). DOI 10.1007/978-1-4899-7993-3_713-2
9. AMD: Secure encrypted virtualization api version 0.24: technical preview. Tech. rep., Advanced Micro Devices, Inc. (2020). URL https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/55766_SEV-KM_API_Specification.pdf
10. Arasu, A., et al.: Concerto: A high concurrency key-value store with integrity. In: Proceedings of the 2017 ACM International Conference on Management of Data, p. 251–266. New York, NY, USA (2017). DOI 10.1145/3035918.3064030
11. Arnautov, S., et al.: Scone: Secure linux containers with intel sgx. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, p. 689–703 (2016). URL url={https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov}
12. Arora, S., et al.: Proof verification and the hardness of approximation problems. J. ACM **45**, 501–555 (1998). DOI 10.1145/278298.278306
13. Atamli-Reineh, A., et al.: Analysis of trusted execution environment usage in samsung knox. In: Proceedings of the 1st Workshop on System Software for Trusted Execution, pp. 1–6 (2016). DOI 10.1145/3007788.3007795
14. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**, 124–142 (1995). DOI 10.1145/200836.200869
15. Aublin, P.L., Kelbert, F., O'keeffe, D., Muthukumaran, D., Priebe, C., Lind, J., Krahn, R., Fetzer, C., Eyers, D., Pietzuch, P.: Talos: Secure and transparent tls termination inside sgx enclaves. Imperial College London, Tech. Rep **5** (2017)
16. Aublin, P.L., et al.: Towards tees with large secure memory and integrity protection against hw attacks (2022)
17. (AWS), A.W.S.: Aws kms cryptographic details (2018). URL https://docs.aws.amazon.com/kms/latest/cryptographic-details/intro.html
18. Bailleu, M., et al.: Speicher: Securing lsm-based key-value stores using shielded execution. In: Proceedings of the 17th USENIX Conference on File and Storage Technologies, p. 173–190 (2019)
19. Bailleu, M., et al.: Avocado: A secure in-memory distributed storage system. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21), pp. 65–79 (2021)
20. Bajaj, S., et al.: Trusteddb: a trusted hardware based database with privacy and data confidentiality. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, p. 205–216 (2011). DOI 10.1145/1989323.1989346
21. Baumann, A., et al.: Shielding applications from an untrusted cloud with haven. ACM Trans. Comput. Syst. **33**, 1–26 (2015). DOI 10.1145/2799647
22. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Note on side-channel attacks and their countermeasures (2009). URL https://keccak.team/files/NoteSideChannelAttacks.pdf
23. Bonawitz, K.A., et al.: Towards federated learning at scale: System design. In: SysML 2019 (2019). DOI 10.48550/arXiv.1902.01046
24. Brasser, F., et al.: Software grand exposure: Sgx cache attacks are practical. In: Proceedings of the 11th USENIX Conference on Offensive Technologies (2017). URL https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser
25. Brasser, F., et al.: Dr.sgx: automated and adjustable side-channel protection for sgx using data location randomization. In: Proceedings of the 35th Annual Computer Security Applications Conference, p. 788–800 (2019). DOI 10.1145/3359789.3359809
26. Braun, B., et al.: Verifying computations with state. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, p. 341–357 (2013). DOI 10.1145/2517349.2522733
27. Brickell, E., et al.: Enhanced privacy id from bilinear pairing for hardware authentication and attestation. In: 2010 IEEE Second International Conference on Social Computing, pp. 768–775 (2010). DOI 10.1109/SocialCom.2010.118
28. Bussani, A., et al.: Trusted virtual domains: Secure foundations for business and it services. IBM Research Division, Tech. Rep (2005)
29. Carlson, J.: Redis in action. Simon and Schuster (2013)
30. Castro, M., et al.: Practical byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation, p. 173–186 (1999). URL https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance
31. Cerrato, I., et al.: Supporting fine-grained network functions through intel dpdk. In: 2014 Third European Workshop on Software Defined Networks, pp. 1–6 (2014). DOI 10.1109/EWSDN.2014.33
32. Chen, G., et al.: Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In: 2019 IEEE European Symposium on

Security and Privacy (EuroS&P), pp. 142–157 (2019). DOI 10.1109/EuroSP.2019.00020
33. Chen, S., Zhang, X., Reiter, M.K., Zhang, Y.: 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, p. 7–18 (2017). DOI 10.1145/3052973.3053007
34. Costan, V., et al.: Intel sgx explained. Cryptology ePrint Archive, Paper 2016/086 (2016). URL https://eprint.iacr.org/2016/086
35. Costan, V., et al.: Sanctum: minimal hardware extensions for strong software isolation. In: Proceedings of the 25th USENIX Conference on Security Symposium, p. 857–874 (2016)
36. Cui, S., et al.: Preserving access pattern privacy in sgx-assisted encrypted search. In: 2018 27th International Conference on Computer Communication and Networks (ICCCN), pp. 1–9 (2018). DOI 10.1109/ICCCN.2018.8487338
37. Dall, F., De Micheli, G., Eisenbarth, T., Genkin, D., Heninger, N., Moghimi, A., Yarom, Y.: Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2018(2) pp. 171–191 (2018). DOI 10.13154/tches.v2018.i2.171-191
38. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. **41**, 205–220 (2007). DOI 10.1145/1323293.1294281
39. Dementiev, R.: Exploring intel transactional synchronization extensions with intel software development emulator. URL https://www.intel.com/content/www/us/en/developer/articles/community/exploring-tsx-with-software-development-emulator.html
40. Erhu, F., et al.: Scalable memory protection in the PENGLAI enclave. In: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), pp. 275–294 (2021). URL https://www.usenix.org/conference/osdi21/presentation/feng
41. Eskandarian, S., et al.: Oblidb: oblivious query processing for secure databases. Proc. VLDB Endow. **13**, 169–183 (2019). DOI 10.14778/3364324.3364331
42. Evtyushkin, D., et al.: Branchscope: A new side-channel attack on directional branch predictor. In: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, p. 693–707 (2018). DOI 10.1145/3173162.3173204
43. Fei, S., et al.: Security vulnerabilities of sgx and countermeasures: A survey. ACM Comput. Surv. **54**, 1–36 (2021). DOI 10.1145/3456631
44. Ghemawat, S., Dean, J.: leveldb (2011). URL https://github.com/google/leveldb
45. Giantsidi, D., Bailleu, M., Crooks, N., Bhatotia, P.: Treaty: Secure distributed transactions. In: 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 14–27 (2022). DOI 10.1109/DSN53405.2022.00015
46. Goldreich, O., et al.: Software protection and simulation on oblivious rams. J. ACM **43**, 431–473 (1996). DOI 10.1145/233551.233553
47. Götzfried, J., et al.: Cache attacks on intel sgx. In: Proceedings of the 10th European Workshop on Systems Security, pp. 1–6 (2017). DOI 10.1145/3065913.3065915
48. Grubbs, P., et al.: Pump up the volume: Practical database reconstruction from volume leakage on range queries. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, p. 315–331 (2018). DOI 10.1145/3243734.3243864
49. Gruss, D., et al.: Strong and efficient cache side-channel protection using hardware transactional memory. In: Proceedings of the 26th USENIX Conference on Security Symposium, p. 217–233 (2017). URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss
50. Gui, Z., et al.: Encrypted databases: New volume attacks against range queries. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, p. 361–378 (2019). DOI 10.1145/3319535.3363210
51. Gullasch, D., et al.: Cache games – bringing access-based cache attacks on aes to practice. In: 2011 IEEE Symposium on Security and Privacy, pp. 490–505 (2011). DOI 10.1109/SP.2011.22
52. Gyselinck, J., et al.: Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In: Engineering Secure Software and Systems, pp. 44–60 (2018). DOI 10.1007/978-3-319-94496-8_4
53. Han, S., et al.: Veil: A storage and communication efficient volume-hiding algorithm. Proc. ACM Manag. Data **1**, 1–27 (2023). DOI 10.1145/3626759
54. Huo, T., et al.: Bluethunder: A 2-level directional predictor based side-channel attack against sgx. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**, 321–347 (2019). DOI 10.13154/tches.v2020.i1.321-347
55. Intel: Runtime encryption of memory with intel® total memory encryption–multi-key (intel® tme-mk). URL https://www.intel.com/content/www/us/en/developer/articles

/news/runtime-encryption-of-memory-with-intel-tme-mk.html
56. Intel: Intel software guard extensions ssl (2017). URL https://github.com/intel/intel-sgx-ssl
57. Intel: Intel software guard extensions (intel sgx) developer guide (2024). URL https://download.01.org/intel-sgx/latest/linux-latest/docs/Intel_SGX_Developer_Guide.pdf
58. Intel, Inc.: Introduction to intel sgx sealing (2016). URL https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-intel-sgx-sealing.html
59. Jang, Y., et al.: Sgx-bomb: Locking down the processor via rowhammer attack. In: Proceedings of the 2nd Workshop on System Software for Trusted Execution, pp. 1–6 (2017). DOI 10.1145/3152701.3152709
60. Jithin, R., Chandran, P.: Virtual machine isolation - a survey on the security of virtual machines. In: International Conference on Application and Theory of Automation in Command and Control Systems, pp. 91–102 (2014). DOI 10.1007/978-3-642-54525-2_8
61. Karantias, K.: Sok: A taxonomy of cryptocurrency wallets. Cryptology ePrint Archive, Paper 2020/868 (2020). URL https://eprint.iacr.org/2020/868
62. Kellaris, G., et al.: Generic attacks on secure outsourced databases. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, p. 1329–1340 (2016). DOI 10.1145/2976749.2978386
63. Kim, D., et al.: Sgx-lego: Fine-grained sgx controlled-channel attack and its countermeasure. Computers & Security **82**, 118–139 (2019). DOI 10.1016/j.cose.2018.12.001
64. Kim, I., et al.: 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). URL https://www.usenix.org/conference/fast22/presentation/kim-igjae
65. Kim, S.w., et al.: Sentry: A binary-level interposition mechanism for trusted kernel extension. In: The Sixth IEEE International Conference on Computer and Information Technology (CIT'06), pp. 169–169 (2006). DOI 10.1109/CIT.2006.165
66. Kim, T., et al.: Shieldstore: Shielded in-memory key-value storage with sgx. In: Proceedings of the Fourteenth EuroSys Conference 2019, pp. 1–15 (2019). DOI 10.1145/3302424.3303951
67. King, S., et al.: Subvirt: implementing malware with virtual machines. In: 2006 IEEE Symposium on Security and Privacy (S P'06), pp. 14–327 (2006). DOI 10.1109/SP.2006.38
68. Kocher, P., et al.: Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1–19 (2019). DOI 10.1109/SP.2019.00002
69. Koruyeh, E.M., et al.: Spectre returns! speculation attacks using the return stack buffer. IEEE Design & Test **41**, 47–55 (2024). URL https://www.usenix.org/conference/woot18/presentation/koruyeh
70. Lamport, L.: Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) pp. 51–58 (2001). URL https://www.microsoft.com/en-us/research/publication/paxos-made-simple/
71. Lee, D., et al.: Keystone: An open framework for architecting trusted execution environments. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–16 (2020). DOI 10.1145/3342195.3387532
72. Lee, J., et al.: Hacking in darkness: return-oriented programming against secure enclaves. In: Proceedings of the 26th USENIX Conference on Security Symposium, p. 523–539 (2017). URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk
73. 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, p. 1–8 (2021). DOI 10.1145/3491084.3491425
74. Li, W., et al.: Adattester: Secure online mobile advertisement attestation using trustzone. In: Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, p. 75–88 (2015). DOI 10.1145/2742647.2742676
75. Lipp, M., et al.: Meltdown (2018)
76. Liu, C., et al.: Ghostrider: A hardware-software system for memory trace oblivious computation. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, p. 87–101 (2015). DOI 10.1145/2694344.2694385
77. Matetic, S., et al.: Rote: Rollback protection for trusted execution. In: Proceedings of the 26th USENIX Conference on Security Symposium, p. 1289–1306 (2017). URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic
78. Maurer, W.D., et al.: Hash table methods. ACM Comput. Surv. **7**, 5–19 (1975). DOI 10.1145/356643.356645

79. Ménétrey, J., et al.: Attestation mechanisms for trusted execution environments demystified. In: Distributed Applications and Interoperable Systems, pp. 95–113 (2022). DOI 10.1007/978-3-031-16092-9_7
80. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Advances in Cryptology — CRYPTO '87, pp. 369–378 (1988). DOI 10.1007/3-540-48184-2_32
81. Messadi, I., et al.: Precursor: A fast, client-centric and trusted key-value store using rdma and intel sgx. In: Proceedings of the 22nd International Middleware Conference, p. 1–13 (2021). DOI 10.1145/3464298.3476129
82. Meta, Inc.: Rocksdb: a persistent key-value store for fast storage. `https://rocksdb.org/` (2012)
83. Mo, F., et al.: Ppfl: Enhancing privacy in federated learning with confidential computing. GetMobile: Mobile Comp. and Comm. **25**, 35–38 (2022). DOI 10.1145/3529706.3529715
84. Moghimi, A., et al.: Cachezoom: How sgx amplifies the power of cache attacks. In: Cryptographic Hardware and Embedded Systems – CHES 2017, pp. 69–90 (2017). DOI 10.1007/978-3-319-66787-4_4
85. Ongaro, D., et al.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14), p. 305–320 (2014). URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`
86. Orenbach, M., et al.: Eleos: Exitless os services for sgx enclaves. In: Proceedings of the Twelfth European Conference on Computer Systems, p. 238–253 (2017). DOI 10.1145/3064176.3064219
87. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Informatica **33**, 351–385 (1996). DOI 10.1007/s002360050048
88. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy, pp. 238–252 (2013). DOI 10.1109/SP.2013.47
89. Patel, S., et al.: Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, p. 79–93 (2019). DOI 10.1145/3319535.3354213
90. Pattuk, E., Kantarcioglu, M., Khadilkar, V., Ulusoy, H., Mehrotra, S.: Bigsecret: A secure data management framework for key-value stores. In: 2013 IEEE Sixth International Conference on Cloud Computing, pp. 147–154 (2013). DOI 10.1109/CLOUD.2013.37
91. Pinto, S., Santos, N.: Demystifying ARM TrustZone: A comprehensive survey. ACM Computing Surveys (CSUR) **51**, 1–36 (2019). DOI 10.1145/3291047
92. Poddar, R., et al.: Practical volume-based attacks on encrypted databases. In: 2020 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 354–369 (2020). DOI 10.1109/EuroSP48549.2020.00030
93. Popa, R.A., et al.: Cryptdb: Protecting confidentiality with encrypted query processing. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, p. 85–100 (2011). DOI 10.1145/2043556.2043566
94. Priebe, C., Muthukumaran, D., Lind, J., Zhu, H., Cui, S., Sartakov, V.A., Pietzuch, P.: Sgx-lkl: Securing the host os interface for trusted execution. arXiv preprint arXiv:1908.11143 (2019)
95. Priebe, C., Vaswani, K., Costa, M.: Enclavedb: A secure database using sgx. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 264–278 (2018). DOI 10.1109/SP.2018.00025
96. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM **33**, 668–676 (1990). DOI 10.1145/78973.78977
97. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: What it is, and what it is not. In: IEEE TrustCom/BigDataSE/ISPA, pp. 57–64 (2015). DOI 10.1109/Trustcom.2015.357
98. Samsung: Samsung pay: What is it, where is it, and how to use it? (2022). URL `https://www.samsung.com/in/support/mobile-devices/samsung-pay-what-is-it-where-is-it-and-how-to-use-it/`
99. Scarlata, V., et al.: Supporting third party attestation for intel sgx with intel data center attestation primitives (2018). URL `https://api.semanticscholar.org/CorpusID:221506554`
100. Schwarz, M., et al.: Malware guard extension: abusing intel sgx to conceal cache attacks. pp. 1–20 (2020). DOI 10.1186/s42400-019-0042-y
101. Seo, J., et al.: Sgx-shield: Enabling address space layout randomization for sgx programs. In: Network and Distributed System Security Symposium 2017 (2017). DOI 10.14722/NDSS.2017.23037
102. Setty, S., Braun, B., Vu, V., Blumberg, A.J., Parno, B., Walfish, M.: Resolving the conflict between generality and plausibility in verified computation. In: Proceedings of the 8th ACM European Conference on Computer Systems, p. 71–84 (2013). DOI 10.1145/2465351.2465359
103. Shepherd, C., et al.: Establishing mutually trusted channels for remote sensing devices

with trusted execution environments. In: Proceedings of the 12th International Conference on Availability, Reliability and Security, pp. 1–10 (2017). DOI 10.1145/3098954.3098971
104. Shepherd, C., et al.: Lira-v: Lightweight remote attestation for constrained risc-v devices. In: 2021 IEEE Security and Privacy Workshops (SPW), pp. 221–227 (2021). DOI 10.1109/SPW53761.2021.00036
105. Shih, M.W., Lee, S., Kim, T., Peinado, M.: T-sgx: Eradicating controlled-channel attacks against enclave programs. In: Network and Distributed System Security Symposium 2017 (NDSS'17) (2017). DOI 10.14722/ndss.2017.23193
106. Shinde, S., Le, D., Tople, S., Saxena, P.: Panoply: Low-tcb linux applications with sgx enclaves. In: Network and Distributed System Security Symposium 2017 (2017). DOI 10.14722/ndss.2017.23500
107. Shinde, S., et al.: Preventing page faults from telling your secrets. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, p. 317–328 (2016). DOI 10.1145/2897845.2897885
108. Sinha, R., Christodorescu, M.: Veritasdb: High throughput key-value store with integrity. IACR Cryptol. ePrint Arch. **2018**, 251 (2018). URL https://api.semanticscholar.org/CorpusID:4311947
109. Smith, R.: Intel publishes spectre & meltdown hardware plans: Fixed gear later this year (2018). URL https://www.anandtech.com/show/12533/intel-spectre-meltdown
110. Standaert, F.X.: Introduction to Side-Channel Attacks, pp. 27–42 (2010). DOI 10.1007/978-0-387-71829-3_2
111. Tsai, C.C., Porter, D.E., Vij, M.: Graphene-sgx: A practical library os for unmodified applications on sgx. In: Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference, p. 645–658 (2017)
112. Tu, S., Kaashoek, M.F., Madden, S., Zeldovich, N.: Processing analytical queries over encrypted data. Proc. VLDB Endow. **6**, 289–300 (2013). DOI 10.14778/2535573.2488336
113. Van Bulck, J., et al.: Telling your secrets without page faults: stealthy page table-based attacks on enclaved execution. In: Proceedings of the 26th USENIX Conference on Security Symposium, p. 1041–1056 (2017). URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck
114. Van Bulck, J., et al.: Foreshadow: extracting the keys to the intel sgx kingdom with transient out-of-order execution. In: Proceedings of the 27th USENIX Conference on Security Symposium, p. 991–1008 (2018). URL https://www.usenix.org/conference/usenixsecurity18/presentation/bulck
115. Wang, J., et al.: Practical volume-hiding encrypted multi-maps with optimal overhead and beyond. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, p. 2825–2839 (2022). DOI 10.1145/3548606.3559345
116. Wang, W., et al.: Engraft: Enclave-guarded raft on byzantine faulty nodes. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, p. 2841–2855 (2022). DOI 10.1145/3548606.3560639
117. Weiser, S., et al.: Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In: Network and Distributed System Security Symposium 2019 (2019). DOI 10.14722/ndss.2019.23068
118. Weisse, O., et al.: Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. SIGARCH Comput. Archit. News **45**, 81–93 (2017). DOI 10.1145/3140659.3080208
119. Xu, Y., et al.: Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: 2015 IEEE Symposium on Security and Privacy, pp. 640–656 (2015). DOI 10.1109/SP.2015.45
120. Yang, R., et al.: The value of hardware-based security solutions and its architecture for security demanding wireless services. In: Security and Management, pp. 509–514 (2006)
121. Yang, Z., et al.: Spdk: A development kit to build high performance storage applications. In: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 154–161 (2017). DOI 10.1109/CloudCom.2017.14
122. Yuan, X., Guo, Y., Wang, X., Wang, C., Li, B., Jia, X.: Enckv: An encrypted key-value store with rich queries. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, p. 423–435 (2017). DOI 10.1145/3052973.3052977
123. Zhang, Y., Wang, Z., Cao, J., Hou, R., Meng, D.: Shufflefl: Gradient-preserving federated learning using trusted execution environment. In: Proceedings of the 18th ACM International Conference on Computing Frontiers, p. 161–168 (2021). DOI 10.1145/3457388.3458665
124. Zhao, S., et al.: Sectee: A software-based approach to secure enclave architecture using tee. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, p. 1723–1740 (2019). DOI 10.1145/3319535.3363205
125. Zheng, W., et al.: Opaque: An oblivious and encrypted distributed analytics platform. In:

14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pp. 283–298 (2017). URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng

126. Zhou, W., Cai, Y., Peng, Y., Wang, S., Ma, K., Li, F.: Veridb: An sgx-based verifiable database. In: Proceedings of the 2021 International Conference on Management of Data, p. 2182–2194 (2021). DOI 10.1145/3448016.3457308