Protecting LSM Key-Value Stores using Secure Enclaves

Giannos Evdorou

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Angelos Bilas*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Protecting LSM Key-Value Stores using Secure Enclaves

Thesis submitted by
Giannos Evdorou
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Giannos Evdorou

Committee approvals: _____
Angelos Bilas
Professor, Thesis Supervisor

_____
Giorgos Vasiliadis
Assistant Professor, Committee Member

_____
Polyvios Pratikakis
Associate Professor, Committee Member

Departmental approval: _____
Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, June 2024

Protecting LSM Key-Value Stores using Secure Enclaves

Abstract

Log-structured merge (LSM) key-value stores are widely used in various applications mainly due to their ability to handle writes efficiently. However, ensuring the security and integrity of the stored data remains challenging, especially in untrusted infrastructures (such as cloud environments). Hardware-based Trusted Execution Environments (TEEs) are a practical solution that provides trust guarantees for code execution in third-party computing environments and protects even against highly privileged adversaries. Previous work has implemented fully functional, secure key-value stores in TEEs; however, they suffer from high memory pressure which is a major limitation for TEE applications.

This thesis presents Fennec, a secure LSM-based key-value store designed to protect data confidentiality and integrity using hardware-based TEEs. Fennec leverages unique, per-level encryption keys and hash-based message authentication codes (HMACs) to safeguard data against various threats, including root-privileged access, tampering, physical attacks, and replay attacks. The system also employs a log protection mechanism to ensure data recoverability in the face of failures while preventing rollback attacks. Our evaluation demonstrates that Fennec achieves strong security guarantees with a slowdown of 6.6× when compared to the unprotected key-value store while reducing the amount of memory needed to store the history of encryption keys by up to 50× compared to previous work.

Προστασία LSM Key-Value Store Συστημάτων χρησιμοποιώντας Ασφαλή Περιβάλλοντα Εκτέλεσης

Περίληψη

Τα key-value store συστήματα που είναι βασισμένα στο LSM δέντρο, χρησιμοποιούνται ευρέως σε διάφορες εφαρμογές κυρίως λόγω της καλής απόδοσης τους στις εγγραφές δεδομένων. Ωστόσο, η διασφάλιση της εμπιστευτικότητας και της ακεραιότητας των δεδομένων που αποθηκεύονται σε αυτά τα συστήματα παραμένει δύσκολη, ειδικά σε μη αξιόπιστες υποδομές (όπως περιβάλλοντα νέφους). Τα Ασφαλή Περιβάλλοντα Εκτέλεσης βασισμένα σε υλικό παρέχουν εγγυήσεις ασφάλειας για την εκτέλεση του κώδικα και προστατεύουν ακόμη και από υψηλά εξιουσιοδοτημένους επιτιθέμενους. Προηγούμενες εργασίες υλοποιούν πλήρως λειτουργικά, ασφαλή key-value stores σε Ασφαλή Περιβάλλοντα Εκτέλεσης· ωστόσο, παρουσιάζουν αυξημένο φόρτο στη μνήμη του Ασφαλούς Περιβάλλοντος Εκτέλεσης.

Αυτή η εργασία παρουσιάζει το Fennec, ένα ασφαλές key-value store σύστημα, σχεδιασμένο να προστατεύει την εμπιστευτικότητα και την ακεραιότητα των δεδομένων χρησιμοποιώντας Ασφαλή Περιβάλλοντα Εκτέλεσης βασισμένα σε υλικό. Το Fennec εκμεταλλεύεται μοναδικά, ανά-επίπεδο κλειδιά κρυπτογράφησης και κωδικούς ελέγχου αυθεντικότητας μηνυμάτων με τεχνικές κατακερματισμού (HMACs) για να προστατεύσει τα δεδομένα από διάφορες απειλές, όπως πρόσβαση με δικαιώματα root, αλλοίωση, φυσικές επιθέσεις και επιθέσεις επανάληψης. Το σύστημα χρησιμοποιεί επίσης έναν μηχανισμό προστασίας του log για να διασφαλίσει την δυνατότητα ανάκτησης δεδομένων σε περίπτωση αποτυχιών, αποτρέποντας τις rollback επιθέσεις. Η αξιολόγησή μας δείχνει ότι το Fennec επιτυγχάνει ισχυρή ασφάλεια με επιβράδυνση $6,6\times$ σε σύγκριση με το μη προστατευμένο key-value store σύστημα, ενώ μειώνει τις απαιτήσεις μνήμης για την αποθήκευση του ιστορικού των κλειδιών κρυπτογράφησης έως και $50\times$ φορές σε σύγκριση με προηγούμενες εργασίες.

Acknowledgements

During my postgraduate studies, many people have supported me, each one of them in their own way. I want to thank my supervisor, Professor Angelos Bilas, for his guidance. Working with him for four years has made me a better scientist. I would also like to thank Professor Giorgos Vasiliadis for his invaluable input to this work and Giorgos Saloustros for his great feedback and advice. I would also like to thank all the people of CARV, where I made some very good friends.

I could not have done this without my family and friends. A huge thank you to my parents, Doros and Maria, for always being by my side and supporting me every step of the way. My sisters, Evelyn and Antonella, for being the best sisters anyone could ask for. Last but not least, my best friend Giannis who always has my back.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Key-value stores (KVS) are storage systems designed to provide fast access to data using a simple key-value lookup system [7, 25]. They have been proven to be very efficient for a plethora of diversified applications that store and retrieve large amounts of data, from real-time analytics systems, to graph-based systems, message queues, and feature storage for recommendation systems [19, 4]. However, the lack of trust in third-party or shared resources poses a significant barrier to outsourcing computations, especially when sensitive data is involved. For example, security violations or attacks can compromise the stored data and the query operations. As the importance of data increases for modern applications and workflows, protecting data from other applications and users becomes a significant challenge. Thus, ensuring data confidentiality and detecting tampering to ensure correct query results becomes critical.

Although current privacy-enhancing technologies, such as homomorphic encryption, can increase security, they are still very slow and have high storage requirements, especially for complex computations [24]. Hardware-based Trusted Execution Environments (TEE) are an attractive alternative that can provide trust guarantees for code execution in third-party computing environments and protect even against highly privileged adversaries [20]. Previous work [2, 10] has implemented fully functional, secure KVSs in TEEs. They achieved strong security guarantees with acceptable performance overhead. However, to ensure the integrity and freshness of their data, they either created Merkle trees [2] or utilized a unique encryption key for each SSTable [10]. Both of these methods impact the scalability of the KVS. The traversal of a Merkle tree is costly and increases the latency of lookups. On the other hand, using a unique encryption key for each SSTable requires maintaining the history of encryption keys, leading to increased memory pressure within the Enclave Page Cache (EPC).

In this thesis, we implement Fennec, a secure KVS that runs inside a TEE. The main goal of Fennec is to improve the scalability of KVS when run in TEEs by (i) securely moving most data structures outside the EPC, as TEE memory pressure is a significant point of performance overhead, and by (ii) leveraging full-leveling

compactions to reduce the number of unique encryption keys required (i.e., one encryption key per level instead of one per SSTable).

We implement Fennec on Tebis [23], an open-source KVS, and evaluate it using the popular Yahoo Cloud Serving Benchmark (YCSB) [6]. Our results show that Fennec can fully protect data against strong adversaries, including physical attacks and root-privileged malware, with less than 13× slowdown compared to the default execution of Tebis. Our experimental evaluation reveals that this slowdown is mainly the result of excessively switching in and out of the TEE, everytime Fennec sends or receives data over the network. We mitigate this by implementing custom operators that batch many GET/PUT requests in the same buffer; this results to a slowdown of 6.6× compared to native Tebis, while maintaining the same, strong security guarantees.

Chapter 2

Background

2.1 Intel SGX

SGX is Intel's hardware that provides confidentiality and integrity to programs, called enclaves, that run in a predetermined address space. The enclaves are signed at compile time so that the hardware can verify that the code we expect runs in the enclave. SGX provides 128MB or, in some cases, 256MB of trusted enclave memory called EPC, in which the enclave's data is secure and cannot be read or modified by anyone outside the enclave, including the OS. Some limitations of SGX include modifications that need to be made to applications in order to run them in an enclave, which are mostly solved by using libary OSs such as SCONE [1], Occlum [21], and others, which allow an unmodified application to run in SGX, limited EPC capacity which results in significant overheads when an application uses more than the available EPC due to the cost of decrypting and encrypting pages to page them in or out, and no support for secure use of storage devices which means that storage applications have to secure anything that goes to a storage device manually.

2.2 LSM Key-Value Stores

Log-structured merge trees (LSM Trees) [15] are a key optimization technique used in modern key-value stores to improve write performance. Modern KVSs consist of a Level 0 (L0) in memory, with the remaining levels stored on disk. The disk levels are sorted by key. Every level has a capacity threshold larger than its previous level by a fixed growth factor. The levels of the KVS consist of SSTables, which are immutable files, each containing a range of keys. The write process involves appending incoming writes to the in-memory Level 0 (L0). Once L0 reaches a capacity threshold, a compaction happens. Compaction is the process of moving data from level n to level n+1 to free up space in level n. In full-leveling compactions, when a level n is full, all its SSTables are merged with all the SSTables in level n+1, and the result becomes the new level n+1. In incremental

compactions, when a level n is full, only one SSTable moves to the next level n+1. This leads to a higher total number of compactions, but it amortizes the cost more efficiently than full-leveling compactions. When performing reads, queries first search in L0. If the key cannot be found, the search continues sequentially to the on-disk levels. LSM Trees offer significant advantages, especially their efficiency in handling write operations, but they have other disadvantages, such as read amplification.

Chapter 3

Methodology

3.1 Design Objectives

Even though TEEs aim to provide strong data protection during processing, it is equally important to guarantee that data remains well protected throughout their whole lifecycle, including when transiting over the network and when stored in main memory and storage devices. One of the most effective ways to protect data is by using encryption, which should be enforced end-to-end in the phases above (namely, transit, in-use, and rest phases). We briefly describe each of them below:

- Data in transit: The data is transferred over the network in a client-server fashion (i.e., the client reads or writes key-value pairs on a remote server). In our case, the network is considered untrusted. Hence, the connection should be encrypted. This can be performed at different levels of the network stack, e.g., on the application level by using TLS/SSL sockets or at the network level by establishing IPsec or VPN channels between the nodes. Unlike the traditional client-server architecture, the operating system is also considered untrusted; hence, the encrypted connection should terminate within the TEE. By doing so, the data will only be decrypted within the protected space of the TEE without risking being leaked into the network or the operating system.

- Data in use: The data are protected during computation using hardware-based TEEs (e.g., Intel SGX, ARM TrustZone, etc.). TEEs provide an isolated space inaccessible to other applications, the operating system, or other hardware modules.

- Data at rest: The storage devices typically reside outside the trusted domain. Hence, the data should always be stored in encrypted form. The encryption can be performed transparently, at the block level (e.g., similar to dm-crypt) or the application layer (e.g., similar to SQL TDE). In any case, it is necessary to redesign the data structures to extend the trust to the untrusted

storage medium in order to achieve end-to-end security properties between the TEE and the storage mediums.

3.2 Design Space

Implementing a secure KVS requires many different design decisions, depending on a user's demands for the security guarantees it provides, its use case, and performance requirements. Table 3.1 summarizes the design space. Before discussing the pros and cons of each row, we clarify the contents of each column in the table.

Explanation of table columns:

- **Configuration:** Describes what kind of encryption scheme is used, who performs the encryption, and whether the KVS runs in a TEE or not.
- **Server Changes:** Denotes if the KVS needs to change for this configuration, and to what degree.
- **Secure Multi-Party:** Indicates if the configuration supports multiple clients querying the KVS securely.
- **Index:** The index is the data structure used to search for a specific key in every level. This column shows where the index is placed (if applicable) and how it is encrypted/decrypted if at all.
- **Confidential:** Indicates if the keys and/or values remain confidential in this configuration.
- **Scans:** Scans are a range query, starting at a key specified by the client. This column represents if the configuration is able to perform scans.
- **EPC Usage:** Shows the EPC memory usage depending on where the index is placed in each configuration for the configurations running in a TEE.
- **Integrity:** Indicates if the integrity of code and/or data can be verified.

Description of configurations:

Symmetric encryption from clients - No TEE: This configuration involves a client that symmetrically encrypts the data before sending it to the server. The server is not running inside a TEE, so the server machine (and operating system, etc) must be trusted. The server can run without any changes since it is unaware that the data is encrypted. This configuration does not allow

Configuration	Server Changes	Secure Multi-party	Index	Confidential	Scans	EPC Usage	Integrity
Symmetric encryption from clients (No TEE)	No	No	Built encrypted	Key Value	No	-	Data
Symmetric encryption from clients (TEE)	Minimal	No	In host memory built encrypted	Key Value	No	Low	Code Data
OPE encryption from clients (No TEE)	No	No	Built encrypted	Key Value	Yes	-	Data
OPE encryption from clients (TEE)	Minimal	No	In host memory built encrypted	Key Value	Yes	Low	Code Data
Symmetric encryption from server (TEE)	Yes	Yes	In EPC (Enc/Dec transparently)	Key Value	Yes	High	Code Data
	”	”	In host memory (Enc/Dec manually)	Key Value	Yes	Low	Code Data
	”	”	In host memory encrypted (never decrypted)	Key Value	No	Low	Code Data
	”	”	In host memory unencrypted	Value	Yes	Low	Code Data
OPE encryption from server (TEE)	Yes	Yes	In host memory encrypted	Key Value	Yes	Low	Code Data

Table 3.1: The different possible secure key-value store designs.

multiple clients to query the KVS securely, since the client holds the encryption key of the data. The index is built based on the encrypted data, so no changes need to be made to preserve the data's confidentiality. The key and the value remain confidential, as only the client can decrypt them. However, scans do not work with this configuration as the server does not know the plaintext order of the keys. Finally, if the client wants to implement integrity checking, it can; however, rollback protection seems complicated.

Symmetric encryption from clients + TEE: This configuration is similar to the previous one, with the difference that the server is now running inside a TEE. Small changes need to be made to the server, first to be able to run inside a TEE, and second, to explicitly place the index outside the EPC and

reduce EPC usage since the client already encrypts the data. Secure multi-party is still not supported since the client holds the encryption key for the data. The confidentiality of the keys and values is preserved since only the client can decrypt them. Scans are still not supported since the server knows nothing about the plaintext order of the keys. EPC usage is low as all the data and metadata can reside in the host's memory since the server never deals with plaintext data. Finally, the client can verify the TEE's integrity using attestation and can again choose to implement integrity checking using hashing for the data, but rollback attack protection remains complicated.

OPE encryption from clients - No TEE: In this configuration, the clients use Order-Preserving Encryption (OPE) (e.g., Boldyreva [3]) to encrypt the data before sending it to the server. OPE is a type of encryption that preserves the lexicographic order of the plaintexts when they are encrypted into ciphertext. However, OPE is significantly more expensive than symmetric encryption (e.g., AES). This configuration offers the same benefits as the first configuration, with the addition of scans being possible since the order between the keys is preserved even after encryption.

OPE encryption from client + TEE: In this configuration, the clients use OPE to encrypt the data before sending it to the server, which runs in a TEE. The pros and cons of this configuration are the same as those of the second configuration, with the additional capability of scans for the same reasons mentioned before.

Symmetric encryption from server + TEE: In this configuration, the server running in a TEE performs data encryption and decryption, as well as any integrity checking or other security mechanisms. This requires changes to the server to preserve the confidentiality and integrity of the data. As the server is responsible for the encryption keys, this configuration can securely support multiple clients. We split this configuration into four variations depending on how we handle the index metadata.

1. The index lives in the EPC. This means that it is encrypted and decrypted transparently when it needs to be paged in or out. This configuration can offer confidentiality for both the keys and the values and can perform scans. However, keeping the index in the EPC can result in memory pressure, a significant overhead in TEE applications due to the need to encrypt and decrypt data to page it in or out.
2. The index lives in the host memory and is encrypted/decrypted manually. This configuration also offers confidentiality for both the keys and the values and can provide scans. Keeping the index encrypted in the host memory reduces memory pressure in EPC. However, every time the server searches for a key, it needs to decrypt the corresponding index entry before performing the comparison.

3. The index lives in the host memory and is built on the encrypted data. If we build the index on the encrypted keys and sort all the levels based on the encrypted key, we can perform lookups without decryption and leaking any data information. However, scans will not be possible since the order of the ciphertext is not the same as the order of the plaintext.
4. The index lives in the host memory, but it is unencrypted. This configuration leaks information about the keys, even if we build the index with keys that do not necessarily exist in the data. However, lookups will be much faster since there is no need for encryption or decryption of the index and no memory pressure since the index is in the host memory.

OPE encryption from server + TEE: In this configuration, the server encrypts the data with OPE. By doing this, the index can now be built on the encrypted data and reside in the host memory, but now scans will be possible since the order between the ciphertexts will be the same as the order between the plaintexts.

3.3 Threat Model

In our work, we assume a strong attacker that can take complete control of the system that our server is running on, except the SGX-protected enclave. Therefore, the attacker can modify the data that resides in memory and the data on disk. The attacker cannot modify the data in the EPC. Our server will not accept the attacker as a client by using appropriate attestation, which is an already solved problem, so the attacker cannot query the KV store. This work does not attempt to solve existing SGX limitations and implementation bugs, which are orthogonal to our proposed system and will be solved in future releases. Most research on secure systems that run on Intel SGX assumes this threat model.

3.4 Design

The overview of Fennec is shown in Figures 3.1 and 3.2 for GET requests and PUT requests respectively. As we can see, Fennec uses a socket-based interface, making integration with any application easy. More specifically, clients can communicate over TLS/SSL sockets with the server running inside the TEE. The clients can either send GET requests to retrieve the values of specific keys or PUT requests to insert or update key-value pairs. The server uses Direct I/O to write to the storage device, and `mmap()` to map the storage device to the host memory address space and perform reads. Fennec handles plaintext data only inside Intel SGX's secure and trusted environment. Data is encrypted every time it leaves the EPC, whether on the network, the host memory, or the storage device. Similarly, whenever Fennec reads data from the network, the host memory, or the storage device, it decrypts it safely inside the EPC.

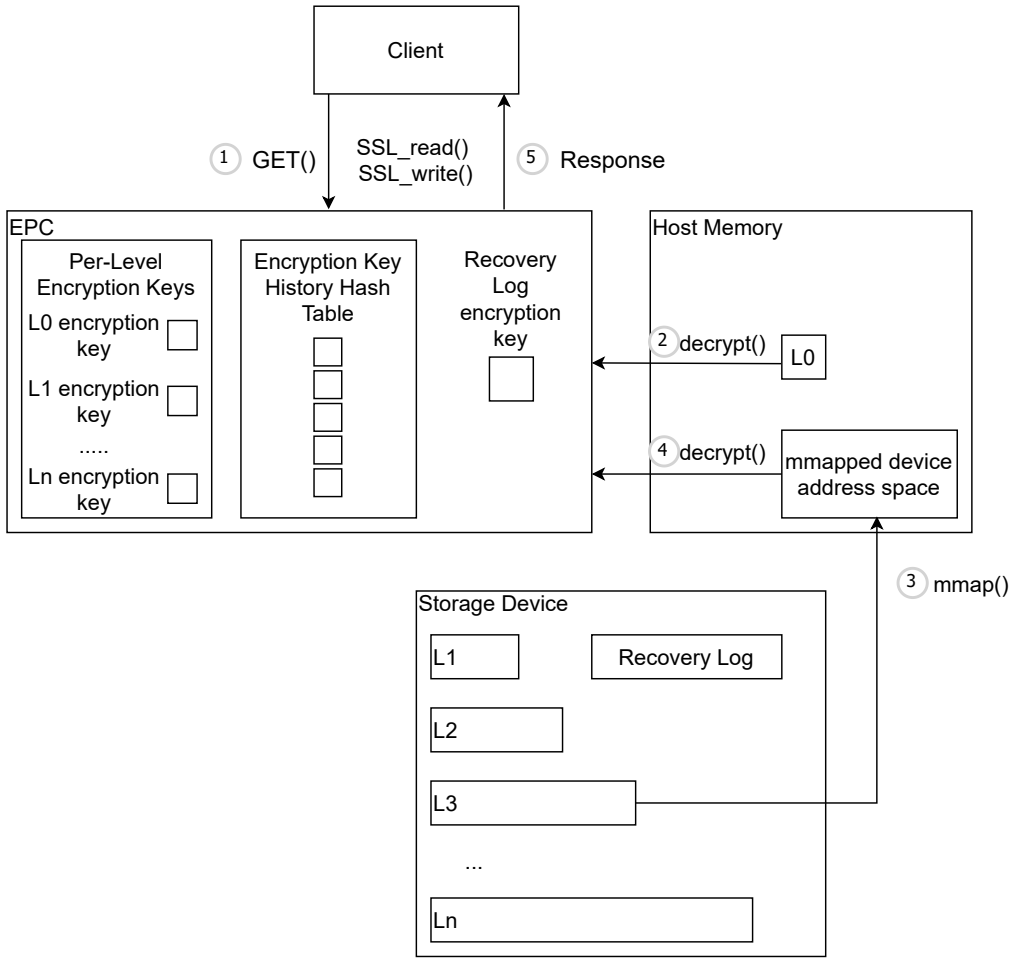


Figure 3.1: The high-level system overview for GET requests.

3.4.1 Client-Server Approach

Implementing a KVS in TEEs requires clear boundaries between the server running in the TEE and the client. The sockets() interface becomes the sole interface between the client and the server, used to send requests and receive responses. Using sockets enables the KVS to serve both local and remote clients securely through the same interface. The clients connect to the server through TLS/SSL connections. Thus, all data is transferred over the network in encrypted form and ends up in the EPC. The current implementation of Fennec uses the mbedtls [12] library for handling the TLS/SSL connections. Since Fennec is TLS-compatible, the clients do not need any modifications to communicate with the server.

3.4.2 Data Encryption

An important decision that we need to make when designing a secure KVS is the granularity at which the data will be encrypted at the storage device. In previous work [2], the data is encrypted at a block granularity. This reduces the spatial overhead that HMACs introduce, but comes at the cost of decreased performance due to the need to decrypt and calculate an HMAC over a whole block every time the system needs to read a specific key-value pair. In Fennec, the granularity of encryption is the key-value pair. This enables us to verify the freshness and integrity of the pair more efficiently, as we only need to calculate the hash of the pair instead of a block any time we need to execute an operation. When the server receives a PUT request from the client, it computes the pair's HMAC using SHA-256, manually encrypts it using symmetric encryption (AES) along with the HMAC, and places it in L0, which resides in the host's memory.

3.4.3 Per-Level Encryption Key

There have been two main approaches proposed to guarantee data freshness. The one approach, used in Speicher [2], uses a single encryption key for all the data and constructs a Merkle tree to verify the integrity and freshness of the blocks holding the key-value pairs. The other approach, used in Tweezer [10], uses a separate, unique, encryption key for each SSTable. By avoiding using the same encryption key twice, Tweezer can verify the integrity and freshness of a key-value pair without constructing a Merkle tree. The reason they are able to achieve this is because of some properties that hold for every SSTable. The first property is that an SSTable is immutable, and the second is that every key in an SSTable is unique. By having an HMAC per key-value pair combined with the secret encryption key, the attacker cannot construct an encrypted key-value pair with a valid HMAC. The attacker's only remaining possibility is copying an existing key-value pair. If the attacker copies a key-value pair with the same key from a different SSTable, it would have been encrypted with a different encryption key, resulting in the HMAC check failing upon decryption. The attacker cannot find a key-value pair with the same key in the same SSTable due to the uniqueness of every key. The drawback of this technique is that it does not scale well. To avoid replay attacks, Tweezer must remember the history of its encryption keys. If an encryption key is used twice, key-value pairs from the previous time it was used will become valid options for a replay attack. For long-running workloads, the memory pressure in the EPC due to the size of the history of encryption keys will become a bottleneck. We reduce the size of the history by leveraging full-leveling compactions. In a system with full-leveling compactions, every level has the same properties as an SSTable; once written during the compaction, it becomes immutable, and every key in a level is unique. Thus, we can use a unique encryption key per level, allowing us to reduce the size of the history of encryption keys significantly while keeping the same security guarantees and reducing EPC usage by not constructing a Merkle

Tree.

3.4.4 Log Protection

The recovery log is where the system persists the KV pairs that are in L0 in order to recover from a failure. The log is stored on the storage device outside the EPC; therefore, an attacker can read or modify its data. Hence, it is crucial to ensure that the data stored in the log is encrypted and to verify its integrity and freshness to protect the log against rollback attacks. Fennec protects its recovery log by leveraging the existing Log Sequence Numbers (LSN), an integer appended to each key-value pair before it becomes a log entry. Each log entry consists of an LSN, the key-value pair, and an HMAC, which we encrypt with a secret key. Every time Fennec flushes a log buffer to the device, we let the client know the LSNs of the log entries we persist. Since the entries are encrypted with a unique key, an attacker cannot construct a valid entry and can only replace one with an older one. However, during recovery, Fennec will let the client know the LSNs of the recovered pairs, and if they are older than they should be, then the trusted client will let the server know that a replay attack happened.

3.4.5 Primitive Operations

3.4.5.1 PUT

When Fennec receives a PUT request, the encryption key for L0 is retrieved from the EPC. It then computes the HMAC of the pair and appends it to the key-value buffer. The buffer is then encrypted and placed in L0, which resides in the host's memory. It then decrypts the buffer, appends the LSN, and re-encrypts it with the log's encryption key so it can be inserted into the log. Once it writes the log entry, the server will respond to the client, stating that the PUT request is complete.

3.4.5.2 GET

When Fennec receives a GET request for a key, it searches level by level for the key to retrieve its value. It performs a binary search on the index of every level, which is made of encrypted keys, by decrypting the index key and comparing it to the plaintext key received from the client. When it reaches the leaf, it decrypts the key-value buffer with the appropriate encryption key retrieved from EPC, calculates the HMAC of the key-value pair, and compares it to the HMAC stored in the buffer. If the two HMACs match, Fennec returns the value to the client.

3.4.5.3 Compaction

When a level of the KV store is full, the server performs a compaction. Compaction is the merging of two levels, n and n+1. Before performing the compaction,

Fennec has to decrypt both levels. There are two reasons for this. 1) Symmetric encryption (AES), which we use, does not preserve the order between plaintexts when it encrypts them to ciphertext, and 2) every new level that is created as a result of compaction must be encrypted with a new, unique encryption key to protect the data from replay attacks as explained in 3.4.3. The compaction is done by decrypting both levels with their respective encryption keys, verifying every pair's HMAC, and merge-sorting since both levels are already sorted. The KV pairs in the new level are then encrypted with a new unique encryption key. The new encryption key is encrypted with the log's encryption key together with an LSN and appended to the log.

3.4.5.4 Recovery

Every key-value pair entry in the log includes a Log Sequence Number (LSN). When a log buffer is flushed, the server lets the client know the LSN of the last KV entry in the buffer. On compaction completion from L0 to L1, Fennec records the offset of the log up to which the entries are in L1 so that it knows where the next recovery operation should start. When the system recovers from a crash, Fennec starts replaying the log entries by decrypting them one by one from the offset previously recorded until the last entry. If, at any point, it encounters an LSN that is not consecutive with the LSN of the previous entry, it aborts compaction since an attacker modified the log. When it reaches the end of the log, it sends the last LSN to the client and confirms that this is the latest entry inserted in the log.

3.4.6 Batched Operations

It has been shown that performing system calls from TEEs is quite expensive, mainly because of the TEE exits and re-entries that occur [14]. We experimentally verify this effect in Section 5.2. To quantify the effects of exiting and re-entering the TEE for receiving and sending network requests, we implement custom GET and PUT operations that batch requests up to a predefined number. The server then reads all the requests in a single call/buffer and unwraps them to execute them. When the server executes all the requests, it similarly responds to the client by batching all responses in one buffer. As we will see in Section 5.2 this can boost the performance up to 1.96× times.

3.5 Implementation

We implement Fennec by modifying Parallax [25], a modern LSM tree KVS, and more specifically, Tebis [23], which offers a socket interface for clients on top of Parallax. Parallax already uses full-leveling compactions, a feature that allows us to reduce the number of encryption keys needed for a specific dataset size.

Implementing an application to run in TEE requires to choose a suitable framework. There are several frameworks (e.g., SCONE [1], Graphene-SGX [5], and Occlum [21]) that aim to ease the development process by offering ready-to-use containers, libOSes, libraries, and runtimes that allow existing applications to run (almost) unmodified. However, these frameworks vastly increase the TCB (Trusted Computing Base) of the application, which now includes any untrusted third-party code, as well as the underlying libOS. Other approaches, such as Google's Asylo [9] and Microsoft's Open Enclave [13], focus on portability by offering a universal interface for cross-platform TEE offloading. The latter approaches support cross-platform implementations, enabling the exact implementation to run over different hardware TEEs. For example, Open Enclave supports Intel SGX and OP-TEE OS on ARM TrustZone.

Fennec is implemented on top of Open Enclave and, more specifically, EdgelessRT [22]. Open Enclave is a hardware-agnostic open-source library for developing applications that run in TEEs. This enables Fennec to run on both Intel SGX and Arm TrustZone platforms without any modifications. EdgelessRT is an SDK built on top of Open Enclave that provides extended C/C++ support, with support for more libc and POSIX functions. We implemented two extra functions that were not supported by the SDK but were needed by Fennec. One function allocates aligned memory from the host, which we can then use for direct I/O. The other is a function that memory maps a region of the host's memory to a file.

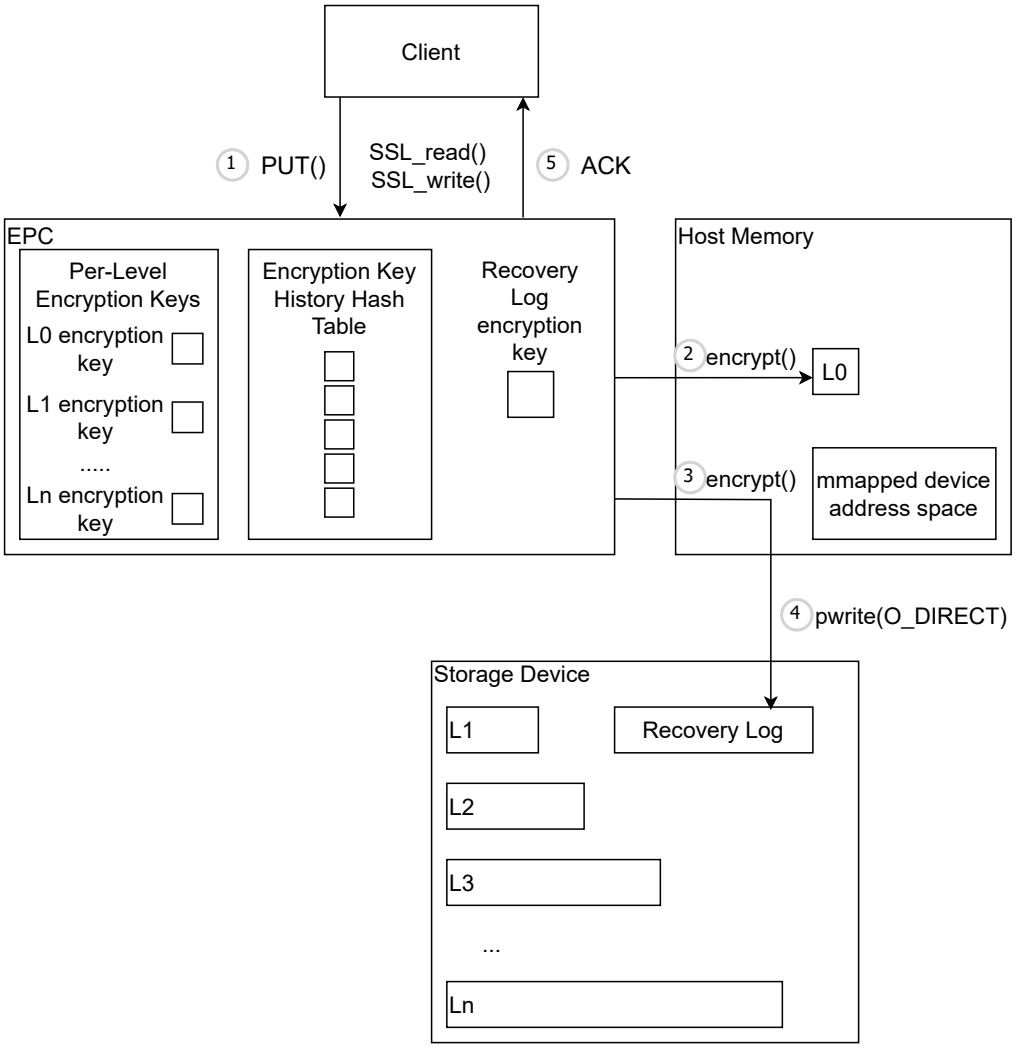


Figure 3.2: The high-level system overview for PUT requests.

Chapter 4

Security Analysis

4.1 Man-in-the-middle attack

A man-in-the-middle attack occurs when an attacker gets in the middle of communication to either eavesdrop or impersonate a legitimate communication party. Our system successfully deals with this kind of attack. The client can perform remote attestation to verify the server's identity and confirm the integrity of the code running inside the TEE before establishing a connection. All the data subsequently traveling over the network is encrypted through TLS/SSL sockets, so any adversary cannot deduce any meaningful information by inspecting the network packets. One can implement a password-based authentication policy to prevent the attacker from impersonating a trusted client, but this is orthogonal to our design.

4.2 Physical Attacks / Cold Boot Attacks

A physical attack or a cold boot attack can happen when the adversary has access to the hardware on which our server is running. There are various kinds of physical attacks. Assuming SGX successfully protects all the data in the EPC from physical attacks, our design is resilient to them. This is because any data leaving the EPC for the client, host memory, or storage device is encrypted before transmission.

4.3 Log Replay

A log replay occurs when an attacker replaces the log or part of the log with a previous version. This causes the server to replay older entries, rendering the database inconsistent. To counter this, we leverage the already existing LSN appended to every key-value pair before inserting it into the log. We encrypt the LSN along with the key-value pair with the log's encryption key and the HMAC of the key-value pair and append it to the log. When we append a new entry to

the log, we let the client know the LSN of the latest entry. Since the attacker does not know the encryption key of the log, they cannot create a valid log entry that will not fail the HMAC comparison. The only option is to replace either part of the log with an older one or the whole log. In the first case, during recovery, the server will notice that the LSN of the entries that are being replayed are not consecutive, as they should be, and it will abort. In the second case, the LSN it will send to the client will not match the latest LSN that the client expects, and the client will let the server know.

4.4 Key-Value Pair Replay

A key-value pair replay attack occurs when the attacker tries to replace either specific key-value pairs with other pairs or a whole level with another level. For the first case, the attacker cannot construct their own encrypted key-value pair with a valid HMAC so their only option is to copy a key-value pair either from the same level or from a different level. If they copy a pair from the same level, then that pair is valid, so the server will not return an incorrect value. If they copy a pair from another level or an older pair from the same level, then the HMAC will not match, as the pair was encrypted with a different encryption key. The same holds for replacing a level with another one since they were encrypted with different encryption keys, and the HMAC computed when retrieving a pair will not match the HMAC found in the pair.

4.5 Rollback Attack

A rollback attack occurs when an attacker replaces all the server data with a snapshot taken at a previous time. This can happen in two ways: online - while the KV store is running, and offline - when the KV store crashes or terminates and resumes execution later. An online attack is impossible because the encryption keys reside in the EPC. If an attacker replaces all the data on disk with a previous version, the data would have been encrypted with a different encryption key, so the HMAC check when decrypting a pair will fail. An offline attack means that the encryption keys will not be in the EPC, and the server will try to recover them from the log. The attacker can replace the log with the previous version from their snapshot so the encryption keys will match the data. However, whenever an encryption key is persisted in the log, the client is informed of the LSN; when the server tries to recover the encryption keys and confirm the LSN with the client, it will realize that this is a previous version of the database.

4.6 Denial-of-Service Attacks

Denial-of-service (DoS) attacks are defined as attempts by the adversary to shut down a machine or service to make it inaccessible to its intended users. Our system

is vulnerable to this type of attack since, for example, someone with physical access to the machine can shut it down and render it unable to respond to requests. However, the data stored in our KVS remains confidential.

Chapter 5

Performance Evaluation

In this section we present the performance evaluation of Fennec when executing in Intel SGX and we compare it with the native execution of Tebis.

5.1 Experimental Setup

5.1.1 Base Setup

We evaluate our work on a desktop machine with SGX1, an Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz processor with 4 cores and 8 threads, 32GB of DDR4 RAM, and 256GB of NVMe SSD. The server and client both run on the same machine. The server runs on the first 2 cores of the machine with 2 threads, and 8 client threads run on the CPU's remaining 2 cores (4 threads). We run our system with 8 regions, each with a L0 size of 8MB and a growth factor of 8.

5.1.2 Workloads

We use the YCSB benchmark [6] to evaluate our system's performance. YCSB has 7 different workloads and we use 2 of them for our evaluation, specifically YCSB Workload A (LoadA), a write-heavy workload with 100% write operations, and Workload C (RunC), a read-heavy workload with 100% read operations. We use small key-value pairs with a total size of 30B.

5.1.3 Baseline and Fennec Configurations

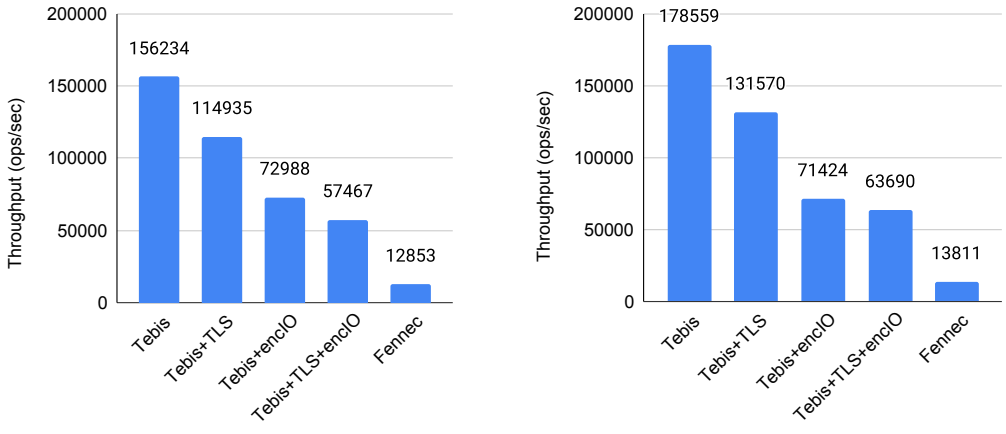
In order to analyse the performance costs of Fennec we use five configurations that also provide different security measures (i.e., securing data when in transit, when in use, and when at rest, and some of their combinations):

- *Tebis*: Native Tebis does not offer any security guarantees for the data. It uses unencrypted sockets, runs outside the TEE, and performs no encryption when storing the data on the storage device.

- *Tebis+TLS*: Native Tebis with TLS sockets. It protects data in transit as it is transferred encrypted over the network.
- *Tebis+encIO*: Native Tebis with encrypted storage I/O and HMAC calculation. It protects data at rest as everything it writes to the storage device is encrypted, integrity-checked, and protected from replay attacks.
- *Tebis+TLS+encIO*: Native Tebis with TLS sockets, encrypted device I/O, and HMAC calculation. It protects data in transit and data at rest. Data in use is still vulnerable since this configuration is not running in a TEE.
- *Fennec*: Fennec runs in an Intel SGX enclave with TLS sockets, encrypted I/O, and HMAC calculation. This configuration offers all the security features and protects data in all three phases.

5.2 Performance Analysis

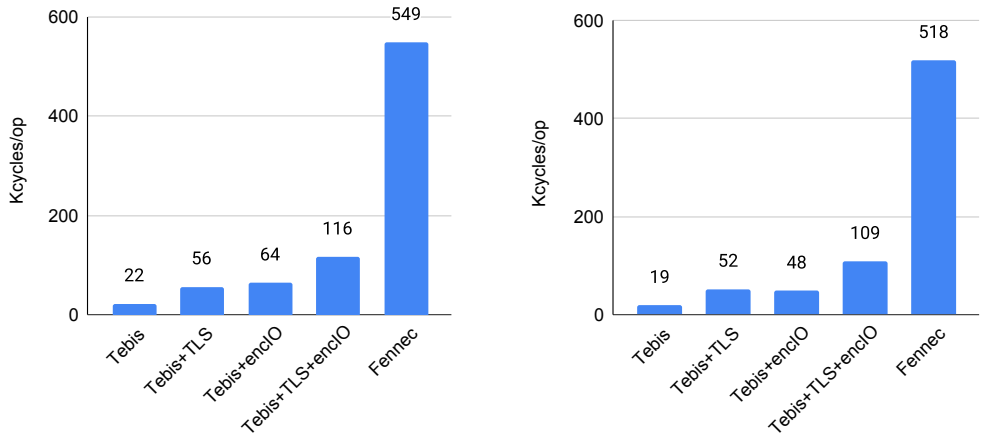
5.2.1 End-to-end Performance



(a) LoadA (b) RunC

Figure 5.1: Throughput achieved for each configuration when runnning LoadA and RunC.

Figures 5.1a and 5.1b show the throughput for the five configurations presented in Section 5.1.3, when running workloads LoadA and RunC respectively. The configuration that offers no data protection at any phase (namely *Tebis*) achieves a throughput of 156234 ops/sec for writes and 178559 ops/sec for reads. When adding TLS encryption to protect the data in transit (namely *Tebis+TLS*), throughput falls to 114935 ops/sec, 1.36× slower, due to the data being encrypted



(a) LoadA (b) RunC

Figure 5.2: The cycles per operation for the different configurations when running LoadA and RunC.

before being sent by the client and decrypted when it reaches the server. The configuration that protects the data at rest (namely *Tebis+encIO*) achieves a throughput of 72988 ops/sec for writes, which is 2.14× slower than default *Tebis*, and a throughput of 71424 ops/sec for reads, which is 2.50× slower. The configuration that combines data protection in transit and at rest (namely *Tebis+TLS+encIO*) achieves a throughput of 57467 ops/sec in LoadA and 63690 ops/sec in RunC. Compared to default *Tebis*, the slowdown is 2.72× and 2.80×, respectively. Finally, the last configuration, namely *Fennec*, runs inside the Intel SGX enclave and protects the data in all three phases. The throughput for LoadA is 12853 ops/sec, 12.16× slower than native Tebis, and 13811 ops/sec for RunC, 12.93× slower than native Tebis.

For the same set of experiments, we also calculate the average number of cycles the server needs for each operation in our workload, for all the configurations: The formula we use to calculate this metric is:

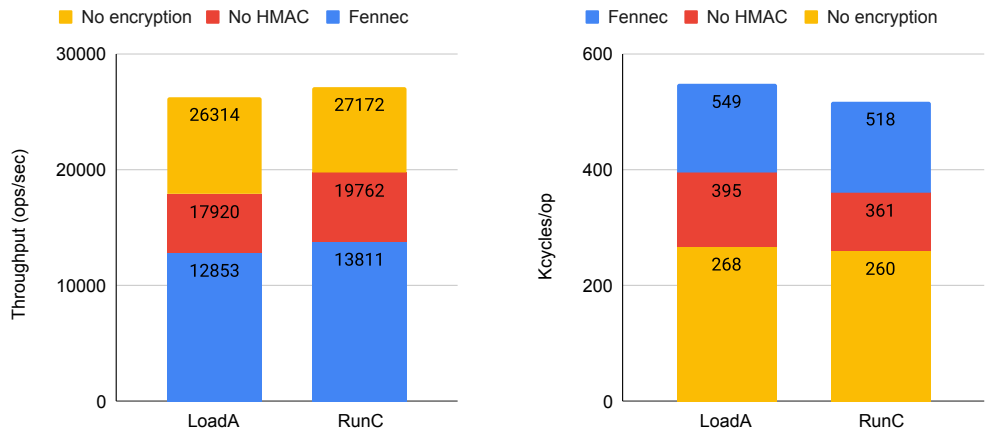
$$cycles/op = \frac{\frac{CPU\_Utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}}$$

Figures 5.2a and 5.2b show the results for LoadA and RunC. As expected, *Tebis* needs the least amount of cycles to complete an operation with 22 Kcycles/op for writes and 19 Kcycles/op for reads. Enabling the TLS sockets to protect data in transit incurs a 34 Kcycles/Op overhead for writes and 33 Kcycles/op for reads. Enabling data encryption on the storage device to protect data at rest incurs a 42 Kcycles/op overhead in LoadA, and 29 Kcycles/op in RunC compared to *Tebis*. When both are enabled, the overhead is 94 Kcycles/op in LoadA, and 90 Kcycles/op in RunC compared to native *Tebis*. Finally, Fennec, which protects

data in all three phases, takes 549 Kcycles to complete a write operation, and 518 Kcycles to complete a read operation, which is 25× ∼ 27× times worse compared to *Tebis*.

5.2.2 Performance Breakdown Analysis

In this section, we analyse the performance of Fennec and assess the performance costs of each of its component individually (e.g., TEE execution, data encryption, hashing).



(a) Throughput breakdown

(b) Cycles per operation breakdown

Figure 5.3: The performance breakdown for Fennec when running LoadA and RunC.

Figure 5.3a shows the throughput results for LoadA and RunC. By running the system inside the TEE and using TLS sockets for the client/server communication, but without performing data encryption and HMAC calculations, Fennec achieves a throughput of 26314 ops/sec in LoadA, and 27172 ops/sec in RunC. Compared to *Tebis+TLS*, the slowdown is 4.37× for LoadA, and 4.84× for RunC, with the only difference between them being running inside the TEE. This demonstrates the impact on performance when running inside a TEE. We then enable encryption for all the data that leaves from the EPC to the host memory or the storage device, but keep HMAC calculation disabled. This causes a further 1.47× slowdown for writes and 1.37× for reads. For writes, the slowdown is caused by the server needing to encrypt every KV pair to insert it into L0 and the recovery log. Also, during compaction, the server needs to decrypt both levels to merge them and then re-encrypt the resulting level with a new key. For reads, the main overhead is that the index is encrypted; in order to perform a lookup, the system must decrypt the appropriate index entry and then compare it to the search key. The Fennec configuration introduces HMAC in every KV pair and log entry. Compared to the

configuration without HMAC calculation, the slowdown is 1.39× for writes and 1.43× for reads. The system has to calculate a hash for every KV pair before encrypting it and verify the correctness of the HMAC after decrypting, which is what is causing this slowdown.

Figure 5.3b shows the cycles per operation for LoadA and RunC. The configuration running in the TEE without disk encryption and HMAC calculation takes 268 KCycles to complete an operation in LoadA and 260 KCycles to complete an operation in RunC. Again, this reveals the impact on performance that running inside the TEE has, prior to adding any storage device encryption, as the operations need 4.8× $\sim$ 5× more cycles than *Tebis+TLS* to complete. Enabling data encryption for everything that leaves the EPC, causes a 47% increase in Cycles/Op for LoadA and a 39% increase for RunC compared to the configuration without any disk encryption. Finally, enabling HMAC calculation causes a further 39% $\sim$ 43% increase in cycles per operation.

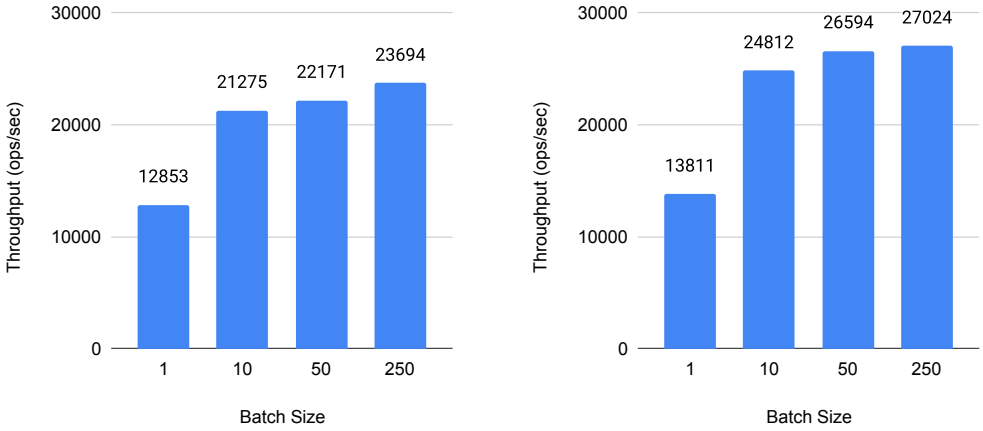
5.2.3 Batched Operations

TEE exits and re-entries have been shown to cause a significant overhead, mainly due to TLB flushes and LLC pollution [14]. Exiting the TEE happens because of system calls. In our system, every time the server receives a request from the client or sends the response to a request back to the client, it has to exit the TEE. In order to measure how much this affects the performance, we modify both the server and the client to batch multiple requests and responses in one TCP request. The batching versions of Fennec offer the same security guarantees as Fennec, only now they receive multiple requests in one buffer and process them all in order, one by one, before responding to the client.

We run the same set of experiments on the same machine as before. Figures 5.4a and 5.4b show the results for LoadA and RunC, respectively. The figures show how the throughput improves as the number of requests in one batch increase. We observe that while the performance improves by batching more requests, the performance gains are marginal after a certain point. This is because of other overheads in the system. For PUT requests, writing to the log, and compactions which read from the storage device and then write to it, still cause TEE exits in order to perform I/O. For GET requests, when reading from the device using mmap, page faults occur which again cause TEE exits. Nonetheless, batching improves Fennec's performance by up to 1.84× for writes, and 1.96× for reads, which reduces the total slowdown of Fennec compared to native Tebis to 6.6× for both reads and writes.

5.2.4 Memory Requirements of Full-leveling and Incremental Compactions

As we mentioned in Section 3.4.3, the full-leveling compactions allow us to use fewer encryption keys compared to incremental compactions, since only one



(a) LoadA (b) RunC

Figure 5.4: The throughput for Fennec with different batch sizes.

key is required per level (instead of one key per SSTable). As a result, keeping all previous encryption keys used (so that they can be queried every time a new encryption key is generated and avoid duplicates), requires much less space. To exemplify the space overheads, we simulate two systems, one with full-leveling compactions and one with incremental and measure the difference in encryption keys used over time. We assume a KVS of a specific size (4TB, 8TB, 16TB, and 32TB) that will be overwritten 1000 times to perform the calculation. The L0 for the simulation is set to 64M and the growth factor to 8. Table 5.1 presents the results regarding the number of compactions of each configuration. We can see that with incremental compactions the system performs 4.7× ∼ 5.5× times more compactions.

KVS Size (TB)	Compactions (M)	
	Full-Leveling	Incremental
4	75	350
8	150	744
16	300	1530
32	599	3328

Table 5.1: The number of compactions performed by the two different compaction policies for a KVS of fixed size overwritten 1000 times.

The impact of this becomes even greater if we consider that every incremental compaction creates on average 9 new SSTables while full-leveling compactions create one new level. Table 5.2 shows the difference in the total number of keys used by the two compaction policies. By leveraging full-leveling compactions we manage to reduce the number of encryption keys by 42× ∼ 50× times.

KVS Size (TB)	Encryption Keys Used (M)	
	Full-Leveling	Incremental
4	75	3154
8	150	6693
16	300	13771
32	599	29949

Table 5.2: The number of keys used by the two different compaction policies for a KVS of fixed size overwritten 1000 times.

An appropriate data structure for keeping track of the history of encryption keys would be a bloom filter. The reason is that we mainly care about never re-using the same encryption key, false positives are not an issue for our use-case. State-of-the-art bloom filters can achieve great accuracy with only 7 bits per key. Table 5.3 shows the memory needed by each of the two policies to maintain the history of encryption keys.

For the full-leveling compactions, for a KVS of size 4TB and 8TB, such a data structure can still fit in the EPC which is usually 128MB, and for newer versions of SGX where the EPC is 256MB, it would fit for the 16TB KVS as well. In contrast, this data structure is more than an order of magnitude larger than the EPC with incremental compactions.

KVS Size (TB)	Memory Usage (MB)	
	Full-Leveling	Incremental
4	62	2632
8	125	5585
16	250	11491
32	500	24991

Table 5.3: The memory needed to store the history of keys using a bloom filter by the two different compaction policies for a KVS of fixed size overwritten 1000 times.

Chapter 6

Related Work

There are various frameworks that are designed to allow applications to run in TEEs without any modifications. Some of them are Open Enclave [13], SCONE [1], Graphene-SGX [5], Occlum [21], and SGX-LKL [17]. As we mentioned before, Fennec is implemented on top of Open Enclave and specifically EdgelessRT [22], which offers support for more libc functions. This enables our system to run on various TEEs, as Open Enclave is a hardware-agnostic framework.

Speicher [2] and Tweezer [10] are the two works most closely related to ours. Speicher was the first secure key-value store system implemented. Speicher adapts RocksDB [8] to run efficiently in a TEE. It calculates a MAC for each data block and then builds a Merkle tree for authentication. It also reduces EPC usage by moving values outside the EPC and into the host's memory with cryptographic protection. Speicher uses user-level I/O through an Intel SPDK-based Direct I/O library to reduce TEE exits, which negatively affect performance.

Tweezer improves Speicher's performance based on a key observation: the SSTables are sorted, immutable, and contain unique keys. By leveraging this, Tweezer uses a per-SSTable authentication scheme and encrypts each key-value pair separately. The per-SSTable authentication scheme allows Tweezer to avoid using a Merkle tree, which helps with performance. However, having a unique encryption key per SSTable does not scale well because as the KV store keeps running, an increasingly large history of encryption keys must be stored in memory and queried every time a new encryption key is created. Our work improves on Tweezer by leveraging full-leveling compactions to reduce the number of encryption keys used by the KVS.

ShieldStore [11] and EnclaveDB [18] are both in-memory storage systems that run in TEEs. ShieldStore is a system for secure in-memory key-value storage using Intel SGX enclaves. It provides confidentiality and integrity guarantees for stored data and supports various privacy-preserving protocols. Like Speicher, it relies on Merkle trees for freshness guarantees. EnclaveDB is a shielded in-memory SQL database. EnclaveDB only uses the storage device for logging and does not provide any freshness guarantees about the data on it. Our work provides security

guarantees for confidentiality, integrity and freshness that extends to the storage device as well.

There are approaches that aim to provide confidentiality in untrusted environments without TEEs. CryptDB [16] is an encrypted database that provides confidentiality guarantees for untrusted hardware but does not guarantee integrity and freshness. It handles unmodified database queries and uses encryption schemes such as OPE to perform computation on encrypted data.

Chapter 7

Conclusions

This thesis presented Fennec, a secure LSM-based key-value store designed to operate within Intel SGX enclaves. By employing per-level encryption keys, HMACs, and a log protection mechanism, Fennec effectively addresses challenges related to data confidentiality, integrity, and recoverability in untrusted environments. Our evaluation results demonstrate that Fennec achieves strong security guarantees with an acceptable performance overhead of 6.6× compared to native Tebis, making it a promising solution for protecting sensitive data in various applications.

There are various improvements that can be explored in the future. One of them is re-enabling bloom filters to avoid the expensive traversal of levels for lookups. Searching for a key involves multiple decryptions which can be avoided with the use of bloom filters. We need to adjust them to work for encrypted keys, and also make sure they are not vulnerable to attacks. Another improvement we can explore is having an exitless service performing I/O. As we mentioned many times, TEE exits are expensive. By having a dedicated process outside of the TEE performing the I/O with a shared buffer, we can reduce this overhead transparently to the application, unlike batched operations for example. Finally, we can investigate whether having the L0 or a part of it in the EPC, can be beneficial. We opted to keep it in the host memory since EPC memory pressure is a major cause of overhead in TEE applications. However, moving the L0 to the EPC can have some benefits. The biggest one is that we won't need to decrypt the KV pairs of L0 to perform compactions from L0 to L1. Since the most compactions are from L0 to L1, this can improve performance, depending on how much the EPC memory pressure affects the performance.

Bibliography

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[2] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. SPEICHER: Securing LSM-based Key-Value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 173–190, Boston, MA, February 2019. USENIX Association.

[3] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[4] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[5] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[7] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications:

The RocksDB experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49. USENIX Association, February 2021.

[8] Facebook. rocksdb. `https://rocksdb.org/`. Accessed: May, 2024.

[9] Google. Asylo. `https://github.com/google/asylo`. Accessed: May, 2024.

[10] Igjae Kim, J. Hyun Kim, Minu Chung, HyunGon Moon, and Sam H. Noh. A Log-Structured merge tree-aware message authentication scheme for persistent Key-Value stores. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 363–380, Santa Clara, CA, February 2022. USENIX Association.

[11] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Mbed-TLS. mbedtls. `https://github.com/Mbed-TLS/mbedtls`. Accessed: May, 2024.

[13] Microsoft. Open enclave SDK. `https://github.com/openenclave/openenclave`. Accessed: May, 2024.

[14] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 238–253, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, jun 1996.

[16] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: processing queries on an encrypted database. *Commun. ACM*, 55(9):103–111, sep 2012.

[17] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter R. Pietzuch. SGX-LKL: securing the host OS interface for trusted execution. *CoRR*, abs/1908.11143, 2019.

[18] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278, 2018.

[19] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. Slimdb: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment*, 10(13):2037–2048, 2017.

[20] Moritz Schneider, Ramya Jayaram Masti, Shweta Shinde, Srdjan Capkun, and Ronald Perez. Sok: Hardware-supported trusted execution environments, 2022.

[21] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 955–970, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Edgeless Systems. Edgeless RT. `https://github.com/openenclave/openenclave`. Accessed: May, 2024.

[23] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tebis: index shipping for efficient replication in lsm key-value stores. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 85–98, New York, NY, USA, 2022. Association for Computing Machinery.

[24] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Exploring the feasibility of fully homomorphic encryption. *IEEE Transactions on Computers*, 64(3):698–706, 2015.

[25] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. Parallax: Hybrid key-value placement in lsm-based key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 305–318, New York, NY, USA, 2021. Association for Computing Machinery.