SACK: Shielding Dynamic Attribute-based Access Control in Persistent Key-Value Stores

Yanjing Ren
The Chinese University of Hong Kong
yjren22@cse.cuhk.edu.hk

Jingwei Li
University of Electronic Science and Technology of China
jwli@uestc.edu.cn

Patrick P. C. Lee
The Chinese University of Hong Kong
pclee@cse.cuhk.edu.hk

ABSTRACT

Enforcing fine-grained access control is critical for secure key-value (KV) stores in cloud environments, yet classical attribute-based encryption incurs significant overhead. We present SACK, a shielded framework leveraging Intel SGX to enable efficient, dynamic attribute-based access control (ABAC) for KV stores in untrusted cloud environments, while ensuring confidentiality, integrity, and freshness. SACK decouples access control and data management by performing ABAC with hardware-assisted shielded execution and leveraging KV separation for secure, efficient, and crash-consistent KV storage. We implement SACK as a middleware system that can run atop general KV stores. Experiments show that SACK achieves high-performance KV operations and lightweight renewal of access rights.

PVLDB Reference Format:
Yanjing Ren, Jingwei Li, and Patrick P. C. Lee. SACK: Shielding Dynamic Attribute-based Access Control in Persistent Key-Value Stores. PVLDB, 19(6): 1128 - 1141, 2026.
doi:10.14778/3797919.3797923

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/adslabcuhk/sack.

1 INTRODUCTION

Modern cloud providers use high-performance persistent key-value (KV) stores to structurally organize data in KV pairs and serve multiple users in shared environments [24, 63, 84]. However, deploying KV stores in the cloud raises vital security concerns. First, there is a risk of unauthorized access to sensitive data by different users that have access to the same KV store [6]. While cloud providers isolate data across users with logical partitioning [10], the risk remains when users share the same physical infrastructure [81, 85, 100]. Second, cloud providers are often considered *untrusted* [75, 91]. Since cloud administrators have full control of storage infrastructure, insider threats are prevalent [15]. There are also concerns that legal frameworks (e.g., the CLOUD Act [67]) allow governments to request access to user data in the cloud without notifying affected users [33].

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 6 ISSN 2150-8097.
doi:10.14778/3797919.3797923

This motivates us to protect cloud-hosted KV stores with *attribute-based access control (ABAC)* [29, 42], which provides fine-grained access management and prevents unauthorized access *without trusting the cloud*. ABAC characterizes a user's access rights by multiple *attributes* and defines access rules for KV pairs by *policies*. Users can access specific KV pairs only if their attributes satisfy a specified policy (see Section 2.1 for details). Enforcing ABAC on encrypted KV pairs can be done by *attribute-based encryption (ABE)* [11, 37, 74, 82], a public-key cryptographic primitive that encrypts data based on a policy and allows only authorized users (i.e., neither unauthorized users nor cloud providers) to decrypt data.

ABE, however, incurs substantial performance overhead. First, ABE relies on computationally expensive public-key cryptography, whose overhead increases as policy complexity grows. Also, ABE embeds policies into encrypted outputs and increases storage overhead. Furthermore, when a user's attributes are updated, ABE must re-issue private keys to other users sharing the same updated attributes and re-encrypt affected KV pairs, leading to excessive computational overhead [43, 98].

We present SACK, a shielded framework that enables efficient, dynamic ABAC for KV stores in untrusted cloud environments, while preserving essential security guarantees including confidentiality (i.e., data remains inaccessible to unauthorized parties), integrity (i.e., data remains intact), and freshness (i.e., only up-to-date data is processed). SACK's core idea is to decouple access control and data management. It leverages Intel Software Guard Extensions (SGX) [50] to manage attributes and policies and realize ABAC inside a hardware-assisted trusted execution environment (called an enclave). It extends KV separation [60] with security protection by storing the keys and metadata of KV pairs separately from values in different secure storage areas. It maintains high performance in core KV operations, including writes (i.e., inserting new KV pairs), reads (i.e., retrieving KV pairs by specific keys), scans (i.e., retrieving KV pairs over a key range; a.k.a. range queries), and updates (i.e., updating KV pair values). The shielded ABAC approach necessitates re-designs of garbage collection and crash consistency, which SACK addresses with security and efficiency guarantees.

SACK is implemented as a middleware system that can run atop general KV stores. As a case study, we deploy SACK atop RocksDB [28] and compare SACK with the state-of-the-art shielded KV store TWEEZER [52], which achieves confidentiality, integrity, and freshness, but without ABAC. SACK reduces the average write and read latencies of ABE's software implementation by 79.9% and 92.8% under a single attribute and policy, respectively. It also achieves throughput gains of 1.9× in update-heavy workloads and 5.4× in read-only workloads over TWEEZER, albeit being worse than TWEEZER in scans, and supports fast renewal of access rights.

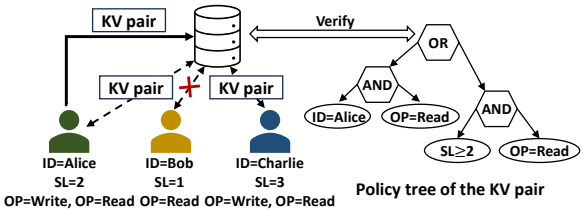


Figure 1: Example of ABAC.

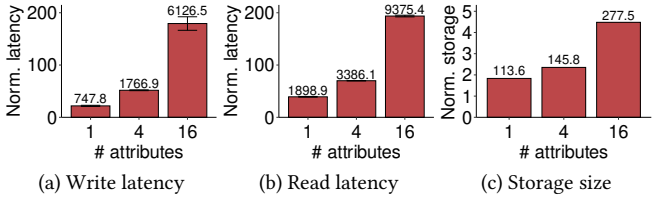


Figure 2: Normalized write latencies, read latencies, and storage size of ABE with respect to without ABE. We report absolute latencies (μs) and storage sizes (GiB) above the bars. Without ABE, the write and read latencies are 34.2μs and 48.4μs, respectively, while the total storage size is 61.9 GiB.

2 BACKGROUND AND MOTIVATION

2.1 Attribute-based Access Control (ABAC)

ABAC is essential for protecting KV stores against unauthorized access, by specifying the access rights of users to KV pairs in the form of *attributes*. We consider an organization with multiple users, where each user is associated with one or multiple attributes. Each attribute is represented as a string (`type=value`) defined by an *administrator* (i.e., a privileged user responsible for managing access rights and the KV store). For example, the administrator can define an attribute of type `identity` for user identification, and a user, say Alice, is assigned an attribute (`identity=Alice`).

Access rights are defined in *policies* represented in the form of a *policy tree*, which combines multiple attributes with logical operators (e.g., `AND` or `OR`). To check if a policy is satisfied, the policy tree recursively computes a Boolean value (true or false) for each node in a bottom-up manner. Each leaf node specifies the condition of an attribute and returns true if a user has an attribute whose value satisfies the condition (or false otherwise). Each non-leaf node represents a logical operator that combines the Boolean results of its corresponding sub-trees. The policy tree grants a user access if the root node returns true.

Figure 1 shows an example of ABAC, where users are associated with three types of attributes: (i) identity (`ID`), which specifies a user's name, (ii) security level (`SL`), a numerical value that specifies a user's privilege to access KV pairs (a higher value implies higher privilege), and (iii) operations (`OP`), which specify permitted operations on KV pairs (e.g., `Read` or `Write`). The policy tree indicates that Alice and Charlie have access rights since their attributes satisfy the conditions of the left and right sub-trees under the root node, respectively. Bob's access is denied since his `ID` does not match `Alice` and his `SL` is below 2.

2.2 Limitations of Attribute-based Encryption

Attribute-based encryption (ABE) [11, 37, 74, 82] is a public-key cryptographic approach for enforcing ABAC in untrusted environments. It encrypts data based on a policy tree, ensuring that only authorized users with permitted attributes can decrypt it, while unauthorized users cannot. At a high level, ABE assigns each user a private key derived from the user's associated attributes. It encrypts a data object based on the policy tree with a public key, and decrypts an encrypted data object only when the attributes for a user's private key match the policy tree. For formal cryptographic treatments of ABE, we refer readers to the literature [11, 37, 74, 82].

Realizing ABE in practical KV stores is non-trivial. We examine a naïve baseline that applies ABE to each KV pair and highlight three deployment-oriented limitations.

High performance overhead. Since ABE relies on public-key cryptography, it incurs much higher computational overhead in encryption and decryption than symmetric-key cryptography. The computational overhead increases with the number of attributes in the policy tree. To justify, we implement ciphertext-policy ABE [11] using OpenABE [99] and define a policy tree that combines attributes with the `AND` operator. We use YCSB [22] to issue ABE-protected reads and writes to RocksDB (v6.14.5) [28]. We load 64 M 1-KiB KV pairs with 24-byte keys and 1,000-byte values. We issue 5 M writes and reads following a Zipf distribution with a Zipfian constant of 0.99 (default in YCSB). To make KV pairs retrievable, we keep the keys of KV pairs intact and only encrypt and decrypt values using ABE. Figures 2(a) and 2(b) show the normalized write and read latencies with respect to without ABE, respectively (see Section 5.1 for testbed details). Even with a single attribute, ABE incurs write and read latency amplifications of 21.9× and 39.2×, respectively. Such amplifications increase with the number of attributes (e.g., 179.4× and 193.7× for 16 attributes), since the policy tree expands and increases encryption and decryption overhead.

High storage overhead. In ABE, each value is encrypted by a public key under a policy and a random factor, such that (i) the policy tree is embedded in the encrypted value to reflect the policy and (ii) identical policy trees or values are mapped to different encryption results for *semantic security* (i.e., an adversary cannot infer the policy and the original value from the encrypted value). Figure 2(c) shows the normalized storage overhead with respect to without ABE. With 16 attributes, the storage blowup is 4.5×.

High attribute update complexity. ABE supports dynamic ABAC via attribute or policy updates. However, affected KV pairs must be re-encrypted under updated attributes and policies [43]. Also, we need trusted key management (which may be a third-party service) to coordinate the attribute update process, which complicates deployment.

2.3 Access Control with Shielded Execution

A natural way to address the limitations of the naïve ABE baseline (Section 2.2) is to use ABE to encrypt a small secret, which then serves as the symmetric key for encrypting the actual KV pairs. The main challenge is to retain strong security guarantees while preserving the performance of modern KV stores under realistic workloads. This motivates the use of *shielded execution*, which creates a hardware-assisted trusted execution environment (TEE) to achieve secure and efficient ABAC while preserving high KV store

performance. We focus on Intel SGX [50], which is supported by modern cloud providers [49] and Intel Xeon platforms [79]. SGX enables target applications to create their own shielded environments for fine-grained security protection, in contrast to other high-overhead TEE technologies (e.g., AMD SEV [2] and Intel TDX [47]) that create secure virtual machines (VMs) to run target applications. Even though VM-based TEE technologies can be used with lightweight secure VMs [56], they still incur higher overhead than SGX due to larger protection scopes.

SGX basics. SGX maintains an *enclave*, a trusted environment that ensures the confidentiality and integrity of in-enclave contents via hardware protection. An enclave is hosted inside an isolated memory region called the *enclave page cache (EPC)*, with a maximum size of 256 MiB in the earlier generation SGXv1 [44] and 512 GiB in the latest generation SGXv2 [46]. SGX provides two functions for secure interactions between applications in untrusted memory and the enclave: (i) *enclave calls (ECalls)*, which allow applications to securely access in-enclave contents, and (ii) *outside calls (OCalls)*, which allow the enclave to access external resources. Both ECalls and OCalls incur substantial context-switch overhead [40].

SGX supports *remote attestation* [50], which allows a remote entity to cryptographically verify an enclave's integrity and confirm that the enclave contains correct, unmodified code and data. Upon successful verification, a secure channel is established between a user and the enclave, and the user can securely access enclave services [50]. Note that remote attestation can be performed each time when a user connects to the enclave, and the secure channel is maintained throughout the session. If the connection is lost, the user can re-establish the channel via remote attestation.

For crash tolerance, applications can securely persist in-enclave contents to external storage via *sealing*, in which an enclave generates a secret *sealing key* for encrypting and authenticating data to be persisted [50]. To recover in-enclave contents from crashes, applications can restart the enclave, which decrypts sealed data with the same sealing key. The sealing key is bound to the specific hardware on which it was created on, so as to prevent direct migration of sealed data to a different machine. This hardware dependency is also enforced in public cloud services that support SGX [17].

Can simple access control work? SGX enables secure and efficient ABAC by hosting attribute and policy management within an enclave. The enclave securely maintains user attributes, verifies user authenticity via secure channels (e.g., SSL/TLS), and enforces access control based on policy trees. If access is allowed, the enclave securely processes authorized requests with the persistent KV store, thereby mitigating ABE's performance and storage overheads. Note that we do not consider putting the whole KV store inside the enclave (even with the large EPC size in the latest SGXv2 generation), as it cannot handle the increased data volume and address data persistence (which SGX does not support).

One naïve approach of implementing ABAC is to directly embed a policy identifier with each KV pair and keep the policy verification logic inside the enclave. This follows the idea of traditional POSIX-based access control for file systems [39], where a static set of access policies is attached to each file. However, this naïve approach is inadequate for KV storage, which has no hierarchical namespace for organizing access rights as in file systems, and incurs highly expensive access right updates under dynamic access control. For example, if two access policies are merged into one, the naïve approach needs to scan the entire KV store and update the identifier for each affected KV pair. This leads to significant I/O overhead.

3 SECURITY GOALS

SACK aims for four security properties: (i) *confidentiality*, KV pairs are encrypted in storage to prevent unauthorized access; (ii) *integrity*, any unauthorized change to attributes, policies, and KV pairs is detectable; (iii) *freshness*, all operations are performed using the latest attributes, policies, and KV pairs; and (iv) *dynamic ABAC*, attributes and policies can be securely renewed.

We consider an adversary that can compromise the KV store and one or multiple users. We host the enclave alongside the KV store. Even if the adversary compromises the KV store, it cannot access any in-enclave contents, but can access contents outside of the enclave, monitor ECall/OCall interactions, and tamper with KV pairs in storage. In addition, the adversary can compromise both the KV store and some users to launch collusion attacks, where the compromised KV store returns unauthorized KV pairs to the compromised users whose access rights have been revoked.

We make the following security assumptions. Users can authenticate the enclave via remote attestation (Section 2.3), while the enclave can authenticate users (e.g., via SSL/TLS). The enclave exports ECalls to add or remove users and dynamically update access rights for existing users and KV pairs. We do not consider access pattern leakage, from which an adversary can learn if two requests are from the same or different keys. We can mitigate such leakage via oblivious RAM [1, 35, 87], yet the performance overhead of oblivious RAM is significant and can reach 83× in SGX implementation [1]. We also do not consider denial-of-service attacks, which can be defended by rate-limiting requests issued by a user [64]. Furthermore, we do not consider known SGX vulnerabilities [13, 51, 89], which are already fixed by Intel [45] and protected by existing countermeasures [68, 71, 78].

4 SACK DESIGN

4.1 Design Goals and Overview

SACK enhances existing SGX-based KV stores (e.g., [8, 52]) with dynamic ABAC and general applicability, while preserving their core security features (i.e., confidentiality, integrity, and freshness). It aims for the following design goals.

- *Security guarantees:* SACK achieves confidentiality, integrity, freshness, and dynamic ABAC (Section 3).
- *Efficient use of SGX:* While SGXv2 supports a huge EPC size (Section 2.3), the actual available EPC space is limited by host memory. Thus, SACK carefully manages attributes and any metadata inside the enclave, while limiting ECall/OCall overhead.
- *Low storage overhead:* SACK should limit storage overhead introduced by security measures (e.g., encryption keys and metadata), and allow reclaiming the storage space of stale KV pairs, while preserving security guarantees.
- *Crash consistency:* SACK must still ensure crash consistency of both in-enclave contents and KV storage.
- *General applicability:* SACK is compliant with general KV stores and uses only the standard KV interface for storage management.

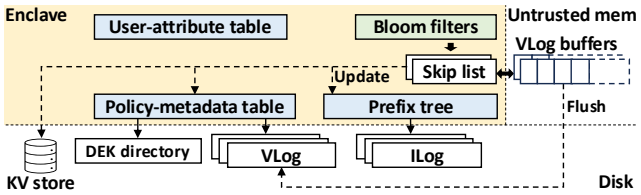


Figure 3: SACK's architecture.

Design novelty. SACK's core idea is to *decouple access control and data management*. It manages attributes and policies and evaluates users' access rights inside the enclave. Such shielded ABAC provides two key design features. First, instead of directly encrypting KV pairs by ABE, SACK uses ABE to manage secret keys (typically smaller than KV pairs) inside the enclave, where the secret keys are used to encrypt and decrypt KV pairs based on symmetric-key cryptography. This mitigates the computational overhead of ABE (based on public-key cryptography). Second, SACK enables lightweight dynamic access control by exporting enclave functions to securely update attributes and policies, thereby avoiding costly key re-issuing and data re-encryption in ABE.

To achieve secure, efficient data management outside of the enclave, SACK uses *KV separation* [60], which stores only keys and metadata in the KV store for indexing and values in separate storage space. The original motivation of KV separation is to improve indexing performance by keeping a small index structure (i.e., the *log-structured merge tree (LSM-tree)* [69]) with only keys and metadata. SACK preserves these performance benefits, and further securely and efficiently applies access control to value management. Note that SACK can be applied to other index structures beyond the LSM-tree, without modifying the index structure implementation.

Design trade-offs. The decoupled design of SACK incurs metadata overhead for separately managing access control and KV pairs. Its KV separation also incurs extra I/Os to separate storage space (as observed in [60]) and complicates crash consistency across the KV store and separate data structures.

Design roadmap. Figure 3 depicts SACK's architecture, including (i) external data structures hosted outside of the enclave (i.e., in untrusted memory and on disk) and (ii) in-enclave data structures. SACK incorporates various techniques: (i) lightweight renewal of access rights (Section 4.2), (ii) efficient freshness verification in reads (Section 4.3), (iii) garbage collection (GC) for encrypted data (Section 4.4), and (iv) crash consistency for both in-enclave contents and KV storage (Section 4.5). Finally, we provide security analysis (Section 4.6) and implementation details (Section 4.7).

4.2 Access Control

Secret key management. SACK manages three types of secret keys for symmetric-key cryptography: the master secret K_M, the signature secret K_S, and a set of *data encryption keys (DEKs)* $\{K_p\}$, where each K_p is linked to a policy with the *policy ID (PID)* p. K_M is used to encrypt the keys of KV pairs for confidentiality under KV separation (Section 4.3), while K_S is used to generate a *hash-based message authentication code (HMAC)* for integrity checking. Both K_M and K_S are initialized during enclave bootstrapping and securely stored inside the enclave.

The DEKs are used for encrypting and decrypting KV pairs governed by policies based on symmetric-key cryptography. Instead of using a single DEK for all KV pairs, employing per-policy DEKs ensures strict isolation of KV pairs across different policies. This approach aligns with ABE, which encrypts data under distinct policies using logically distinct keys. When a new policy is created, the enclave randomly generates a new DEK linked to the policy. All DEKs are maintained in an external encrypted storage area called the *DEK directory*. They are protected by ABE and accessible only through the enclave when the attributes are satisfied under ABE.

SACK uses ABE to protect DEKs, as ABE enforces access control policies on encrypted data and systematically supports access right updates when attributes or policies change. With KV separation, SACK can achieve lightweight dynamic access control for access right updates, without scanning the entire KV store (Section 2.3).

To efficiently manage numerous policies, SACK hosts the DEK directory outside of the enclave, as the DEKs under ABE protection incur high storage overhead. For example, a 32-byte DEK encrypted with ABE expands to 564 bytes when the policy includes a single attribute, and 2,188 bytes when the policy includes 10 attributes.

Access management. SACK manages access control via two in-enclave structures: a *user-attribute table* and a *policy-metadata table*. The user-attribute table tracks the mappings between each user ID and the associated attributes. SACK allows only the administrator to update the user-attribute table (e.g., creating users and assigning attributes) using a secure authenticated ECall. Each attribute is represented as a string (`type=value`) (Section 2.1), and multiple attributes are concatenated with a delimiter '|'. Figure 4 shows the entry format of the user-attribute table. Each entry not only stores the mapping of the user ID and its associated attributes, but also caches the PIDs and the corresponding DEKs of the valid policies satisfied by the user's attributes. This saves the overhead of retrieving DEKs from persistent storage in subsequent KV operations.

The policy-metadata table references the DEK directory and VLogs (see Section 4.3 for details on VLog management) based on policies, each represented as a byte string encoded from its policy tree. Figure 5 shows the entry format of the policy-metadata table. Each entry in the policy-metadata table stores the PID p for locating DEKs (protected by ABE under the policy) in the DEK directory, as well as the HMAC of the DEK K_p. It also tracks the VLogs associated with the policy, identified by VLog IDs and verified by Merkle tree roots [62] (Section 4.3).

Upon receiving the first request from a user, the enclave checks the user's attributes from the user-attribute table and identifies all valid policies. It then retrieves the corresponding DEKs from the DEK directory using the PIDs, and caches the PIDs and DEKs in the user-attribute table for the user's subsequent requests.

Lightweight dynamic access control. For dynamic access control, SACK supports two primary cases of changes to access rights:

- *Case 1: Changing the access rights of users.* When a user's access rights change, the administrator updates the user's attributes in the user-attribute table. The enclave then drops the cached PIDs and DEKs that are no longer valid, and caches the new PIDs and DEKs under the updated attributes to enforce the user's new access rights. Note that the policy-metadata table remains intact.
- *Case 2: Changing the access rights to KV pairs.* When the access rights to a KV pair are updated (e.g., a new attribute is needed to

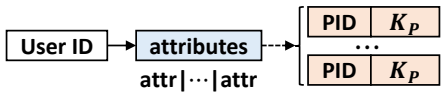


Figure 4: Entry format in the user-attribute table.

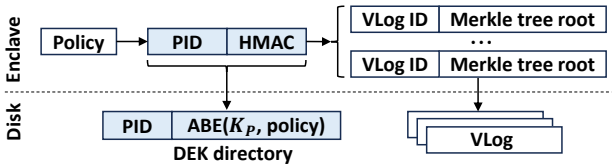


Figure 5: Entry format in the policy-metadata table.

access the KV pair), the administrator updates the corresponding policy in the policy-metadata table. SACK supports three types of policy updates: (i) changing the current policy to a new policy, (ii) transitioning from one policy to another existing policy, and (iii) merging two policies into a new policy (i.e., the original two policies become sub-trees of the new policy). In all cases, the enclave consolidates PIDs, DEKs, and VLogs into the resulting policy's metadata in the policy-metadata table. It also updates ABE-protected DEKs with the resulting policy in the DEK directory, by first retrieving DEKs based on the PIDs of the policies being changed, decrypting the retrieved DEKs with the attributes of old policies, and re-encrypting the DEKs with the resulting policy using ABE. Furthermore, it traverses the user-attribute table to identify the users that currently have access rights to the KV pairs under the old policy (based on the PID). It removes any cached PID and DEK if a user no longer retains access rights under the new policy.

Both cases achieve lightweight updates of access rights, as the currently stored KV pairs are unaffected. To avoid access violations, SACK temporarily suspends any requests during updates, but the suspension only lasts for a short duration as updates are lightweight. Currently, SACK does not support multiple policies for a KV pair. When a KV pair is rewritten with a new policy, the previous version of the KV pair (under the old policy) will be invalidated, and only the latest version is accessible under the new policy.

4.3 Data Management

KV separation. SACK adopts KV separation [60] to support operations for *general* KV stores. It stores the metadata records of KV pairs (called *KV-metas*) in the KV store, while storing encrypted KV pairs in multiple append-only *value logs (VLogs)* in persistent storage. Both the keys and values of KV pairs are encrypted for confidentiality, yet KV pairs should still remain searchable even after encryption. Thus, SACK deterministically encrypts the keys of KV pairs by the master secret K_M, so that identical unencrypted keys are mapped to identical encrypted keys for queries. It also probabilistically encrypts values by a random initialization vector [23] and K_p for the same policy with PID p, such that identical values are mapped to distinct encrypted outputs for semantic security.

Figure 6 shows KV separation and the VLog structure, where each KV pair is converted into a KV-meta and an encrypted KV pair. A KV-meta is identified by the encrypted key (encrypted by

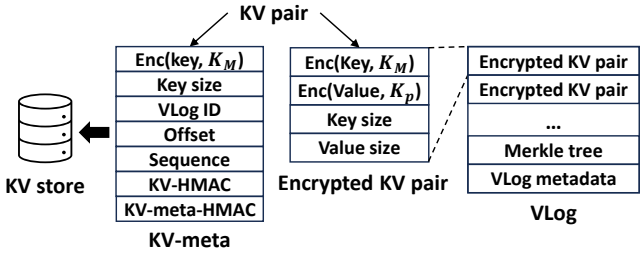


Figure 6: KV separation and VLog data structure.

K_M) and keeps multiple metadata fields for KV pair indexing and integrity verification: (i) the encrypted key size, (ii) the VLog ID (incremented each time a new VLog is created) for identifying the VLog holding the KV pair, (iii) the offset of the KV pair in the VLog, (iv) the sequence number of the KV pair, (v) the HMAC of the KV pair (called *KV-HMAC*), and (vi) the HMAC of all above metadata fields (called *KV-meta-HMAC*). Each VLog groups encrypted KV pairs under the same policy to mitigate overhead of changing access rights to KV pairs (Case 2 in Section 4.3), achieved by updating only VLog metadata in the policy-metadata table without traversing and updating all affected KV pairs. Each encrypted KV pair in a VLog includes the key and value sizes and the encrypted key and value (encrypted by K_M and K_p, respectively). Also, each VLog's end stores a Merkle tree formed by the KV-HMACs of all KV pairs in the VLog for integrity checking, plus the metadata fields including the PID, total KV pair size, and Merkle tree size.

To mitigate the enclave's context-switch overhead (Section 2.3), SACK adopts buffered writes [60] to trade immediate persistence for performance. It maintains *VLog buffers* in untrusted memory to batch small I/Os into a single large I/O. Each VLog buffer temporarily holds encrypted KV pairs under the same policy (note that SGX allows the enclave to directly operate on untrusted memory via memory addressing). When a VLog buffer is full, the enclave flushes encrypted KV pairs to disk with a single OCall. Also, the enclave maintains multiple *skip lists* to efficiently index unflushed KV pairs in VLog buffers (for reads).

KV-meta indexing. Keeping KV-metas in the KV store faces two challenges. First, encrypting the keys of KV pairs disrupts the ordering of the original unencrypted KV pairs, thereby hindering scans. Second, verifying the freshness of a KV pair becomes infeasible, as an adversary can compromise the KV store to return an older KV-meta version and break freshness. Note that HMACs in KV-metas cannot ensure freshness, as old KV-metas still have valid HMACs.

SACK separately manages KV-metas in multiple on-disk *index logs (ILogs)*, each containing KV-metas whose unencrypted keys share a common prefix. Figure 7 shows the ILog layout. Each ILog is identified by an ILog ID (incremented each time when an ILog is created) and comprises a *sorted part* and an *unsorted part*. The sorted part contains all KV-metas sorted by *unencrypted* keys for fast reads, while the unsorted part contains newly inserted KV-metas in an append-only manner to support fast writes. The unsorted part is merged into the sorted part when its number of KV-metas exceeds a threshold, while each ILog has a capacity (i.e., the maximum number of KV-metas that can be stored) and is split in GC when the capacity is reached (Section 4.4).

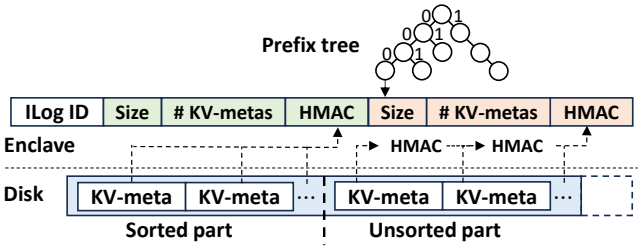


Figure 7: Prefix tree and ILogs.

SACK indexes KV-metas in all ILogs with a binary *prefix tree* inside the enclave (Figure 7). In the prefix tree, each leaf node corresponds to an ILog, while each root-to-leaf path represents a bit-level prefix of unencrypted keys in the ILog for lookups. Each leaf node contains the ILog ID and two metadata tuples, one for the sorted part and another for the unsorted part. Each metadata tuple contains the total size of KV-metas, number of KV-metas, and an HMAC that protects against ILog tampering. For the sorted part, the HMAC is computed from the KV-meta-HMACs of all KV-metas, while for the unsorted part, the HMAC is computed from the current HMAC and the KV-meta-HMACs of newly inserted KV-metas. SACK ensures freshness by verifying retrieved KV-metas based on ILogs and the prefix tree.

KV operations. SACK supports the basic KV operations, including writes, reads, scans, and deletes.

- *Writes.* Suppose that a KV pair is written under a policy with PID p. The enclave locates the DEK K_p and encrypts the key and value of the KV pair by K_M and K_p, respectively. It stores the KV-meta in a skip list and the encrypted KV pair in a VLog buffer. Both the skip list and the VLog buffer are associated with PID p. When the VLog buffer is full, the enclave flushes the contents in the skip list and VLog buffer to the KV store and the VLog associated with PID p, respectively. It also appends a Merkle tree to the VLog and flushes each KV-meta to the unsorted part of an ILog based on the prefix tree.
- *Reads.* Suppose that a KV pair is read. Based on the encrypted key (encrypted by K_M) of the KV pair, the enclave first checks if the skip lists and VLog buffers cache the KV pair; if not, it retrieves the KV-meta from the KV store and the encrypted KV pair from the corresponding VLog based on the VLog ID and offset in the KV-meta. It verifies the KV-meta using the KV-meta-HMAC. Also, it verifies the KV pair using the KV-HMAC, the Merkle tree in the VLog, and the Merkle tree root in the policy-metadata table. Finally, the enclave decrypts the encrypted KV pair by K_M and K_p and returns the results to the user, assuming that the user's attributes satisfy the policy with PID p. To verify KV pair's freshness, the enclave retrieves the corresponding ILog that contains the KV-meta, and verifies the ILog by the HMACs of both sorted and unsorted parts. Note that if the KV-meta has multiple versions, they are in the same ILog as SACK groups KV-metas by unencrypted keys. The enclave checks if the latest version matches the one retrieved from the KV store.
- *Scans.* SACK supports scans over ranges of unencrypted keys using the prefix tree and ILogs. Specifically, the enclave uses the prefix tree to retrieve all ILogs that contain KV-metas in the key range. It then decrypts the ILogs, sorts the KV-metas from both the sorted and unsorted parts by their unencrypted keys, and filters out invalid entries based on sequence numbers, so that all KV-metas within the specified key range are accurately identified. Like reads, the enclave verifies ILogs by the HMACs of both sorted and unsorted parts and retrieves encrypted KV pairs from the corresponding VLogs, and returns the latest versions of KV pairs (assuming that the user has access rights).
- *Deletes.* Similar to RocksDB's tombstone design [28], SACK deletes a key by writing a tombstone (a special KV-meta) to the ILog, so that the corresponding KV-metas of the deleted key are treated as invalid. It also issues a delete operation to the KV store. The tombstones in the ILogs and the corresponding KV pairs in the VLogs are later physically removed during GC.

Optimizing freshness verification in reads. To verify the freshness of a KV pair being read, the enclave retrieves the corresponding ILog and locates the KV-meta. This incurs significant I/Os as the ILog also contains KV-metas of other KV pairs. Also, the enclave performs HMAC verifications, which incur substantial computational overhead.

SACK introduces two types of Bloom filters (BFs) [12], namely *per-ILog insert BFs* and an *update BF*, to avoid accessing ILogs during reads for freshness verification. Our insight is that freshness verification is needed only when KV pairs are updated and have multiple versions. For newly inserted KV pairs, there is only a single version, so an adversary cannot replay outdated KV pairs without valid KV-metas from older versions. Note that these BFs are distinct from the SSTable BFs in RocksDB [28], which aim to eliminate reads to KV pairs that do not exist in SSTables. Instead, the BFs in SACK are tailored to identify newly inserted KV pairs and optimize freshness verification for updates.

Each per-ILog insert BF is allocated with a small memory budget (1.5 KiB by default (Section 5.1)) and stored in untrusted memory, so as to accommodate data growth (i.e., more ILogs) while maintaining a low false positive rate. The per-ILog insert BFs are loaded into an in-enclave least-recently-used (LRU) cache *on demand* for modifications to ensure integrity and freshness. They are authenticated by HMACs stored in the enclave, and the HMACs are updated when the per-ILog insert BFs are evicted from the enclave. On the other hand, the update BF always resides in the enclave.

When flushing a VLog buffer, the enclave checks if the key of a KV pair exists in the corresponding per-ILog insert BF. If the key is new (i.e., not present in the per-ILog insert BF), the enclave adds the key to the per-ILog insert BF; otherwise, the enclave adds the key to the update BF. To read a KV pair, the enclave checks the update BF. If the key does not exist, the enclave knows the key is new and will not access ILogs. Note that any false positive (i.e., a new key is treated as an updated one) does not compromise correctness, as the enclave still accesses ILogs and only incurs performance overhead.

Since not all reads will access ILogs with our BF optimization, an attacker may attempt to replay a deleted KV pair. Specifically, consider a key with only a single version (i.e., the key is not added to the update BF). If the key is deleted (with its tombstone written) but not yet physically removed by GC, the attacker can control the KV store, retrieve the KV-meta of the deleted key, replay the KV-meta to the enclave via the controlled KV store, and finally access the KV pair in the VLog that remains valid before GC. To

address this security issue, SACK also inserts deleted keys into the update BF during deletes. This ensures that any subsequent reads to a deleted key must access the ILogs, enabling the enclave to detect the deletion via the ILogs and stop the reads.

4.4 Garbage Collection (GC)

While the KV store has its own GC mechanism, SACK regularly performs GC on VLogs and ILogs to reclaim free space from stale KV pairs. SACK starts with the GC of ILogs and uses the results to guide the GC of VLogs. It also performs GC on BFs to prevent their saturation. It ensures confidentiality, integrity, and freshness during GC without compromising access control.

GC of ILogs. To maintain efficient reads to ILogs, SACK performs *sort-based GC* for an ILog to merge the unsorted part into the sorted part when the number of KV-metas in the unsorted part exceeds a threshold (e.g., 5% of the ILog capacity in our prototype). The enclave loads the whole ILog and sorts all KV-metas based on their decrypted keys. It removes stale KV-metas and keeps only the latest versions based on the sequence numbers of KV-metas. If the latest KV-meta is a tombstone (i.e., the key is deleted), the enclave removes all KV-metas of the deleted key. Finally, the enclave includes the sorted KV-metas into the sorted part and rebuilds the corresponding per-ILog insert BF based on all valid keys.

If the total number of KV-metas in an ILog reaches the capacity, SACK performs *split-based GC* to prevent the ILog from growing indefinitely. In the prefix tree, the enclave splits the leaf node corresponding to the ILog into two new leaf nodes and extends the prefix length by one bit. It also splits KV-metas into the sorted parts of the respective two ILogs, sorted by the decrypted keys. A new per-ILog insert BF is built for each new ILog based on all valid keys.

During the GC of ILogs, the enclave records stale KV-metas and tombstones in a temporary file, called *GC-Log*, which is stored outside of the enclave and protected by an in-enclave HMAC against tampering.

GC of VLogs. During idle periods, SACK suspends KV operations, and the GC of VLogs removes stale KV pairs based on the GC-Log and updates indexes with new KV-metas for remaining KV pairs in the affected VLogs. Those remaining KV pairs are written to new VLogs and the policy-metadata table is updated. SACK always performs the GC of *all* ILogs first to ensure that stale KV pairs are fully removed, and the update BF is reset only after the GC of VLogs is completed. The new enclave version is then persisted, and the GC-Log and affected VLogs are reclaimed to ensure reliable crash recovery (Section 4.5).

GC of BFs. To prevent BF saturation, SACK rebuilds per-ILog insert BFs during ILog GC based on all valid keys in the ILogs. It also resets the update BF after VLog GC is completed. All KV operations are temporarily suspended until the GC of BFs is done, yet the suspension time is very short.

4.5 Crash Consistency

Crash consistency requires a KV store to maintain correct and recoverable storage even after a system crash. SACK employs VLogs for write-ahead logging as in KV separation [60] and maintains crash consistency among the enclave, KV store, VLogs, and ILogs. It also maintains integrity and freshness during crash recovery.

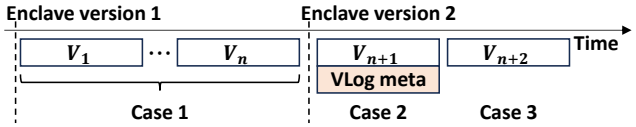


Figure 8: Crash recovery for VLogs.

Enclave sealing. The contents of in-enclave structures are protected by sealing (Section 2.3), yet frequent sealing incurs high overhead. Thus, SACK seals the entire enclave and persists the sealed enclave at a low frequency (e.g., every hour). It further applies sealing to the metadata of any new VLog created and flushed (i.e., the VLog ID, policy, and Merkle tree root). It keeps the sealed VLog metadata until the sealed enclave is persisted.

Enclave versioning. SACK keeps an *enclave version count (EVC)* in the enclave to track the number of times the enclave has been sealed. When the enclave is sealed, the EVC is incremented and persisted in the sealed enclave. The EVC is initialized with a random number when the enclave is first created, and is sent to all users during enclave sealing. Thus, a user can verify the EVC of the enclave and ensure that only the latest enclave is used.

Crash recovery for ILogs. In crash recovery, SACK unseals the latest sealed enclave and recovers ILogs to be consistent with the latest sealed enclave. The enclave reads each ILog included in the prefix tree (restored from the latest sealed enclave), and verifies the HMACs of both sorted and unsorted parts. Note that the GC of an ILog may occur after the latest sealing, so the enclave may not track removed KV-metas after being unsealed. Thus, the enclave uses the GC-Log (Section 4.4) to recover any removed KV-metas.

Crash recovery for VLogs. Figure 8 shows how SACK handles crash recovery for VLogs. Suppose that there exists a sequence of VLogs $V_1, V_2, \cdots, V_n, V_{n+1}, V_{n+2}$ and a crash happens, and there are three cases: (i) $V_1, V_2, \cdots, V_n$ are generated before the enclave is sealed; (ii) V_{n+1} is generated after the sealed enclave and its VLog metadata is sealed; and (iii) V_{n+2} is generated after the sealed enclave, but its VLog metadata is not yet sealed before the crash happens. Now, SACK restarts from the crash. It unseals the latest sealed enclave, recovers the Ilogs, and checks the consistency of all VLogs on disk. It handles the three cases as follows:

- *Case (i):* The policy-metadata table in the latest sealed enclave records $V_1, V_2, \cdots, V_n$, all of which are persisted. Thus, there is no special recovery action.
- *Case (ii):* SACK unseals V_{n+1}'s metadata, and verifies V_{n+1} on disk based on V_{n+1}'s Merkle tree. If V_{n+1} is valid, SACK decrypts all KV pairs in V_{n+1}, rewrites KV-metas into the KV store and ILogs, and updates the VLog ID of V_{n+1} and Merkle tree root in the policy-metadata table; otherwise, SACK discards the invalid V_{n+1}. Note that ILogs (recovered to the latest sealed enclave) now have consistent KV-metas with VLog V_{n+1}.
- *Case (iii):* Since SACK cannot verify the authenticity of V_{n+2} without the sealed metadata, it discards V_{n+2}.

4.6 Security Analysis

We discuss how SACK maintains its security properties (Section 3).

Confidentiality. An adversary can have access to any data outside of the enclave, including external structures (Figure 3 in Section 4.1)

and the sealed enclave (Section 4.5). SACK encrypts all sensitive contents (e.g., KV pairs, DEKs, and sealed enclave) to prevent unauthorized access. Note that the keys of KV pairs are deterministically encrypted to preserve searchability (Section 4.3), and non-sensitive contents (e.g., metadata fields of KV-metas) are unencrypted. Also, an adversary may monitor the enclave's ECalls/OCalls and disk I/Os to learn the access pattern of KV pairs, or infer the ordering of KV pairs from the sorted parts of ILogs. Nevertheless, they do not introduce additional information leakage about the values of KV pairs, which are probabilistically encrypted for semantic security (Section 4.3). Also, since the ordering across ILogs is independent of the ordering of keys in each ILog, the leakage of ordering of KV pairs is limited to individual ILogs and cannot be used to deduce the ordering of all KV pairs.

Integrity. SACK can detect any tampering against encrypted KV pairs, KV-metas, and DEKs via HMAC-based verification. It also enforces integrity for the sealed enclave via SGX sealing (Section 4.5).

Freshness. SACK prevents any adversarial attempt to reuse outdated KV pairs, KV-metas, or DEKs. It ensures freshness mainly through ILogs, by only returning KV-metas with the latest sequence number. Also, it protects ILogs via HMAC-based integrity verification and ensures that ILogs are not maliciously modified and are up-to-date with in-enclave contents. Furthermore, it prevents forking [57] and rollback [72] attacks by ensuring that only the latest version of a sealed enclave is used (Section 4.5) (via the EVC broadcast to all users during enclave sealing).

While SACK uses BFs (Section 4.3) to mitigate freshness verification overhead for reads via ILogs, the BFs will not introduce any security concern. As SACK only resets BFs after VLog GC is done (Section 4.4), all invalid KV pairs are removed and the in-enclave metadata are updated. Even though the KV pairs are later updated, no old KV pairs can pass integrity checks, and the updated KV pairs will be treated as newly inserted KV pairs.

Dynamic access control. SACK securely manages users' attributes and all access rights defined by policies inside the enclave. Even if an adversary has access to the DEK directory or knows the attributes of a compromised user, SACK prohibits the adversary from feasibly decrypting ABE-protected DEKs or modifying any policy to grant unauthorized access to other non-compromised users. Note that a malicious administrator can assign unauthorized attributes to a user to gain illicit access to KV pairs, yet it can still be mitigated by requiring multiple administrators to manage access control and approve changes to attributes or policies [58].

4.7 Implementation Details

We implement SACK as a middleware system that runs atop general KV stores and supports both SGXv1 [44] and SGXv2 [46]. We adopt Intel SGX as Intel currently occupies over 60% of the server CPU market [73] and provides the most comprehensive encryption, integrity, and freshness guarantees [61]. The implementation can be adapted to other TEEs with a similar programming model (e.g., ARM TrustZone [88] and RISCV-V Penglai [25]) that separate trusted and untrusted components and export function calls to access them securely.

Our SACK prototype is written in C++ and consists of 19 K lines of code. It leverages third-party libraries for major operations, including OpenABE-1.0 [99] for ABE, SGX SDK 2.21 [50] and the corresponding SGX SSL [48] for SGX enclave management, and OpenSSL 1.1.1u [31] for cryptographic operations. It implements HMACs based on SHA3-384 (as in TWEEZER [52]), and symmetric-key encryption based on AES-256 in cipher feedback mode. It uses xxHash [18] as the hash function for BFs (Section 4.3) and indexes the HMACs of per-ILog insert BFs by ILog IDs via a hash map in the enclave. We adopt standard performance optimization techniques, such as parallelizing I/O operations with multi-threading and batching multiple reads to a VLog during scans. SACK processes incoming KV operations regardless of whether they originate from a single or multiple users, so as to ensure proper functionality in multi-user scenarios.

SACK caches the matched policy IDs inside the EPC for active users. If the EPC is full, some cached data can be securely swapped out via the SGX page swapping mechanism, which is natively supported by SGX without manual intervention.

SACK can be deployed atop general KV stores. Our evaluation focuses on RocksDB [28] (Section 5), yet our current prototype also supports other KV stores, including BerkeleyDB [70] with B-tree-based and hash-based storage backends, B^ϵ-tree-based SplinterDB [21], and the LSM-tree-based LevelDB [36]. Our evaluation findings also hold for other KV stores.

5 EVALUATION

We evaluate SACK's performance and resource usage. As there is no existing SGX-based KV store that supports dynamic access control for more fair comparisons, we mainly compare SACK with TWEEZER [52], a state-of-the-art SGX-based KV store that provides confidentiality, integrity, and freshness guarantees but lacks dynamic access control and general applicability. TWEEZER extends RocksDB [28], which manages KV pairs in the LSM-tree. It protects KV pairs with symmetric-key encryption and per-KV-pair HMACs. It applies KV separation within each data block (the storage unit of an SSTable) and reorganizes key blocks and value blocks for fine-grained encryption and authentication, yet it still keeps complete KV pairs within the LSM-tree. It needs to perform LSM-tree compaction and maintain LSM-tree metadata inside the enclave, so it cannot be directly applied to general KV stores without re-engineering their index structures. In contrast, SACK separates value storage from the LSM-tree, significantly reducing the LSM-tree size and achieving strong security guarantees without modifying the index structure (Section 4.1). This design ensures compatibility with various KV stores (e.g., hash-based KV stores) (Section 4.7). We also compare SACK against ABE in Exp#2 and Exp#6, where ABE's results are referenced from Figure 2 (Section 2.2) due to its significantly higher overhead than SACK and TWEEZER.

5.1 Methodology

Testbed. SACK is compatible with both SGXv1 and SGXv2, yet the open-source TWEEZER prototype supports SGXv1 only, but not SGXv2, as reported by TWEEZER's authors [65]. Thus, for fair comparisons with TWEEZER, we mainly conduct our evaluation on an SGXv1 server. The server has a six-core Intel Core i5-8400 2.8 GHz CPU, 32 GiB RAM, 128 MiB EPC memory, and a 1 TiB PCIe 3.0 NVMe SSD. It runs Ubuntu 18.04.6 LTS with Linux kernel

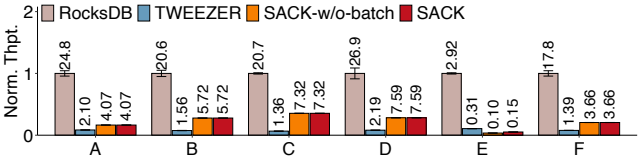
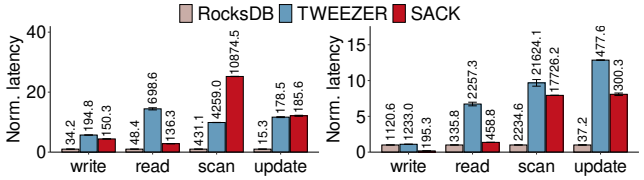


Figure 9: Exp#1 (Performance under YCSB core workloads). We plot the throughput normalized by RocksDB's. We also report the absolute throughput in KOPS above each bar.



(a) Normalized average latency (b) Normalized P99 tail latency

Figure 10: Exp#2 (Performance of KV operations). We plot latency results normalized by RocksDB's. We also report the latencies in microseconds above the bars.

4.15.0-213-generic, as required by TWEEZER. We run TWEEZER on SCONE [5] version 5.4.0[38].

Default configurations. We run SACK atop RocksDB v6.14.5 [28] as in TWEEZER for fair comparisons. We follow the default settings of RocksDB and TWEEZER. For SACK, we configure the capacities of a VLog and an ILog to store at most 1,000 KV pairs and 1,000 KV-metas, respectively. We set eight VLog buffers. We configure the BFs with three hash functions each. We configure the per-ILog insert BF size as 1.5 KiB, which yields a theoretical false positive rate of less than 1% for 1,000 KV-metas per ILog. Similarly, we set the update BF size as 16 MiB, which provides a theoretical false positive rate of less than 1% for 10 M unique updates. In addition, we cache per-ILog insert BFs with a 1 MiB in-enclave LRU cache.

We focus on a single attribute and a single policy for access control to isolate system variables and incrementally validate core mechanisms. We also evaluate the impact of a varying number of attributes and policies in Exp#7.

Workloads. We generate workloads using YCSB [22]. By default, in each experiment, we first load 64 M 1-KiB KV pairs composed of 24-byte keys and 1000-byte values into storage. We then generate 5 M requests for each workload. The requests follow a Zipf distribution with a Zipfian constant of 0.99 (default in YCSB). We report the average results across five runs, with error bars representing 95% confidence intervals based on the student's t-distribution (we omit error bars from line charts for brevity).

Additional experiments. In the technical report [80], we present additional evaluation results, including the effects of in-enclave LRU cache sizes for per-ILog insert BFs, value sizes, ILog and VLog capacities, database volume, and SSTable BFs. We also evaluate the compatibility of SACK with SGXv2 and various KV stores.

5.2 Results

Exp#1 (Performance under YCSB core workloads). We compare RocksDB, TWEEZER, and SACK using six YCSB core workloads [22]: A (50% reads, 50% writes), B (95% reads, 5% writes), C (100% reads), D (95% reads, 5% writes), E (95% scans, 5% writes), and F (50% reads, 50% read-modify-writes). All workloads (except D) follow a Zipf distribution, while Workload D reads the most recently written KV pairs.

Figure 9 shows that SACK achieves throughput gains of 1.9-5.4× over TWEEZER across all workloads except Workload E. SACK's throughput gains are mainly due to KV separation and reduction in EPC paging for reads (Exp#5), and increase with the read ratio (e.g., 1.9× in A and 5.4× in C). However, SACK has 51.6% lower throughput than TWEEZER in Workload E due to KV separation, as it requires extra reads from multiple VLogs (the scan overhead is also observed in [60]). Our batched-read VLog optimization for scan operations (Section 4.7) mitigates this overhead and increases Workload E's throughput by 50%, from 0.10 KOPS (i.e., SACK-w/o-batch) to 0.15 KOPS (i.e., SACK). Overall, the average throughput drop of SACK over RocksDB (without security protection) is 6.7×, while that of TWEEZER is 12.5×.

Exp#2 (Performance of KV operations). We evaluate the average and 99th-percentile latencies for specific KV operations, including reads, writes, scans, and updates. We first load 64 M 1-KiB KV pairs, followed by issuing 5 M requests for each type of KV operations under a Zipf distribution with a Zipfian constant of 0.99. For scans, the scan length is 100 KV pairs as in YCSB Workload E.

Figure 10 shows that compared with TWEEZER, SACK reduces average write and read latencies by 22.8% and 80.5%, respectively, but increases average scan and update latencies by 155.3% and 4.0%, respectively (Figure 10(a)). The latency increase stems from reads to separate VLogs and from GC overhead. Compared with RocksDB, SACK incurs 2.8-25.2× higher average latencies due to added cryptographic features. The latency increase is the most in scans, consistent with Workload E in Exp#1. With the batched-read VLog optimization (Section 4.7), SACK's average scan latency decreases from 13.8 ms (not shown in the figure) to 10.9 ms.

For 99th-percentile tail latencies (Figure 10(b)), SACK consistently maintains lower tail latencies than TWEEZER across all KV operations, with an average reduction of 54.7%. Notably, although SACK has higher average scan latencies, it has lower tail latencies for scans due to 19.9× EPC paging reduction (Exp#5). Disabling the batched-read VLog optimization increases SACK's scan tail latency by 80.8% to 32.0 ms (not shown in the figure). Compared with RocksDB, SACK has 82.6% lower tail latencies in writes as KV separation reduces the LSM-tree size, while it has higher tail latencies in reads, scans, and updates. Compared with ABE (Figure 2 in Section 2.2), SACK reduces average write and read latencies by 79.9% and 92.8%, respectively, under a single attribute and policy, as it employs lightweight symmetric-key encryption instead of public-key-based ABE.

Exp#3 (Performance breakdown). We provide a performance breakdown for multiple steps of writes, reads, and GC. We report the time of processing 1 MiB of KV pairs for each step based on the workloads in Exp#2. Table 1 shows the breakdown results. For writes, the most time consuming step is ILog updates, which account for 46.0% of total write time. The reason is that the KV-metas of written KV pairs may reside in different ILogs, and the enclave needs to recalculate and update HMACs after appending KV-metas to ILogs. For reads, the most time-consuming step is fetching KV-metas from ILogs, where the enclave retrieves the whole ILog for

Table 1: Exp#3 (Performance breakdown). We show the average results (in milliseconds/MiB) and 95% confidence intervals for different steps of writes and reads.

Steps		Writes
Generate KV-meta		19.66±0.04
Encrypt KV pairs		23.08±0.09
Build Merkle tree		3.48±0.02
Update BFs		32.15±0.13
Flush VLogs		0.72±0.10
Update ILogs		70.81±0.14
Update KV store		3.99±0.07
Steps		**Reads**
Query BFs		4.78±0.04
Fetch KV-meta	Query KV store	33.22±0.55
	Query ILogs	477.42±13.61
Read value from VLog		38.16±0.08
Access control checking & Decryption		16.46±0.07
Verify integrity and freshness		42.56±0.08
Steps		**Scans**
Query ILogs		34.59±0.10
Read values from VLogs		55.35±0.21
Access control checking & Decryption		10.06±0.02
Verify integrity and freshness		15.66±0.04
Steps		**GC**
GC of ILogs		6.43±0.08
GC of VLogs		45.16±1.55

each KV-meta and performs HMAC verification. Nevertheless, the BFs significantly save access to ILogs in reads (Exp#8).

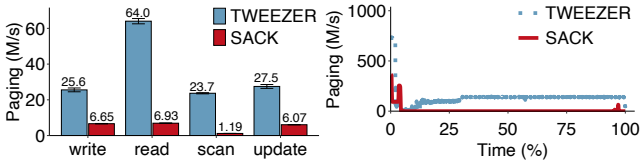
In contrast, scans read 100 sequential KV pairs at a time (0.1 MiB per scan), while reads fetch one KV pair (1 KiB) per operation. Thus, the most time-consuming step for scans is reading values scattered across multiple VLogs, even for sequential keys. The overhead is exacerbated by the large data transfers between the enclave and untrusted memory, which amplify context-switch ovrehead due to the limited EPC size. On the other hand, querying ILogs is faster in scans than in reads, since ILogs store contiguous KV-metas. Also, the access control checking, decryption, and integrity verification are faster in scans than in reads, as their processing can be batched for multiple values.

For GC, ILog GC is efficient, while VLog GC is more time-consuming as it needs to verify data integrity and update the KV-metas of valid KV pairs in both ILogs and the KV store. VLog GC runs in system idle periods to limit its overhead to normal operations (Section 4.4).

Exp#4 (Crash recovery performance). We evaluate SACK's crash recovery performance. We load 64 M 1-KiB KV pairs into SACK and issue 5 M writes as in Exp#2. We crash SACK after issuing about 2.5 M writes using the `kill -9 processID` command. We restart SACK to initiate recovery and measure the times of various recovery steps. Table 2 shows SACK's crash recovery performance. The enclave unsealing, ILog recovery, and VLog recovery account for 1.3%, 38.6%, and 60.1% of total recovery time. During VLog recovery, ILog updates are the most time-consuming step as they involve HMAC recalculations and KV-meta rewriting.

Table 2: Exp#4 (Crash recovery performance). We show average latencies and 95% confidence intervals for different steps.

Steps		Time
Unseal enclave		6.68 ± 0.06 s
Recover ILogs		194.72 ± 0.66 s
Recover VLogs	Fetch KV pairs	15.73 ± 2.86 s
	Decryption	31.32 ± 3.60 s
	Verify KV pairs	45.42 ± 5.23 s
	Update BFs	40.94 ± 4.94 s
	Update ILogs	101.73 ± 14.06 s
	Update KV store	10.32 ± 1.07 s
Recovery time		504.69 ± 38.64 s



(a) Average paging rates (b) Paging rate in reads

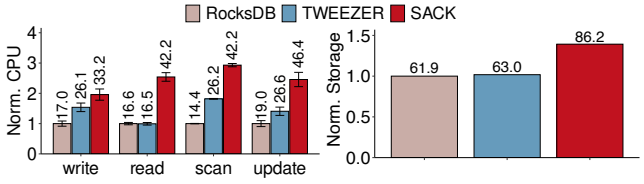
Figure 11: Exp#5 (Enclave usage).

Exp#5 (Enclave usage). We analyze SACK's EPC usage, assuming a scenario of 64 M 1-KiB KV pairs, 1,000 users with 16 attributes each, and 2^{16} policies. The EPC usage is calculated based on the following persistent data structures:

- *User-attribute table:* We assume 4 bytes for the user ID and 16 bytes for an attribute string. We do not consider the cached PIDs and DEKs. Thus, the user-attribute table size is (4 + 16 × 16 (attributes)) × 1,000 (users) ≈ 0.25 MiB.
- *Policy-metadata table:* We assume 256 bytes for a policy string (using 16 attributes), 4 bytes for a PID, 48 bytes for a DEK HMAC, 4 bytes for a VLog ID, and 48 bytes for a Merkle tree root, and each policy corresponds to one VLog. The policy-metadata table size is (256 + 4 + 48 + 4 + 48) (bytes) × 2^{16} (policies) = 22.5 MiB.
- *Prefix tree:* We assume 4 bytes for each ILog ID, 4 bytes for each size entry, 4 bytes for the number of KV-metas, and 48 bytes for HMAC in both sorted and unsorted parts. Thus, each leaf node has 116 bytes. Also, each non-leaf node has 24 bytes (three pointers for referencing the parent node and two child nodes). Assuming 64 M KV pairs and 1,000 KV-metas in each ILog, there are 64 K ILogs, which imply 64 K leaf nodes and 64 K non-leaf nodes for a binary prefix tree. The prefix tree size is about 8.5 MiB.
- *SACK BFs:* We assume 17 MiB (1 MiB for the in-enclave LRU cache for per-ILog insert BFs and 16 MiB for the update BF). Additionally, HMACs (48 bytes each) for per-ILog insert BFs are indexed by 64 K ILog IDs (4 bytes each) and account for (4 + 48) (bytes) × 64 K ≈ 3.3 MiB. The total size is 20.3 MiB.

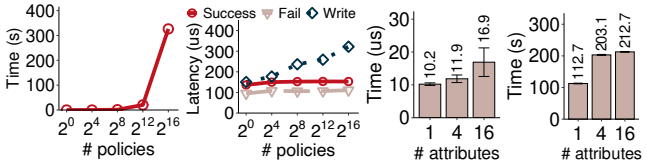
We do not consider skip-lists for indexing KV pairs in VLog buffers (Section 4.3) as they are temporary (with less than 1 MiB of EPC). Thus, the total EPC size for SACK's persistent data structures is about 51.6 MiB.

We also evaluate the EPC paging overheads of TWEEZER and SACK. We use `sgxtop` [30] to measure the page rate in Exp#2 (a higher paging rate means higher overhead). Figure 11 shows that SACK reduces the EPC paging rate of TWEEZER by 74.0%,



(a) CPU usage (b) Storage size

Figure 12: Exp#6 (Server resource usage). We report results normalized by RocksDB's. We report the absolute CPU usage (%) and storage size (GiB) above the bars.



(a) Initialization (b) Avg. latency (c) Updating (d) Rekeying

Figure 13: Exp#7 (Impact of the number of policies). In figure (b), "Success" and "Fail" denote whether a read successfully retrieve a KV pair or is denied due to insufficient access rights, while "Write" denotes write operations.

89.2%, 95.0%, and 77.9% in writes, reads, scans, and updates, respectively (Figure 11(a)), as it maintains small-size data structures in the enclave, while TWEEZER maintains SSTables' metadata in the enclave for LSM-tree management. Figure 11(b) examines the paging rates over the time of issuing reads in TWEEZER and SACK. SACK incurs paging mainly at the beginning and end of reads, which correspond to the initialization of in-enclave data structures and the enclave sealing when all reads are completed for crash consistency (Section 4.5), respectively. In contrast, TWEEZER performs paging throughout reads to manage SSTables' metadata.

Exp#6 (Server resource usage). We compare the CPU usage and storage sizes of RocksDB, TWEEZER, and SACK in our server using the workloads in Exp#2. We disable compression in all systems (the default setting in RocksDB). We collect CPU usage using the Linux system `proc` files every 1 s. We also collect storage sizes using the `du -s --bytes` command at the end of evaluation.

Figure 12(a) shows that SACK incurs 95.3-193.1% higher CPU usage than RocksDB, as it performs additional cryptographic operations. SACK also incurs higher CPU usage than TWEEZER due to the extra operations for enforcing access control and managing VLogs and ILogs.

Figure 12(b) shows that SACK incurs 39.3% more storage space than RocksDB, mainly due to (i) KV-metas stored in the KV store and ILogs and (ii) Merkle trees in VLogs. Specifically, SACK's storage comprises: (i) enclave states and DEKs (0.04 GiB), (ii) persistent per-ILog insert BFs (0.23 GiB), (iii) ILogs (8.84 GiB), (iv) underlying KV store (8.95 GiB), and (v) VLogs (68.2 GiB) (not shown in the figure). In contrast, TWEEZER shows only 1.8% more storage size than RocksDB. Our observed storage overhead is much less than reported in TWEEZER's paper as it enables compression for plain KV pairs in RocksDB [52], while we disable compression. Compared to ABE (Section 2.2), SACK reduces storage overhead from 83.5% to 39.3%, as it removes the embedding of policies within KV pairs.

Exp#7 (Impact of the number of policies). We evaluate SACK's performance by varying the number of policies. We define 16 attributes (`attr=i`, where $1 \leq i \leq 16$) and generate up to 2^{16} policies by randomly selecting different subsets of the 16 attributes. Since practical ABAC systems typically limit the number of policies (e.g., fewer than 10,000) for efficient access control [41, 92], the range of policies considered here reflects real-world use cases. We load 64 M 1-KiB KV pairs and evenly assign them to different policies. We focus on a single user with a single attribute (`attr=1`) to analyze its performance when accessing a KV store that contains KV pairs with mixed attributes and policies. We let the user issue 5 M reads or writes.

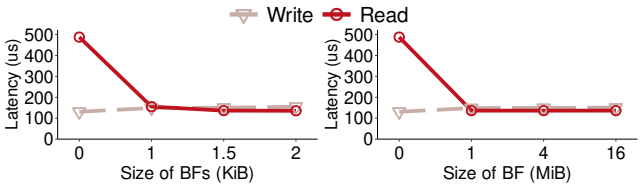
Figure 13(a) shows the initialization time for a different number of policies, starting from when a user connects to the system until the enclave caches all associated PIDs and DEKs in the user-attribute table. The initialization time increases from 8 ms to 326.4 s when the number of policies increases from 1 to 2^{16}. SACK decrypts the DEK by ABE for each policy that contains at least one of the user's attributes, so as to cache the DEK in the user-attribute table. Note that this is only a one-time process when the user first connects to SACK or when its attributes are updated (i.e., Case 1 in Section 4.3), and it does not affect subsequent KV operations.

Figure 13(b) shows the average write latency and the average read latencies for the scenarios when the user has the access rights to successfully read the KV pairs and otherwise. As the number of policies increases from one to 2^{16}, the latency for successful reads increases from 136.3 μs to 152.9 μs, and that for failed reads increases from 95.1 μs to 111.1 μs. The latency for failed reads is 29.3% lower than successful reads, as it does not need to fetch DEKs and decrypt KV pairs. As the number of policies increases, the average write latency increases from 150.3 μs to 321.8 μs. The reason is that the number of available VLog buffers are limited, so VLog buffers are flushed more frequently as more policies are involved, leading to more context switches and disk I/Os. Nevertheless, the latency increase remains acceptable, and the write latency remains much lower than in ABE (Figure 2 in Section 2.2).

With 2^{16} policies, SACK maintains strong read efficiency: its average latency for successful reads remains 78.1% lower than TWEEZER's (Exp#2), while its average write latency is 65.2% higher.

Figure 13(c) shows the time to update a varying number of attributes of a user (i.e., Case 1 in Section 4.3) in the user-attribute table. When the user's number of attributes increases from one to 16, the updating time slightly increases from 10.2 μs to 16.9 μs. Note that after updating attributes, the enclave needs to re-initialize the user-attribute table to cache the DEKs for the new attributes before any KV operations, and the overhead is analyzed in Figure 13(a).

Figure 13(d) shows the rekeying time when some policies are updated (i.e., Case 2 in Section 4.3), in which DEKs with outdated policies are decrypted and re-encrypted with new policies by ABE. Here, we consider the cases where one, four, and 16 attributes are updated, and renew the affected policies that involve at least one of the updated attributes (the affected policies account for 48.8%, 91.6%, and 100% of all policies, respectively). We observe that the rekeying time increases from 112.7 s (for one updated attribute) to 212.7 s (for all 16 updated attributes). Note that the rekeying time does not depend on the storage size of KV pairs, and renewing a single DEK for a policy takes only 3.4 ms.



(a) Varying sizes of IBF, UBF=16 MiB, Cache=1 MiB

(b) Varying sizes of UBF, IBF=1.5 KiB, Cache=1 MiB

Figure 14: Exp#8 (Impact of the sizes of BFs). We vary the sizes of per-ILog insert BFs (IBF) and update BF (UBF), while fixing the in-enclave LRU cache for per-ILog insert BFs (Cache) as 1 MiB. A zero size means BFs are disabled.

Exp#8 (Impact of the sizes of SACK BFs). We evaluate various sizes of per-ILog insert BFs and update BF (Section 4.3) on SACK's read performance. We load 64 M 1-KiB KV pairs and issue 5 M reads. The SSTable BFs of RocksDB are disabled by default (as in RocksDB and TWEEZER). Note that we have also evaluated the effects of various in-enclave LRU cache sizes for per-ILog insert BFs (Section 4.3) and SSTable BFs in the technical report [80].

Figure 14(a) shows that as the per-ILog insert BF size increases, the write latency increases from 130.9 μs (without BFs) to 154.9 μs (2 KiB per-ILog insert BF) as fewer per-ILog insert BFs are cached, while the read latency decreases by 72.2% from 487.5 μs (without BFs) to 135.6 μs (2 KiB per-ILog insert BF) as larger per-ILog insert BFs reduce false positives and ILog reads. Notably, even without BFs, SACK's read latency is still 30.2% lower than TWEEZER (Exp#2).

Figure 14(b) shows a similar trend for the update BF, where the write latency increases from 130.9 μs (without BFs) to 150.3 μs (16 MiB update BF), while the read latency decreases to an average of 136.3 μs (across 1 MiB, 4 MiB, and 16 MiB update BF). The read latency is stable as it only queries the update BF (Section 4.3), while the overall false positive rate mainly depends on the per-ILog insert BF size as the read-only workload has no updates.

To maintain SACK's performance, we recommend users to configure the per-ILog insert BF size according to the ILog capacity and a fixed false positive rate (e.g., 1%) (computed following [12]). For the update BF, we provide two practical strategies: (i) fix the update BF size and adjust the frequency of VLog GC (which resets the update BF) based on the observed update rate; or (ii) fix the VLog GC frequency and adjust the update BF size based on the expected number of updates within the time window between two VLog GC cycles. For example, a 16 MiB update BF can support up to 10.9 M updates with a 1% false positive rate, while increasing its size to 32 MiB doubles the capacity to 21.7 M updates.

6 RELATED WORK

Software-based access control in KV stores. Traditional KV stores implement cryptography-based access control in software [19, 20, 32, 34]. Attribute-based group key management (AB-GMK) [66] lowers the overhead of ABE-based ABAC, and has been used for access management for user certificates [90] and enforcing access control in relational databases [83]. It uses attributes to manage group membership and control access to DEKs, which are used to protect data with symmetric encryption. While SACK adopts a similar approach to protect DEKs with ABE and encrypt data symmetrically, it targets persistent KV stores, decouples policy enforcement from data storage, and leverages SGX to achieve confidentiality, integrity, freshness, and dynamic ABAC. In contrast, AB-GMK offers access control and data confidentiality, but does not natively guarantee integrity or freshness.

Shielded execution for relational databases. Shielded execution has been extensively studied in relational databases, both in academia [9, 27, 59, 76, 95, 96] and industry [3, 16, 101]. TrustedDB [9], ObliDB [27], EnclaveDB [76], SQL Server [3], GaussDB [101], and ApsaraDB [16] execute SQL statements inside an enclave with privacy protection. VeDB [96] uses a native monotonic counter of TEEs and a trusted timestamp for verifiable endorsement. HEDB [59] protects sensitive data against smuggling attacks. SecuDB [95] is a multi-granularity, privacy-preserving, and tamper-resistant relational database within Intel TDX. Our work targets KV stores, with higher performance and scalability than relational databases.

Shielded execution for persistent KV stores. Shielded execution has been applied to in-memory KV stores for secure and fast data access [7, 14, 53, 94, 97], while our work targets persistent KV stores. Concerto [4] stores incremental hashes of reads and writes in the enclave to verify the integrity of KV pairs. VeritasDB [86] provides a secure proxy that uses Merkle B+-trees and SGX to verify data integrity of server responses before forwarding them to clients. TEE-KV [54] applies SGX to blockchains to store transactions in external KV stores. SPEICHER [8] uses SGX in LSM-tree-based KV stores and designs asynchronous trusted monotonic counters to ensure confidentiality, integrity, and freshness. TWEEZER [52] improves the I/O performance of SPEICHER with per-KV-pair message authentication for fine-grained integrity protection. SACK further supports dynamic access control with SGX and outperforms TWEEZER via KV separation (Section 5.2). A closely related work is Pesos [55], which uses SGX and trusted storage (Kinetic Object Storage) to achieve user-defined policy-based access control. In contrast, SACK does not rely on trusted storage. It also supports policy renewal and is compatible with general KV stores.

7 CONCLUSIONS

SACK is a shielded framework that enables secure and efficient dynamic access control in cloud-hosted KV stores. It decouples access control and data management, by managing attributes and policies inside an SGX enclave and leveraging KV separation for secure and efficient data management. Experiments demonstrate SACK's performance efficiency in KV operations and low overhead in access control. Future work includes enhancing SACK with faster scans, recovery, and policy renewal mechanisms, and supporting distributed KV stores with low performance overhead.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (62522204 and 62561160158), Open Research Fund of State Key Laboratory of Internet Architecture under Grant (HLW2025ZD04), Innovation and Technology Commission of Hong Kong (GHX/076/20), Research Grants Council of Hong Kong (GRF 14214622), and Research Matching Grant Scheme. The corresponding author is Jingwei Li.

REFERENCES

[1] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. OBFUSCURO: A commodity obfuscation engine on Intel SGX. In *Proceedings of Network and Distributed System Security Symposium (NDSS'19)*. 1--15.

[2] AMD. 2016. AMD secure encrypted virtualization (SEV). Retrieved January 30, 2026 from https://developer.amd.com/sev/.

[3] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL database always encrypted. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data (SIGMOD'20)*. 1511--1525.

[4] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*. 251--266.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, and Mark Stillwell. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 689--703.

[6] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. 2008. Multi-tenant databases for software as a service: schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD'08)*. 1195--1206.

[7] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Vijay Nagarajan, and Pramod Bhatotia. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*. 65--79.

[8] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 173--190.

[9] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD'11)*. 205--216.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 164--177.

[11] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE symposium on security and privacy (S&P'07)*. 321--334.

[12] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422--426.

[13] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. 2018. Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P'18)*. 178--194.

[14] Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. 2019. EnclaveCache: A secure and scalable key-value cache in multi-tenant clouds using Intel SGX. In *Proceedings of the 20th International Middleware Conference (Middleware'19)*. 14--27.

[15] William R Claycomb and Alex Nicoll. 2012. Insider Threats to Cloud Computing: Directions for New Research Challenges. In *Proceedings of the 2012 IEEE 36th annual computer software and applications conference (COMPSAC'12)*. 387--394.

[16] Alibaba Cloud. 2021. ApsaraDB RDS for PostgreSQL - Supports Fully Encrypted Databases Based on Intel SGX. Retrieved January 30, 2026 from https://www.alibabacloud.com/en/news/product/supports-fully-encrypted-databases-based-on-intel-sgx-g5s?_p_lc=1.

[17] Alibaba Cloud. 2025. Build an SGX confidential computing environment. Retrieved January 30, 2026 from https://www.alibabacloud.com/help/en/ecs/user-guide/build-an-sgx-encrypted-computing-environment.

[18] Yann Collet. 2020. xxHash v0.7.3. Retrieved January 30, 2026 from https://github.com/Cyan4973/xxHash/releases/tag/v0.7.3.

[19] Pietro Colombo and Elena Ferrari. 2016. Towards virtual private NoSQL datastores. In *Proceedings of the 2016 IEEE 32nd International Conference on Data Engineering (ICDE'16)*. 193--204.

[20] Pietro Colombo and Elena Ferrari. 2017. Towards a unifying attribute based access control approach for NoSQL datastores. In *Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE'17)*. 709--720.

[21] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: closing the bandwidth gap for NVMe Key-Value stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. 49--63.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SOCC'10)*. 143--154.

[23] Morris Dworkin. 2001. Recommendation for Block Cipher Modes of Operation. Retrieved January 30, 2026 from https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38a.pdf.

[24] Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC'22)*. 1037--1048.

[25] Penglai Enclave. 2026. Penglai: Scalable trusted execution environment for RISC-V. Retrieved January 30, 2026 from https://penglai-enclave.systems/.

[26] Robert Escriva and Jeremy Fitzhardinge. 2016. HyperLevelDB. Retrieved January 30, 2026 from https://github.com/rescrv/HyperLevelDB.

[27] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: oblivious query processing for secure databases. *Proceedings of the VLDB Endowment* 13, 2 (2019), 169–183.

[28] Facebook. 2020. RocksDB v6.14.5. Retrieved January 30, 2026 from https://github.com/facebook/rocksdb/releases/tag/v6.14.5.

[29] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. 2001. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security* 4, 3 (2001), 224--274.

[30] Fortanix. 2020. sgxtop. Retrieved January 30, 2026 from https://github.com/fortanix/sgxtop.

[31] OpenSSL Software Foundation. 2023. OpenSSL 1.1.1u. Retrieved January 30, 2026 from https://github.com/openssl/openssl/releases/tag/OpenSSL_1_1_1u.

[32] The Apache Software Foundation. 2011. Apache Accumulo. Retrieved January 30, 2026 from https://accumulo.apache.org.

[33] Juliette Galletti. 2024. The CLOUD Act: Implications of Storing Data with Cloud Providers - What Does it Mean for the Partner Ecosystem? Retrieved January 30, 2026 from https://blog.ovhcloud.com/cloud-data-act/.

[34] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. 2010. Comet: An Active Distributed Key-Value Store. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*. 1--14.

[35] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431--473.

[36] Google. 2021. LevelDB v1.23. Retrieved January 30, 2026 from https://github.com/google/leveldb/releases/tag/1.23.

[37] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. 2006. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS'06)*. 89--98.

[38] Systems Engineering group at TU Dresden. 2026. SCONE: Ubuntu18.04-scone5.4.0. Retrieved January 30, 2026 from https://gitlab.scontain.com.

[39] Andreas Grünbacher. 2003. POSIX access control lists on linux. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC'03)*. 259--272.

[40] Danny Harnik, Eliad Tsfadia, Doron Chen, and Ronen Kat. 2018. Securing the storage data path with SGX enclaves. *https://arxiv.org/abs/1806.10883* (2018).

[41] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Arthur R. Friedman, Alan J. Lang, Margaret M. Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. 2013. Guide to attribute based access control (ABAC) definition and considerations (draft). *NIST special publication* 800, 162 (2013), 1--54.

[42] Vincent C. Hu, D. Richard Kuhn, David F. Ferraiolo, and Jeffrey Voas. 2015. Attribute-based access control. *Computer* 48, 2 (2015), 85--88.

[43] Junbeom Hur and Dong Kun Noh. 2010. Attribute-based access control with efficient revocation in data outsourcing systems. *IEEE Transactions on Parallel and Distributed Systems* 22, 7 (2010), 1214--1221.

[44] Intel. 2020. 10th generation intel core processor families datasheet volume 1. Retrieved January 30, 2026 from https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/10th-gen-core-families-datasheet-vol-1-datasheet.pdf.

[45] Intel. 2022. 2022.1 IPU - Intel SGX Advisory. Retrieved January 30, 2026 from https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00614.html.

[46] Intel. 2023. 5th generation intel xeon processor datasheet. Retrieved January 30, 2026 from https://ark.intel.com/content/www/us/en/ark/products/237261/intel-xeon-platinum-8592-processor-320m-cache-1-9-ghz.html.

[47] Intel. 2023. Architecture Specification: Intel Trust Domain Extensions (Intel TDX) Module. Retrieved January 30, 2026 from https://cdrdv2-public.intel.com/733568/tdx-module-1.0-public-spec-344425005.pdf.

[48] Intel. 2023. Intel SGX SSL. Retrieved January 30, 2026 from https://github.com/intel/intel-sgx-ssl/releases/tag/lin_2.21_1.1.1u.

[49] Intel. 2026. Intel SGX Product Offerings. Retrieved January 30, 2026 from https://www.intel.com/content/www/us/en/architecture-and-technology/sgx-product-offerings.html.

[50] Intel. 2026. Intel Software Guard Extensions SDK. Retrieved January 30, 2026 from https://software.intel.com/en-us/sgx/sdk.

[51] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. 2016. Verena: End-to-end integrity protection for web applications. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P'16)*. 895--913.

[52] Igjae Kim, J. Hyun Kim, Minu Chung, Hyungon Moon, and Sam H. Noh. 2022. A Log-Structured Merge Tree-aware Message Authentication Scheme for Persistent Key-Value Stores. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*. 363--380.

[53] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded in-memory key-value storage with SGX. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*. 1--15.

[54] Atsushi Koshiba, Ying Yan, Zhongxin Guo, Mitaro Namiki, and Lidong Zhou. 2018. TEE-KV: Secure Immutable Key-Value Store for Trusted Execution Environments. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'18)*. 535--535.

[55] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy enhanced secure object store. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*. 1--17.

[56] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. 2024. Gramine-TDX: A lightweight os kernel for confidential VMs. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS'24)*. 4598--4612.

[57] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. 1--14.

[58] Jin Li, Qiong Huang, Xiaofeng Chen, Sherman S.M. Chow, Duncan S. Wong, and Dongqing Xie. 2011. Multi-authority ciphertext-policy attribute-based encryption with accountability. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (CCS'11)*. 386--390.

[59] Mingyu Li, Xuyang Zhao, Le Chen, Cheng Tan, Huorong Li, Sheng Wang, Zeyu Mi, Yubin Xia, Feifei Li, and Haibo Chen. 2023. Encrypted Databases Made Secure Yet Maintainable. In *Proceedings of the 17th USENIX symposium on operating systems design and implementation (OSDI'23)*. 117--133.

[60] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 133--148.

[61] Jämes Ménétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. 2022. Attestation mechanisms for trusted execution environments demystified. In *Proceedings of IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 95--113.

[62] Ralph C. Merkle. 1980. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 IEEE Symposium on Security and Privacy (S&P'80)*. 122--122.

[63] Microsoft. 2022. What is Azure Table Storage? Retrieved January 30, 2026 from https://learn.microsoft.com/en-us/azure/storage/tables/table-storage-overview.

[64] Jelena Mirkovic and Peter Reiher. 2004. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review* 34, 2 (2004), 39--53.

[65] Hyungon Moon. 2024. TWEEZER's known issue with SGXv2. Retrieved January 30, 2026 from https://github.com/cssl-unist/tweezer/issues/5.

[66] Mohamed Nabeel and Elisa Bertino. 2014. Attribute Based Group Key Management. *Trans. Data Privacy* 7, 3 (dec 2014), 309--336.

[67] U.S. Department of Justice. 2019. CLOUD Act. Retrieved January 30, 2026 from https://www.justice.gov/d9/pages/attachments/2019/04/09/cloud_act.pdf.

[68] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *Proceedings of the 2018 Usenix Annual Technical Conference (USENIX ATC'18)*. 227--240.

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

[70] Oracle. 2026. BerkeleyDB. Retrieved January 30, 2026 from https://www.oracle.com/hk/database/technologies/related/berkeleydb.html.

[71] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*. 238--253.

[72] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. 2011. Memoir: Practical state continuity for protected modules. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P'11)*. 379--394.

[73] Andy Patrizio. 2025. Despite the hubbub intel is holding onto server market share. Retrieved January 30, 2026 from https://www.networkworld.com/article/4040551/despite-the-hubbub-intel-is-holding-onto-server-market-share.html.

[74] Matthew Pirretti, Patrick Traynor, Patrick McDaniel, and Brent Waters. 2006. Secure attribute-based systems. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS'06)*. 99--112.

[75] Krešimir Popović and Željko Hocenski. 2010. Cloud computing security issues and challenges. In *Proceedings of the 33rd international convention mipro (MIPRO'10)*. 344--349.

[76] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P'18)*. 264--278.

[77] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 497--514.

[78] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital Side-Channels through obfuscated execution. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. 431--446.

[79] Anil Rao. 2022. Rising to the challenge—data security with intel confidential computing. Retrieved January 30, 2026 from https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141.

[80] Yanjing Ren, Jingwei Li, and Patrick P. C. Lee. 2026. *SACK: Shielding Dynamic Attribute-based Access Control in Persistent Key-Value Stores*. Technical Report. CUHK. http://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_sack.pdf.

[81] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)*. 199--212.

[82] Amit Sahai and Brent Waters. 2005. Fuzzy identity-based encryption. In *Proceedings of the Annual international conference on the theory and applications of cryptographic techniques (EUROCRYPT'05)*. 457--473.

[83] Muhammad I. Sarfraz, Mohamed Nabeel, Jianneng Cao, and Elisa Bertino. 2015. DBMask: Fine-Grained Access Control on Encrypted Relational Databases. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. 1--11.

[84] Amazon Web Services. 2026. Amazon SimpleDB. Retrieved January 30, 2026 from https://aws.amazon.com/simpledb/.

[85] Ashish Singh and Kakali Chatterjee. 2017. Cloud security issues and challenges: A survey. *Journal of Network and Computer Applications* 79 (2017), 88--115.

[86] Rohit Sinha and Mihai Christodorescu. 2018. VeritasDB: High throughput key-value store with integrity. *Cryptology ePrint Archive* (2018).

[87] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1--26.

[88] ARM Security Technology. 2009. Building a Secure System using TrustZone Technology. Retrieved January 30, 2026 from https://static.docs.arm.com/genc009492/c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

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

[90] Tanya Verma. 2023. Inside Geo Key Manager v2: re-imagining access control for distributed systems. Retrieved January 30, 2026 from https://blog.cloudflare.com/inside-geo-key-manager-v2/.

[91] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. 2010. Privacy-preserving public auditing for data storage security in cloud computing. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'10)*. 1--9.

[92] WSO2. 2026. Improving XACML PDP Performance with Caching Techniques. Retrieved January 30, 2026 from https://is.docs.wso2.com/en/5.9.0/learn/improving-xacml-pdp-performance-with-caching-techniques/.

[93] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. 2020. AC-Key: Adaptive caching for LSM-based Key-Value stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. 603--615.

[94] Fan Yang, Youmin Chen, Youyou Lu, Qing Wang, and Jiwu Shu. 2021. Aria: Tolerating Skewed Workloads in Secure In-memory Key-value Stores. In *Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE'21)*. 1020--1031.

[95] Xinying Yang, Cong Yue, Wenhui Zhang, Yang Liu, Beng Chin Ooi, and Jianjun Chen. 2024. SecuDB: An in-enclave privacy-preserving and tamper-resistant relational database. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3906--3919.

[96] Xinying Yang, Ruide Zhang, Cong Yue, Yang Liu, Beng Chin Ooi, Qun Gao, Yuan Zhang, and Hao Yang. 2023. VeDB: A software and hardware enabled trusted relational database. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1--27.

[97] Junseung You, Kyeongryong Lee, Hyungon Moon, Yeongpil Cho, and Yunheung Paek. 2023. KVSEV: A Secure In-Memory Key-Value Store with Secure

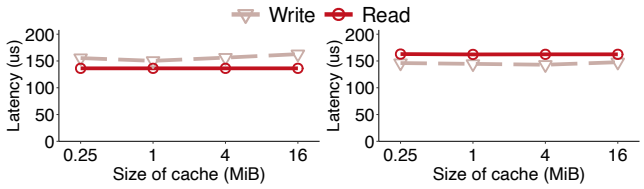
Encrypted Virtualization. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SOCC'23)*. 233--248.

[98] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. 2010. Attribute based data sharing with attribute revocation. In *Proceedings of the 5th ACM symposium on information, computer and communications security (CCS'10)*. 261--270.

[99] Zeutro. 2018. OpenABE v1.0. Retrieved January 30, 2026 from https://github.com/zeutro/openabe/releases/tag/v1.0.

[100] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS'12)*. 305--316.

[101] Jinwei Zhu, Kun Cheng, Jiayang Liu, and Liang Guo. 2021. Full Encryption: An end to end encryption mechanism in GaussDB. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2811--2814.



(a) Varying sizes of cache, IBF=1.5 KiB, UBF=16 MiB

(b) Varying sizes of cache, IBF=1 KiB, UBF=1 MiB

Figure 15: Exp#9 (Impact of the size of the in-enclave LRU cache for per-ILog insert BFs). We vary the sizes of the in-enclave LRU cache for per-ILog insert BFs (Cache). The sizes of the per-ILog insert BFs (IBF) and update BF (UBF) are fixed.

APPENDIX: ADDITIONAL EXPERIMENTS

We evaluate the impact of several parameters on SACK's performance, including: (i) in-enclave LRU cache sizes for per-ILog insert BFs (Exp#9), (ii) value sizes (Exp#10), (iii) VLog and ILog capacities (Exp#11), (iv) database volume (Exp#12), and (v) sizes of SSTable BFs (Exp#13). We conduct the experiments on the SGXv1 testbed (Section 5.1) to examine the performance trade-offs of SACK under varying parameter settings. In addition, we evaluate SACK's compatibility with different SGX versions, including SGXv1 and SGXv2 (Exp#14), as well as its applicability to various KV stores (Exp#15). Our goal is to demonstrate SACK's general applicability to various KV stores that support standard KV interfaces (Section 4.1) and its compatibility with different SGX versions.

In the following experiments, we follow the Zipf distribution with a Zipfian constant of 0.99 for the KV access distribution and a scan length of 100 KV pairs, as in Section 5.2.

Exp#9 (Impact of the size of the in-enclave LRU cache for per-ILog insert BFs). Following Exp#8, we further evaluate various sizes of in-enclave LRU cache for per-ILog insert BFs (Section 4.3) on SACK's read performance. We load 64 M 1-KiB KV pairs and issue 5 M reads. The SSTable BFs of RocksDB are disabled by default (as in RocksDB and TWEEZER).

Figure 15(a) shows that the LRU cache size does not affect the read latency (136.3 μs), as the read-only workload only queries the update BF (Section 4.3) and the overall false positive rate is unaffected. With a larger cache, the write latency first decreases from 155.4 μs (at 0.25 MiB) to 150.3 μs (at 1 MiB) due to fewer HMAC updates for evicted per-ILog insert BFs, and increases to 162.8 μs (at 16 MiB) due to higher EPC paging rates (e.g., from 6.0 K pages/s at 1 MiB to 9.5 K pages/s at 16 MiB; not shown in the figure).

Figure 15(b) shows an extreme case using the smallest per-ILog insert BF (1 KiB) and update BF (1 MiB) sizes under various LRU cache sizes. Similar to the previous case, with a larger cache, the read latency remains stable at an average of 162.5 μs, while the write latency first decreases from 145.6 μs (at 0.25 MiB) to 143.0 μs (at 4 MiB) and increases to 148.8 μs (at 16 MiB). Note that smaller BFs result in a higher false positive rate, leading to increased average read latency compared to the previous case (i.e., 162.5 μs vs. 136.3 μs). Conversely, the average write latency is lower (i.e., 145.6 μs vs. 156.2 μs) due to reduced EPC usage (i.e., a 1 MiB update BF vs. a 16 MiB update BF), which in turn lowers the EPC paging rate.

Exp#10. (Impact of the value size). We evaluate SACK for different value sizes. We load 64 GiB of KV pairs, issue 5 M requests for each type of KV operations (i.e., reads, writes, scans, and updates) as in Exp#2, and measure the average latencies. We adjust the number of KV pairs to keep the same total data volume across all value sizes. Since TWEEZER cannot support a value size larger than 4,000 bytes, we only show results for value sizes up to 4,000 bytes for TWEEZER.

Figure 16(a) shows that the write latencies of RocksDB and TWEEZER increase with the value size. SACK's write latencies initially decrease from 327.1μs to 138.2μs as the value size increases from 250 bytes to 4,000 bytes, and then increase to 602.5μs when the value size is 64,000 bytes. For small value sizes, SACK has high overhead of managing individual KV-metas in different ILogs (e.g., 41.4× RocksDB's and 1.7× TWEEZER's at 250 bytes). For large value sizes, the performance gain of KV separation outweighs the overhead of managing KV-metas. Even with high cryptographic computation overhead (Exp#3), SACK's write latencies are 86.5% lower than RocksDB's at 64,000 bytes.

Figure 16(b) shows that SACK has lower read latencies than TWEEZER for different value sizes due to KV separation and reduced EPC paging (Exp#5). SACK's read latencies initially decrease and then increase as the value size grows. SACK has high read latencies for small value sizes (e.g., 11.8× RocksDB's at 250 bytes) due to the overhead of retrieving individual KV-metas from ILogs for verification. Also, with a fixed total data volume, smaller value sizes lead to more KV pairs and hence higher false positive rates of the insert and update BFs in SACK. Thus, SACK accesses ILogs more often during reads.

Figure 16(c) shows that SACK has higher scan latencies than RocksDB and TWEEZER across all value sizes due to the need to retrieve KV pairs from separate VLogs. Again, SACK's scan latencies initially decrease and then increase as the value size grows.

Figure 16(d) shows that for value sizes of 250 bytes and 1,000 bytes, SACK has 29.4% and 3.9% higher update latencies than TWEEZER, respectively, due to frequent GC in updates. Nevertheless, as the value size increases to 4,000 bytes, SACK reduces TWEEZER's update latency by 23.4% as its performance gain from KV separation outweighs the GC overhead. SACK also has less update overhead than RocksDB as the value size grows; for example, its update latency blowup over RocksDB decreases from 53.4× to 9.6× as the value size increases from 250 bytes to 64,000 bytes.

Exp#11 (Impact of VLog and ILog capacities). We evaluate the impact of varying the capacities of VLogs and ILogs on the average latencies for specific KV operations, including reads, writes, scans, and updates. Following Exp#2 (which evaluates the average and tail latencies on the four different types of KV operations with default VLog and ILog capacities of 1,000 entries), we load 64 M 1-KiB KV pairs and issue 5 M request for each type of KV operation. We vary the VLog and ILog capacities from 250 to 4,000 entries. When varying the VLog (resp. ILog) capacity, we fix the ILog (resp. VLog) capacity at 1,000 entries. To maintain a stable theoretical false positive rate of the insert BFs, we increase the size of each insert BF in accordance with the ILog capacity. Specifically, the size is set to 384, 768, 1,536, 3,072, and 6,144 bytes for ILog capacities of 250, 500, 1,000, 2,000, and 4,000 entries, respectively.

Figure 17 shows the average latency of SACK's different operations with varying VLog and ILog capacities. The read latency increases with VLog capacity (Figure 17(a)) due to the increased

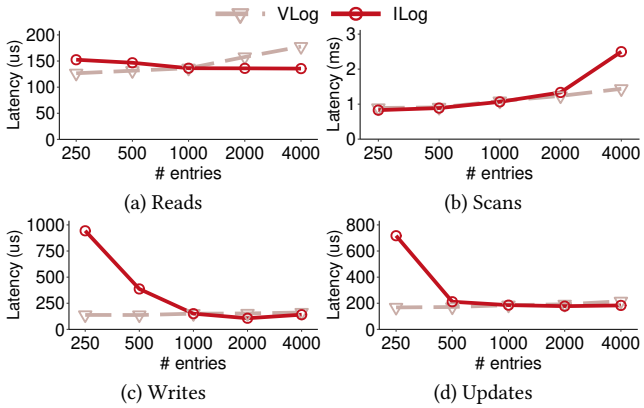


Figure 17: Exp#11 (Impact of VLog and ILog capacities). We plot the average latency of SACK with different capacities of VLog and ILog.

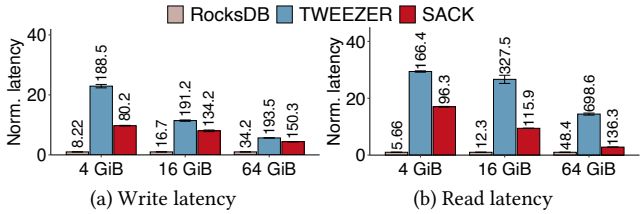


Figure 18: Exp#12 (Impact of database volume). We plot the average latency normalized by RocksDB's. We also report the absolute average latency in μs above each bar.

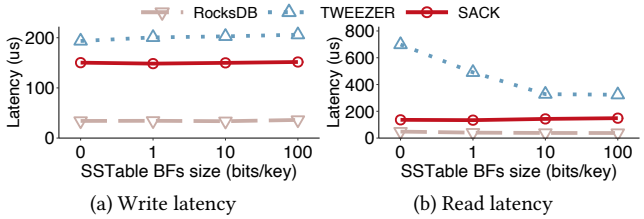


Figure 19: Exp#13 (Impact of SSTable BFs). Note that 0 bits per key means disabling the SSTable BFs.

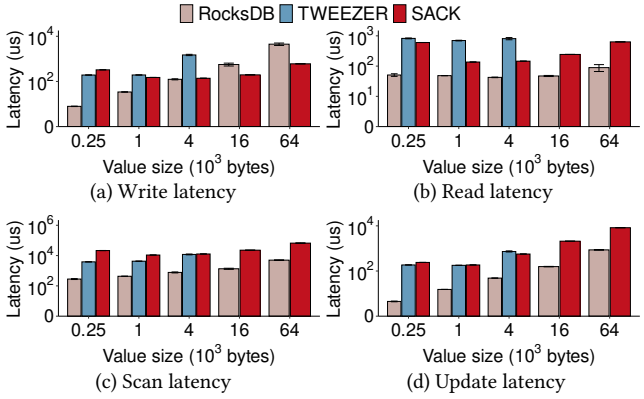


Figure 16: Exp#10 (Impact of the value size). Note that the y-axis is plotted in log scale.

overhead of VLog Merkle tree verification. However, it decreases with ILog capacity (Figure 17(a)). This is because, although the actual proportion of ILog reads remains low (around 0.4%), larger ILogs result in less in-enclave metadata and lower EPC paging rate (e.g., from 38.9 K pages/s for an ILog capacity of 250 entries to 6.7 K pages/s for 4000 entries, not shown in the figure), thereby improving performance. Its scan latencies increase with the capacities of VLogs and ILogs (Figure 17(b)) due to the extra overhead of verifying the Merkle tree of VLogs and scanning ILogs to locate KV-metas, respectively. In addition, the ILog capacity has higher impact on scans than VLog capacity due to the increased overhead of sorting ILog entries, and more useless ILog entries are processed for a scan length of 100 KV pairs (Section 4.3).

The ILog capacity also significantly affects the performance of writes and updates (Figures 17(c) and 17(d), respectively), especially with a small capacity (e.g., 250 entries). This is due to the increased frequency of ILog GC, which blocks KV operations since the ILog cannot be updated during GC (Section 4.4). In contrast, the VLog capacity has a negligible impact compared to the ILog capacity, as larger VLogs only slightly increase the Merkle tree construction and I/O overhead. Our results suggest that 1,000 entries (Section 5.1) for both VLog and ILog are suitable for balancing the performance of reads, writes, scans, and updates.

Exp#12 (Impact of database volume). We evaluate the impact of varying database volumes on the performance of RocksDB, TWEEZER, and SACK. Specifically, we load 4 M, 16 M, and 64 M 1-KiB KV pairs (i.e., approximately 4 GiB, 16 GiB, and 64 GiB of data, respectively) into each system and issue 5 M writes and reads. We measure the average write and read latencies of the systems.

Figure 18 presents the results. As the database volume increases, the average latencies for RocksDB, TWEEZER, and SACK increase accordingly. SACK consistently outperforms TWEEZER across all database volumes, with 36.5% and 62.4% lower average write and read latencies, respectively. This improvement is due to (i) SACK's KV separation, which reduces the LSM-tree size in the underlying RocksDB, and (ii) reduced EPC paging during reads (Exp#5). However, compared to RocksDB, SACK incurs 7.4× and 9.8× higher average write and read latencies across different data volumes due to the additional cryptographic operations. Nevertheless, as the database volume increases, the performance gap between RocksDB and SACK narrows since SACK's cryptographic overhead remains constant, while both systems see performance drops due to the growing indexing overhead.

Exp#13 (Impact of SSTable BFs). We evaluate the impact of SSTable BFs on the performance of RocksDB, TWEEZER, and SACK. Specifically, we load 64 M 1-KiB KV pairs into the systems and issue 5 M KV operations (writes or reads). We measure the average write and read latencies for different sizes of SSTable BFs, defined in terms of bits per key. According to the RocksDB documentation, the default size is 10 bits per key. We also evaluate the configurations with 1 and 100 bits per key to analyze their effects.

Figure 19 presents the results. As the size of SSTable BFs increases, the average write latencies for RocksDB, TWEEZER, and SACK remain nearly constant (Figure 19(a)) as the time required to create SSTable BFs is negligible. On the other hand, the average read latencies for RocksDB and TWEEZER decrease with a larger size of SSTable BFs, while SACK's read latency stays stable (Figure 19(b)). The reason is that SACK utilizes KV separation, which

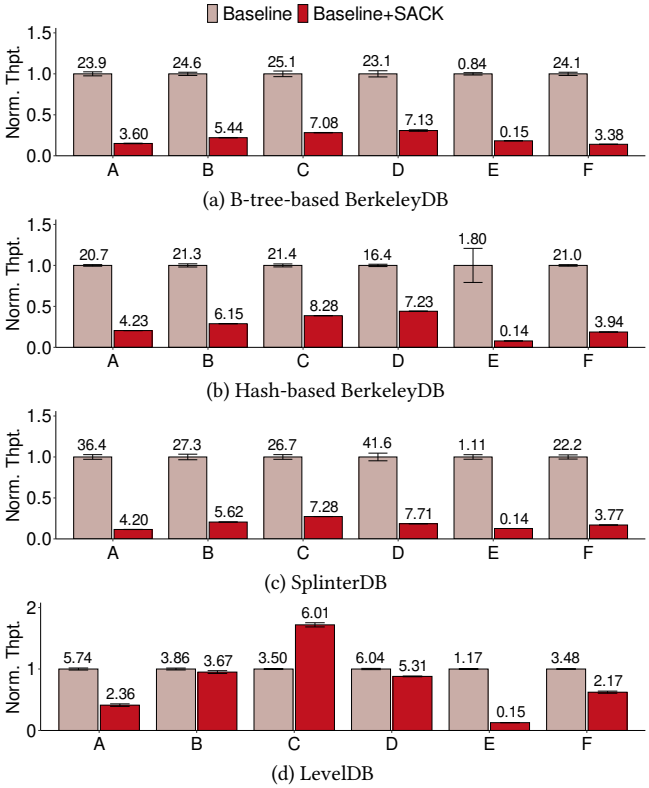


Figure 21: Exp#15 (Applicability of SACK to different KV stores). We consider SGXv1. We plot the throughput normalized by the baseline KV stores without SACK on the SGXv1 testbed. We also report the absolute throughput in KOPS above each bar.

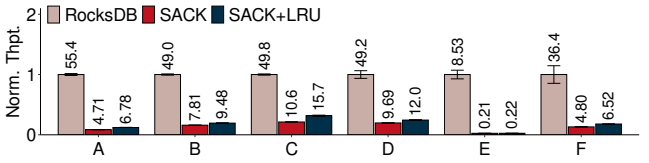


Figure 20: Exp#14 (Compatibility with SGXv2). We plot the throughput normalized by RocksDB's. We also report the absolute throughput in KOPS above each bar.

reduces the size of the LSM-tree in the underlying RocksDB, resulting in marginal benefit from the SSTable BFs. RocksDB and TWEEZER experience a 22.0% and 53.7% reduction in read latency when the size of SSTable BFs increases from 0 to 100 bits per key, respectively. The reason is that larger SSTable BFs reduce the false positive rate, thereby decreasing the number of disk reads required to locate a KV pair in a larger LSM-tree. Note that since TWEEZER requires additional decryption and authentication operations for each SSTable access, reducing the false positive rate via SSTable BFs has a more significant impact on reducing the read latency compared to RocksDB, as both disk reads and cryptographic operations can be reduced.

Exp#14 (Compatibility with SGXv2). We evaluate SACK on an SGXv2 server, which has two eight-core Intel Xeon Silver 4309Y 2.8 GHz CPUs, 256 GiB RAM, 8 GiB EPC memory, and a 1 TiB PCIe 4.0 NVMe SSD. Since SGXv2 supports a larger EPC, we introduce a least-recently-used (LRU) cache inside the enclave to cache the most recently accessed KV pairs (in unencrypted form) and their PIDs. Thus, the enclave can return the KV pairs being read directly from the LRU cache if their policies are satisfied, while still maintaining all security guarantees of SACK. Currently, we set the cache size as 1 GiB and refer to this optimized version as SACK+LRU. We use the same settings as in Exp#1 and compare RocksDB, SACK, and SACK+LRU across the six YCSB core workloads.

Figure 20 presents the results. Compared to SACK, SACK+LRU improves the average throughput by 29.6% since the KV cache reduces the read overhead associated with fetching, verifying, and decrypting KV pairs. For example, in Workload C, which has 100% reads, SACK+LRU achieves 48.1% higher throughput than SACK and 68.5% lower throughput than RocksDB. However, in Workload E, the throughput gain of SACK+LRU over SACK is limited to 4.8%, as the LRU cache does not address the adjacency of KV pairs, leading to frequent cache evictions and misses; a scan-tailored cache may be more suitable for scan-intensive workloads [93].

Due to different testbed hardware configurations, we observe different performance results here compared to those in Exp#1. In particular, the SGXv2 testbed has faster I/Os with a PCIe 4.0 SSD. Thus, SACK and SACK+LRU on the SGXv2 testbed have 37.7% and 78.2% higher throughput than SACK on the SGXv1 testbed on average across all workloads, respectively; RocksDB's throughput on the SGXv2 testbed is 130.2% higher than on the SGXv1 testbed.

Exp#15 (Applicability of SACK to different KV stores). We evaluate the applicability of SACK with various KV stores, including BerkeleyDB [70] with B-tree-based and hash-based storage backends, B^ϵ-tree-based SplinterDB [21], and the LSM-tree-based LevelDB [36] (note that LevelDB shares API compatibility with other KV stores such as PebblesDB [77] and HyperLevelDB [26]). To ensure fair comparisons, we disable compression in LevelDB, as other KV stores lack this feature. Note that scans in the hash-based BerkeleyDB do not provide lexical ordering of keys, which distinguishes it from the other KV stores. Following Exp#1, we load 64 M 1-KiB KV pairs into the systems and issue 5 M KV operations using the six YCSB core workloads. Our analysis focuses on the performance of these systems with and without SACK on both SGXv1 (Section 5.1) and SGXv2 (Exp#14) testbeds. We also compare the performance on the SGXv2 testbed with and without the in-enclave KV cache in SACK, as in Exp#14.

Figure 21 shows the results on the SGXv1 testbed. For BerkeleyDB with B-tree-based storage backend (Figure 21(a)), BerkeleyDB with hash-based storage backend (Figure 21(b)), SplinterDB (Figure 21(c)), and LevelDB (Figure 21(d)), integrating SACK leads to an average throughput drop of 5.1×, 5.2×, 6.1×, and 2.4× compared to the baseline systems, respectively. In particular, for LevelDB (Figure 21(d)), LevelDB+SACK outperforms the baseline LevelDB by 71.7% in the read-intensive Workload C. This improvement is attributed to SACK's KV separation mechanism, which reduces the size of the LSM-tree in the underlying LevelDB.

Figure 22 shows the results on the SGXv2 testbed. Similar to the SGXv1 results, SACK incurs average throughput drops of 16.2×, 63.1×, 9.8×, and 5.9× without the in-enclave KV cache, and 10.3×, 37.6×, 6.8×, and 4.9× with the in-enclave KV cache, compared to the baseline systems for BerkeleyDB with B-tree-based storage back-

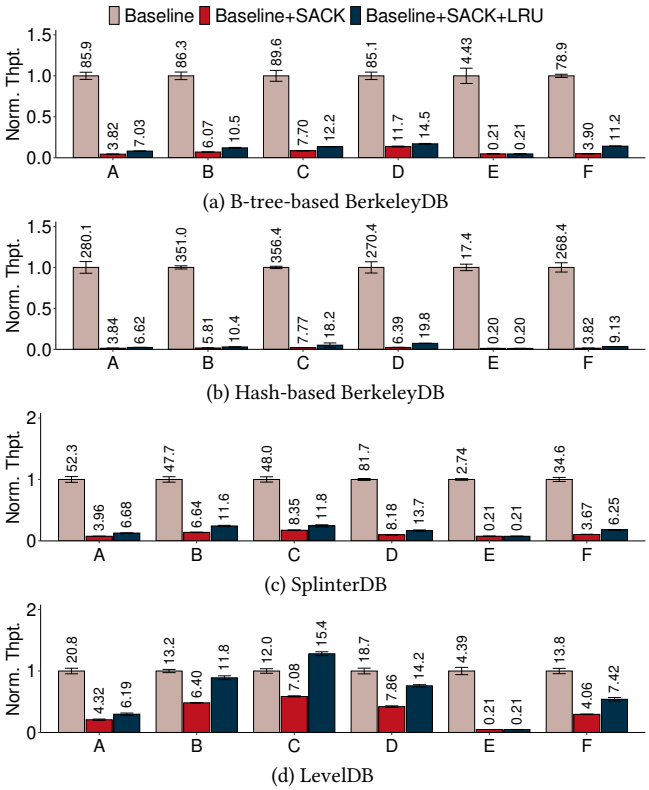


Figure 22: Exp#15 (Applicability of SACK to different KV stores). We consider SGXv2. We plot the throughput normalized by the baseline KV stores without SACK on the SGXv2 testbed. We also report the absolute throughput in KOPS above each bar.

end (Figure 22(a)), BerkeleyDB with hash-based storage backend (Figure 22(b)), SplinterDB (Figure 22(c)), and LevelDB (Figure 22(d)), respectively. As analyzed in Exp#14, the in-enclave KV cache in SACK reduces the read overhead associated with fetching, verifying, and decrypting KV pairs, resulting in higher throughput for read-intensive workloads. Note that the performance gap between SACK and the baseline systems is larger on the SGXv2 testbed than on the SGXv1 testbed. The reason is that the faster I/Os provided by the PCIe 4.0 SSD in the SGXv2 testbed alleviates the I/O bottleneck in all systems (Exp#14), while SACK remains bounded by cryptographic overhead (Exp#3).

Our results demonstrate that SACK is applicable to various KV stores, with reasonable overheads on both SGXv1 and SGXv2 testbeds.