Authenticated Key-Value Stores with Hardware Enclaves

Kai Li
Syracuse University
Syracuse, NY, USA
kli111@syr.edu

Yuzhe Tang
Syracuse University
Syracuse, NY, USA
ytang100@syr.edu

Qi Zhang
IBM Research T.J. Watson
Yorktown Heights, NY, USA
Q.Zhang@ibm.com

Jianliang Xu
Hong Kong Baptist University
Kowloon Tong, Hong Kong SAR
xujl@comp.hkbu.edu.hk

Ju Chen
Syracuse University
Syracuse, NY, USA
jchen133@syr.edu

Abstract

Authenticated data storage on an untrusted platform is an important computing paradigm for cloud applications ranging from data outsourcing, to cryptocurrency and general transparency logs. These modern applications increasingly feature update-intensive workloads, whereas existing authenticated data structures (ADSs) designed with in-place updates are inefficient to handle such workloads. This work addresses the issue and presents a novel authenticated log-structured merge tree (eLSM) based key-value store built on Intel SGX.

We present a system design that runs the code of eLSM store inside enclave. To circumvent the limited enclave memory (128 MB with the latest Intel CPUs), we propose to place the memory buffer of the eLSM store outside the enclave and protect the buffer using a new authenticated data structure by digesting individual LSM-tree levels. We design protocols to support data integrity, (range) query completeness, and freshness. Our protocol causes small proofs by including the Merkle proofs at selected levels.

We implement eLSM on top of Google LevelDB and Facebook RocksDB with minimal code change and performance interference. We evaluate the performance of eLSM under the YCSB workload benchmark and show a performance advantage of up to $4.5X$ speedup.

CCS Concepts: • **Security and privacy → Distributed systems security**; **Trusted computing**.

Keywords: Data integrity, Query authentication, Storage consistency, Key-value stores, LSM trees, SGX, Enclave.

Middleware '21 Industrial Track, December 6–10, 2021, Virtual Event, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9152-8/21/12. . . $15.00
https://doi.org/10.1145/3491084.3491425

ACM Reference Format:
Kai Li, Yuzhe Tang, Qi Zhang, Jianliang Xu, and Ju Chen. 2021. Authenticated Key-Value Stores with Hardware Enclaves. In *22nd International Middleware Conference Industrial Track (Middleware '21 Industrial Track), December 6–10, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3491084.3491425

1 Introduction

Outsourcing data storage to a third-party host, such as the public cloud, has been an emerging practice with increasing popularity in security-critical applications. For instance, using Amazon S3 to store the Bitcoin ledger or transaction history or to host general transparency logs (e.g., in Google Certificate Transparency [3, 25]) can significantly bring down the operational cost and are adopted in practice [12]. The modern security applications features user-generated content continuously generated in an intensive data-write stream.

Data authenticity is the primary concern for storing and serving data on the untrusted cloud services which are constantly caught compromised in the real world. To guarantee data authenticity, a common approach is to employ the protocols of authenticated data structures (ADSs) between an untrusted cloud server and trusted clients (i.e., data owner and query users). However, existing ADS schemes [21, 22, 24, 28, 29, 31, 35, 37, 42, 43] have several major limitations. First, they are designed based on update-in-place data structures (i.e., requiring excessive communication and large proofs between the data owner and the cloud for updating the ADS), leading to known inefficiency problems in handling data updates [30, 33]. Second, existing schemes require the users to verify the proof of results obtained from the cloud, thus incurring high bandwidth and computation overheads.

To address these limitations, in this paper, we leverage off-the-shelf hardware enclaves, in particular Intel Software Guard eXtension (SGX) [10], and propose a novel authenticated key-value store based on LSM trees. The motivation of our design is two-fold: 1. (Why LSM tree?) An LSM tree (log-structured merge tree) is a data structure that supports append-only writes and random-access data reads. Periodically, it conducts a batch operation, called COMPACTION, that reorganizes the data layout for better read performance in the future.

By this design, an LSM tree has performance advantages in serving high-speed write streams and is widely adopted as the external-memory index structure in modern storage systems including Google's BigTable [19]/LevelDB [8], Facebook RocksDB [7], Apache HBase [2], Apache Cassandra [1]. 2. (Why Intel SGX?) Without a trusted party, realizing an authenticated LSM tree on the cloud requires using costly cryptographic protocols, such as verifiable computations (VC [15, 18, 32, 34, 41]), to support verifiable COMPACTION.[1] With the advent of commercial trusted execution environments, notably Intel Software Guard eXtension or SGX [10], it becomes practically feasible to build a trusted execution environment or enclave in proximity to the untrusted cloud platform. This makes bulk data transfer viable and is promising to support verifiable and efficient COMPACTION for authenticated LSM trees. Besides, query users are alleviated from the burden of result verification. Note that our trusted design is specific to Intel SGX and should be differentiated from blockchain-based hardening of storage systems [23, 38, 39].

In our envisioned architecture, trusted cloud applications (e.g., a database server) run inside SGX enclaves and issue data read/write requests to our authenticated key-value store that is co-located in the cloud. In our system design, the code (namely the codebase of a vanilla LSM store[2]) is placed inside the enclave, which relies on the Intel SGX SDK [11] or an in-enclave Library OS (LibOS) [17, 40] to handle system calls. In terms of placing data (e.g., program states), a naive design is to store the data in the memory region inside the enclave. When handling data of Gigabytes, this design, however, imposes huge memory pressure inside enclave and would cause significant performance slowdown. To be more specific, the current family of Intel CPUs support 128 MB physical memory in the enclave, and when the enclave memory hosts more than 128 MB, it causes expensive enclave paging [27], leading to performance slowdown. More fundamentally, the slowdown is inevitably caused by the security needs to protect enclave memory. Even if Intel may remove the hard memory limits of 128 MB in future releases, putting data in a enclave incurs slowdown caused by memory protection.

To circumvent the inefficiency, we propose to place the memory data outside the enclave. More precisely, among various memory data structures in an LSM store, we place the read buffer outside the enclave and leave other structures that often grow sublinearly with the data size inside the enclave, including index structures (e.g., a bloom filter), write buffer, etc. To ensure the integrity of the data placed outside the enclave, we propose an authenticated LSM tree, named by eLSM. eLSM builds a forest of Merkle trees, each digesting a "level" in an LSM tree. eLSM supports efficient reads and small-sized query proofs by presenting Merkle proofs at selected levels. We proved the security of the query authentication schemes in eLSM (deferred in technical report [36]).

We also build the eLSM systems on Google LevelDB [8] and Facebook RocksDB [7]. In our systems, the eLSM Merkle proofs are embedded in individual data records in such a way that the proof of a query can be naturally constructed from the Merkle proofs embedded in the data records included in the query result. By this means, we minimize the code change needed in Google LevelDB, reducing performance interference at runtime. For RocksDB, the eLSM system is implemented as a middleware that does not require code change of the underlying RocksDB, but instead just relies on its callback interface [4]. More specifically, we implement authenticated COMPACTION in some event handlers in the COMPACTION path of RocksDB. With this, we believe the add-on design of eLSM is generally applicable to any LSM stores. By contrast, existing work in the field, notably Speicher [16], all requires significant code change of underlying LSM store. At last, the code of eLSM is open-source [5].

We conduct a comprehensive performance study of eLSM under the YCSB workload benchmark [20]. The performance result shows that eLSM achieves lower operation latency than the baseline of update-in-place data structures by more than one order of magnitudes. Compared with in-enclave memory buffers, the design of data buffer outside the enclave achieves up to $4.5X$ speedup in most YCSB workloads.

The contributions made in this paper include the following:

1. This work addresses an emerging security workload, that is, supporting query authentication in the presence of frequent data updates. We propose a novel SGX-enabled authenticated key-value store.

2. We present the system designs of eLSM that are secure, efficient and generic. It places the memory data outside the enclave to circumvent the limited memory size in the enclave. It builds an authenticated LSM tree with small query proofs at selected tree levels. To the best of our knowledge, eLSM is the first data-authentication middleware on LSM stores, without any code change of the underlying store.

3. We implemented functional prototypes of eLSM on Google LevelDB and Facebook RocksDB. The code of our prototype is open-sourced [5]. We conducted a comprehensive performance study under the YCSB workload benchmark that shows up to $4.5X$ performance advantage.

2 eLSM-P2 System

2.1 System Design & Overview

Baseline design: We consider a strawman design of the system, named eLSM-P1, that places the entire user-space codebase of an LSM store inside the enclave. The data files, named SSTables, are stored outside the enclave. The interaction between the enclave and the untrusted host occurs at the syscall levels, primarily for file management (e.g., `fwrite`

[1]Another possible approach is to transfer the whole dataset back to trusted data owners over the Internet, which is also expensive.

[2]"LSM store" denotes the class of key-value stores designed based on LSM trees.

and `fread`). Specifically, eLSM-P1 places outside the enclave the files at all LSM-tree levels, including the WAL file at level L_0 and data files at levels $L_{\geq 1}$. Inside the enclave are the codebase of an LSM store and metadata including indices and data buffers.

Overview: Figure 1a depicts the system architecture of eLSM-P2 which runs the code for operations PUT,GET,COMPACTION inside enclave. The memory data including the buffer at Level L_0 and file indices at Levels $L_{\geq 1}$ are also placed inside the enclave. The read buffers and all data files at Levels $L_{\geq 1}$ are placed outside the enclave. The WAL file is also stored outside enclave. The figure also illustrates the dataset in the LSM store, which we will use throughout this section to describe the details of eLSM-P2 system. In this example dataset, there is an LSM tree of three levels and six key-value records. Level L_1 contains record $\langle A, 9\rangle$, level L_2 contains three records $\langle T, 4\rangle$, $\langle Z, 7\rangle$, $\langle Z, 6\rangle$ and level L_3 contains four records $\langle A, 2\rangle$, $\langle T, 0\rangle$, $\langle Y, 3\rangle$, $\langle Z, 1\rangle$. Here, we show the key-value record by its data key and timestamp. Record $\langle T, 0\rangle$ is the oldest and is of key T and timestamp 0. For simplicity, the data is omitted in the example.

Protecting data outside enclave: Because eLSM-P2 places outside enclave the data at non-zero levels, it entails data protection mechanisms. For data confidentiality, we require the data key in each record to be encrypted with deterministic encryption (DE), such that it can directly search the domain of ciphertext. We discuss data confidentiality in technical report [36]. Data authenticity is handled by the eLSM digest structure, as described next.

2.2 Digest Structure

To digest an LSM tree, we propose a novel authenticated data structure, eLSM-P2 digests. There are two key designs: 1) eLSM-P2 builds a "forest" of Merkle trees, each digesting one LSM-tree level and each having its root stored in the enclave. 2) In a per-level Merkle tree, data records of the same key are digested in hash chains and records of different keys are digested in a Merkle tree. In particular, the hash chain is built in a temporal order where the chain header is the oldest record and the tail is the newest record. In Figure 1a, each of the three LSM tree levels is associated with a Merkle tree. Case 1): For a level of distinct data keys, such as level L_3 in Figure 1a, it builds the leaf set of the Merkle tree directly on the data records. For instance, $h_7 = H(\langle A, 2\rangle)$ (H is a standard cryptographic hash algorithm with variable-length input) and $h_6 = H(\langle T, 0\rangle)$. An intermediate node is the hash digest of the concatenation of the two children, for instance, $h_8 = H(h_6 \| h_7)$ (here, $\|$ is a concatenation operation). Case 2) For a level that contains some records of the same keys, it constructs a hash chain over these records. For instance, level L_2 contains two records of the same key, $\langle Z, 7\rangle$ and $\langle Z, 6\rangle$. eLSM-P2 builds a hash chain on these two records, that is, $h_4 = H(\langle Z, 7\rangle \| H(\langle Z, 6\rangle))$. Then, it builds the Merkle tree over h_4 (for records of key Z) and h_2 (for record $\langle T, 4\rangle$) for level L_2.

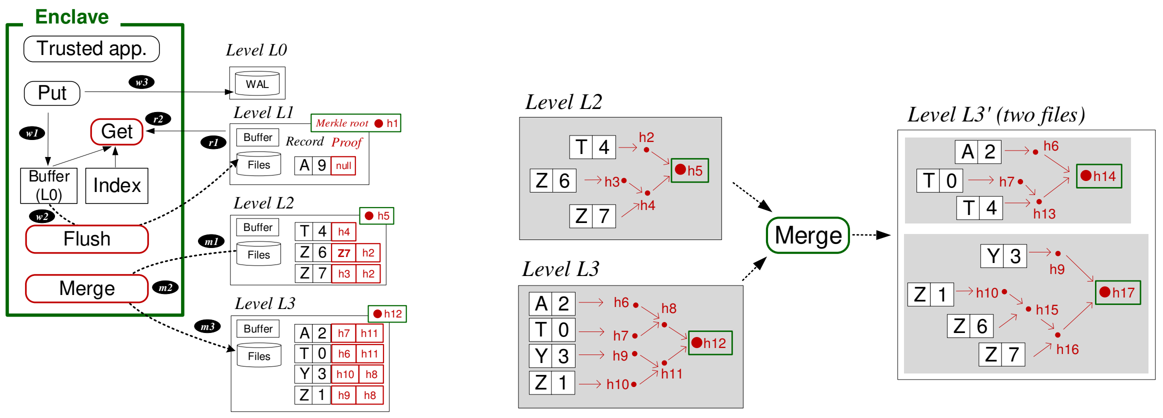
To materialize the eLSM-P2 digest structure, we present a simple storage design: Given a level L_i and its Merkle tree, each record at the level $\langle k, v\rangle$ is augmented with its eLSM-P2 proof π, that is, $\langle k, v \| \pi_i\rangle$. Given a record, an eLSM-P2 proof is the set of Merkle tree nodes that surround the path from the leaf node of the record to the root node. For instance, in Figure 1a, the eLSM-P2 proof for record $\langle A, 2\rangle$ consists of hashes h_7 and h_{11} (i.e., the siblings to nodes h_6 and h_8).

2.3 Read/Write Protocol

Data read path starts with the trusted application issuing a read operation, $v = \text{GET}(k)$. The enclave looks up its index to locate the target level and file, and it then notifies the eLSM-P2 store. r1 The untrusted store serves the read operation on the target file and, in addition, runs algorithm $\pi, v = \text{QUERY}_{\text{GET}}(m, k)$ to prepare a proof for authenticating the read result (m is the key-value dataset). The proof consists of Merkle authentication paths or Merkle proofs [26] at "relevant" LSM tree levels. Recall that a Merkle authentication path consists of the hashes surrounding the path from a leaf to the root in a Merkle tree and it can be used to verify the membership and non-membership of a record in a dataset. r2 The eLSM-P2 store in the untrusted host then sends the result of GET (k) as well as the proofs (π) to the enclave. The enclave verifies the authenticity of the read result, by running algorithm Yes|No= $\text{VRFY}_{\text{GET}}(\pi, v)$. The verification algorithm iterates through relevant levels, and, for each level, verifies the (non)-membership of the queried data key (k) using the Merkle proof (in π) and the locally stored root hash.

To verify the non-membership of a key on a level, it commonly returns the two Merkle authentication paths that surrounds the key. If the queried key is smaller than the smallest key on a level, it simply returns the single Merkle proof that would authenticate the membership of the leftmost key. For instance, consider authenticating the non-membership of a queried key B on level L2 in Figure 1b. The proof is $\langle T, 4\rangle$, $h4$, with which and root $h5$ the enclave would verify that $\langle T, 4\rangle$ is the smallest key on the level, thus queried key B does not exist on Level L2. Note that our in-enclave compaction (will be described) ensures the invariant that data is sorted by key at the same level.

A strawman of designing the eLSM-P2 proof is to scan all levels to prepare a proof (in algorithm QUERY). We propose to reduce the proof size by including only Merkle proofs of the levels no higher than the level of the result record. This will allow algorithm QUERY to stop early when it reaches the first level, say L_i, that finds a matching record. The returned proof $\pi = \pi_1, \ldots \pi_i$, where $\pi_1, \ldots \pi_{i-1}$ are the Merkle proofs for non-membership (there is not any matching record at levels $L_1, L_2, \ldots L_{i-1}$). π_i is the Merkle proof for membership (there is a matching record at level L_i). All Merkle proofs after level L_i, as will be seen, do not contain fresher records and



(a) System architecture

(b) Authenticated COMPACTION with Merkle trees: It merges two data files (SSTables) at Levels L_2 and L_3 into two data files (SSTables) at Level L_3. Each SSTable file is represented by a gray box.

Figure 1. eLSM-P2 system with an LSM tree of three levels: A rectangle depicts data and a rounded rectangle depicts code. Shapes in red are where eLSM-P2 makes code change over the original LSM store. The root hashes (red dots in green boxes) are maintained with copies inside enclave.

are deliberately omitted. When there is no matching record, $i = q$.

An example: In Figure 1a, suppose the trusted application issues GET(Z) over dataset m. In step r1, the untrusted host serves QUERY$_{\text{GET}}(m, Z)$ with authentic result $\langle Z, 7\rangle$ (the benign case). $\langle Z, 7\rangle$ is the newest record matching queried key Z and is located at level L_2. The proof is two Merkle authentication paths at levels L_1 and L_2. Note that there is no need to include level L_3 in the eLSM-P2 proof. Concretely, the proof at the first level is $\langle A, 9\rangle$ (denoted by π_1). The proof at the second level is h_3, h_2 (denoted by π_2). Then in step r2, the enclave can verify the result authenticity in freshness and completeness based on the proof $\pi = [\pi_1, \pi_2]$ (i.e., algorithm VRFY($[\pi_1, \pi_2], \langle Z, 7\rangle, [h_1, h_5]$)).

Concretely, with the first-level proof $\pi_1 = \langle A, 9\rangle$, the enclave verifies result authenticity by checking $H(\pi_1) \overset{?}{=} h_1$. If the VRFY algorithm runs through, it authenticates the fact that record $\langle A, 9\rangle$ is the only record at level L_1. From this, it can be derived that level L_1 does not contain any record of key Z (i.e., the non-membership of a data key Z at level L_1). With the second-level proof $\pi_2 = h_3, h_2$, the enclave verifies by checking $H(h_2 \| H(\langle Z, 7\rangle \| h_3)) \overset{?}{=} h_5$. If successful, it authenticates the fact that a) record $\langle Z, 7\rangle$ is a valid record at level L_2 (result integrity), b) record $\langle Z, 7\rangle$ is the newest record with key Z (result freshness). Fact b) is based on that there are no other records of key Z in the proof π_2. Based on these two proofs, one can establish that record $\langle Z, 7\rangle$ is the newest record in the dataset m.

Consider the malicious case when the untrusted host can return a stale record, say $\langle Z, 6\rangle$, to the enclave. In this case, the malicious host can only present the following as a valid level-L_2 proof, that is, $\pi_2' = \langle Z, 7\rangle, h_2$. By this means, the enclave can verify the result integrity successfully by checking $H(h_2 \| H(\langle Z, 7\rangle \| H(\langle Z, 6\rangle))) \overset{?}{=} h_5$. However, as the newer result record $\langle Z, 7\rangle$ has to be included in the proof π_2, the enclave can detect that $\langle Z, 6\rangle$ is not the most fresh record (violating freshness).

Data write path starts with the trusted application issuing a write operation PUT(k, v). To serve the write, the enclave maintains two in-enclave structures, a write buffer of level L_0 and, for data recovery, a digest of the write-ahead log (WAL). Recall that a WAL stores recent data writes in temporal order and serves as the base to recover recent data in the case of fault. The storage of WAL is placed outside the enclave, while the enclave stores the hash digests of the WAL.

w1 Serving the write PUT (k,v), the enclave first assigns to the record to write the latest timestamp ts. It then writes to the memory buffer of level L_0 inside enclave. Serving a timestamped write PUT(k, v, ts), the enclave iteratively update its WAL digest by $dig' = H(dig \| \langle k, v, ts\rangle)$. w2 When the write buffer at level L_0 overflows, it is triggered to flush the content at Level L_0 and to generate a file at Level L_1. In the system of an LSM store, the codebase for flush is shared with that for COMPACTION. w3 The enclave switches out to append the write to the WAL in the untrusted domain. Enclave WAL can be extended to defend rollback attacks, as discussed in technical report [36].

An example: In Figure 1a, suppose the application calls PUT(Y). The enclave assigns to the record the latest timestamp 10. It updates the WAL digest from dig to dig', such that $dig' = H(dig \| \langle Y, 10\rangle)$) (w1). The host appends the record to the WAL outside enclave (w3). If the buffer of Level L_0 is overflown by the new record, it will sort all records stored in L_0, and flush them to a new file at L_1 (w2).

Compaction path starts with the trusted application in enclave issuing operation $(L'_i, L'_{i+1}) = \text{Compaction}(L_i, L_{i+1})$. For simplicity, we consider the most basic form of Compaction, namely, merging two adjacent levels. It is natural to extend it to more complicated cases such as merging more than two levels or merging subsets at the two levels. For the Compaction across two levels, eLSM-P2 carries out the computation inside enclave and only switches the execution outside enclave for file access. The process runs in the following steps: m1 the enclave starts to issue OCalls to load all input files to untrusted memory so that the enclave can read the streams of data records (in their sorted order). m2 The enclave then runs "authenticated Compaction" that merges input data at the two levels into one level. Internally, the enclave needs to verify the authenticity of input data, to conduct the actual computation for Compaction, to produce the digest of output data, and to generate the proofs embedded in the output data. m3 The untrusted host makes effect of the Compaction by flushing merged data and proof to disk. The enclave updates the per-level digests by the newly produced ones.

An example: In Figure 1a, suppose the application calls $\text{Compaction}(L_2, L_3)$. In step m1, the host loads the data at the two levels from disk to memory (in the untrusted world). In step m2, the enclave verifies the data authenticity of input levels by reconstructing the Merkle tree at level L_2 (and L_3) and by checking if its root hash is equal with h_5 (and h_{12}). It will then merge the two levels' data into one merged list, that is, from $L_2 = [\langle T,4\rangle, \langle Z,7\rangle, \langle Z,6\rangle]$ and $L_3 = [\langle A,2\rangle, \langle T,0\rangle, \langle Y,3\rangle, \langle Z,1\rangle]$ to output level $L'_3 = [\langle A,2\rangle, \langle T,4\rangle, \langle T,0\rangle, \langle Y,3\rangle, \langle Z,7\rangle, \langle Z,6\rangle, \langle Z,1\rangle]$. Meanwhile, it builds the Merkle tree over the output list, and based on it, generates the proofs embedded in data records. In step m3, the digest of the new Merkle tree replaces that of level L_3 (i.e., h_{12}). L_2 becomes an empty list, which also updates its digest. We defer the security protocol analysis to technical report [36].

2.4 System Implementation

We have implemented eLSM-P2 on Google LevelDB [8] and Facebook RocksDB [7]. For LevelDB, eLSM is implemented by directly changing the LevelDB codebase. This implementation approach has the advantage in performance. For RocksDB, eLSM is implemented as a RocksDB add-on, that is, without code changes to RocksDB. In this subsection, we present the RocksDB implementation in § 2.4.1.

In the protocol of eLSM-P2, a key-value record is stored with its proof, that is, $\langle k, v \| \pi_i \rangle$, where i is the index of the level where the record is currently located. To implement the embedded proof in an LSM store, it is required to add the code change in two paths, that is, a) the Compaction paths for updating records' proof when they are merged to a different level (Note that the proof is sensitive to which level the record is located), and b) the Get path where the proof is used to authenticate the record membership/non-membership in a level.

2.4.1 Implementation as a RocksDB Add-on.

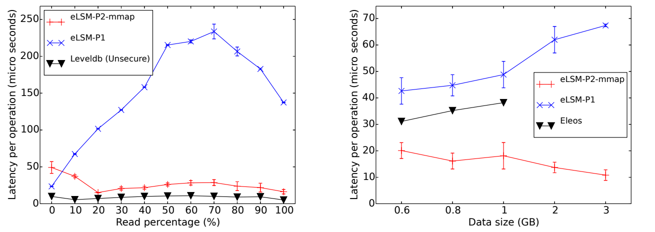
The eLSM-P2 implementation on LevelDB requires the change of LevelDB's codebase. A more modular approach (with benefits in easy maintenance and deployment, etc.) is to implement eLSM-P2 without the code change of underlying LSM stores. For this reason, we present the second implementation on RocksDB.

Comparing with LevelDB, the system of RocksDB exposes callbacks through which application programs can listen to and handle RocksDB's internal events. Different RocksDB events exposed occur at the granularity of level, file, record, etc. The callback API is similar to stored procedures (supported in commercial database systems) and is widely available in other LSM stores (e.g., HBase Coprocessor [9]).

The RocksDB-based eLSM is implemented as a series of event handlers. 1) As previously described, the authenticated Compaction is implemented as handlers for events `Filter` and `OnTableFileCreated`. The former event is triggered every time the underlying RocksDB encounters a key-value record during a Compaction, and the latter is when a new file is written to the disk. 2) The embedded proof on RocksDB is extended to cover not only the Merkle proof at the current file, but also all Merkle proofs at the current level and previous non-hit levels. The Merkle proofs at non-hit levels are for the non-membership. 3) To implement the authenticated flush, it wraps the code for digesting a MemTable in the iterator (i.e., `next()`) exposed by a pluggable MemTable [13].

3 Performance Evaluation

This section presents performance evaluation of eLSM-P2 and eLSM-P1 under YCSB benchmarks [20].



(a) Varying read-write ratio **(b)** Varying data size (workload A)

Figure 2. Performance of eLSM-P2 and eLSM-P1 under YCSB workloads

Experiment setup: In our experiments, we use a laptop equipped with an SGX CPU. Specifically, the hardware specs include an Intel 8-core i7-6820HK CPU of 2.70 GHz with 8 MB cache, a 16 GB RAM and 1 TB disk.

We run our experiments in the YCSB framework [20]. We use YCSB to both generate the workload and to execute the experiments. YCSB framework works in two phases: the load phase when it initializes the system by populating the dataset, and the evaluation phase when it drives the target workload to the system and measures the performance. When initializing each experiment, we typically scan the loaded dataset so that it is loaded in the untrusted memory. By this means, we mainly consider the setting of memory-resident data with size ranging from several hundreds of megabytes to four gigabytes (that is, millions of records, each with a 16-byte key and 100-byte value by default). With such small record size, four gigabytes is the maximal data size that we can tolerate in experiment time.

We port the open-source LevelDB-YCSB adapter [14] to the SGX architecture. This is done by running the YCSB platform in the untrusted world and wrap each PUT/GET request as an ECall (as in SGX SDK) into the enclave. Inside the enclave, we run the YCSB measurement code that measures various performance metrics.

Baseline: Eleos: We implement a baseline of an in-memory data store. In this in-memory store, the entire dataset is stored in enclave as a sorted array. To make data update efficient, we leave 30% of the array space empty to accommodate data insertions without moving existing data. For implementation, we use Eleos [27], a state-of-the-art virtual memory management engine in enclave without calling expensive enclave paging. Their approach, briefly, is to monitor all memory references and to relocate data between enclave and untrusted memory. We implement a sorted array in enclave linked with Eleos. The array serves data reads with binary search and is updated "in place". For a fair comparison, the data in Eleos is persisted to disk periodically, by using a write buffer to store recent data and switching out of enclave (through an OCall).

Macrobenchmark performance: We present the performance result of eLSM under YCSB. In this set of experiments, we vary the workloads in terms read-write ratio, key distribution, etc. and evaluate eLSM performance. The purpose is to present a holistic view regarding the performance of eLSM-P1 and eLSM-P2.

To conduct the experiments, we fix the initial dataset at 3 GB (or 25 million records). In the evaluation phase, we drive millions of operations to the key-value store for performance measurement. We use the uniform distribution to generate the dataset and queries. We configure the authenticated compaction using default parameters (e.g., in terms of the scope of keys to be compacted). By this means, we conduct a series of experiments with varying the read-write ratio of the workload. Each experiment runs three times and the average performance metric and standard deviation are reported. The operation latency numbers of eLSM-P1, eLSM-P2 and the unsecured LevelDB under varying read-write ratios are shown in Figure 2a. The result shows that eLSM-P2 outperforms eLSM-P1 in most workloads except for a small set of write-only workloads. Specifically, as the workload becomes more read intensive, eLSM-P2 has its operation latency decreased. This performance characteristic is due to that eLSM-P2 has to cause disk IO for data persistence on the write path while on the read path it can read the memory (through the `mmap` files). As the workload transitions from writes to reads, eLSM-P1's latency first increases and then decreases near the end. The increase of latency is caused by overflowing the enclave memory (of 128 MB) and enclave paging. In addition, compared with the ideal approach (running an unsecured LevelDB), the slowdown caused by eLSM-P2 is between $1.5\times$ and $4X$.

Comparing eLSM-P2 and eLSM-P1, when the workload is write-only, eLSM-P1 is faster. For most workloads, eLSM-P2 has a smaller operation latency than eLSM-P1, and the performance discrepancy reaches the highest when the workload consists of 70% reads (Note the uniform key distribution in this workload). In this setting, eLSM-P2 achieves $4.5X$ performance speedup comparing eLSM-P1. This performance result clearly supports the design tradeoff made in eLSM-P1 and eLSM-P2, where eLSM-P2 optimizes the read path by placing the read buffer outside enclave and avoiding enclave paging, which inevitably causes the write overhead, including authenticating COMPACTION and embedding eLSM-P2 proofs in the software layer. eLSM-P1 does not have such write overhead (data security is provided by the hardware-level memory protection in SGX). From the performance result, it can be seen that the eLSM-P2's design to trade off write performance for read is worthwhile, as the majority of workloads favors eLSM-P2.

The second experiment is to report the operation latency under varying data sizes. We initialize the system with data of varying sizes from 0.6 GB (5 million records) to 3 GB (25 million records). In the evaluation phase, we drive into the system YCSB workload A which consists of 50% reads and 50% writes with data keys generated following a Zipfian distribution. We measure the operation latency for eLSM-P2 (in `mmap` configuration), eLSM-P1 and the baseline of Eleos. The result is shown in Figure 2b. With the increasing data sizes, Eleos can scale only to 1 GB data which is limited by their open-source project [6, 27]. The discrepancy between the latency of eLSM-P2 and eLSM-P1 increases, and reaches $7X$ when the data size is 3 GB.

Acknowledgments

The second author thanks Heng Yin, Scott D. Constable and Amin Fallahi for the helpful discussion in early state of this work. The first two authors are partially supported by the National Science Foundation under Grant CNS1815814.

References

[1] [n. d.]. Apache Cassandra, http://cassandra.apache.org/.

[2] [n. d.]. Apache HBase, http://hbase.apache.org/.

[3] [n. d.]. Certificate transparency, the Internet standards, https://tools.ietf.org/html/rfc6962.

[4] [n. d.]. Compaction Filter, https://github.com/facebook/rocksdb/wiki/Compaction-Filter.

[5] [n. d.]. eLSM source code, https://drive.google.com/file/d/1flnP4BlmwwNThPO-7wWqeAKwluUQygkr/view?usp=sharing.

[6] [n. d.]. ExitLess services for SGX enclaves, https://github.com/acsl-technion/eleos.

[7] [n. d.]. Facebook RocksDB, http://rocksdb.org/.

[8] [n. d.]. Google LevelDB, http://code.google.com/p/leveldb/.

[9] [n. d.]. HBase Coprocessor, https://blogs.apache.org/hbase/entry/coprocessor_introduction.

[10] [n. d.]. Intel Corp. Software Guard Extensions programming reference, https://goo.gl/Ka3pnU.

[11] [n. d.]. Intel Software Guard Extensions (Intel SGX) SDK, https://software.intel.com/en-us/sgx-sdk/download.

[12] [n. d.]. Introducing Oak, a Free and Open Certificate Transparency Log (Let's Encrypt), https://bit.ly/2HtgDA0.

[13] [n. d.]. RocksDB Hooks, https://github.com/facebook/rocksdb/wiki/rocksdb-basics.

[14] [n. d.]. YCSB LevelDB adaptor, https://github.com/jtsui/ycsb-leveldb.

[15] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. 1998. Proof Verification and the Hardness of Approximation Problems. J. ACM 45, 3 (1998), 501–555. https://doi.org/10.1145/278298.278306

[16] 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 17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019. 173–190. https://www.usenix.org/conference/fast19/presentation/bailleu

[17] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. ACM Trans. Comput. Syst. 33, 3 (2015), 8:1–8:26. https://doi.org/10.1145/2799647

[18] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. 2013. Verifying computations with state. In ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013. 341–357. https://doi.org/10.1145/2517349.2522733

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In OSDI. 205–218.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In SoCC. 143–154.

[21] Premkumar Devanbu, Michael Gertz, Charles Martel, and Stuart G. Stubblebine. 2003. Authentic Data Publication over the Internet. Journal of Computer Security 11 (2003), 2003.

[22] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In SIGMOD Conference. 121–132.

[23] Kai Li, Yuzhe Tang, Beom Heyn Kim, and Jianliang Xu. 2019. Secure Consistency Verification for Untrusted Cloud Storage by Public Blockchains. SecureComm abs/1904.06626 (2019). arXiv:1904.06626 http://arxiv.org/abs/1904.06626

[24] Charles U. Martel, Glen Nuckolls, Premkumar T. Devanbu, Michael Gertz, April Kwong, and Stuart G. Stubblebine. 2004. A General Model for Authenticated Data Structures. Algorithmica 39, 1 (2004), 21–41.

[25] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. 2015. CONIKS: Bringing Key Transparency to End Users. In 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015., Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 383–398. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara

[26] Ralph C. Merkle. 1980. Protocols for Public Key Cryptosystems. In IEEE Symposium on Security and Privacy. 122–134.

[27] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 238–253. https://doi.org/10.1145/3064176.3064219

[28] HweeHwa Pang and Kian-Lee Tan. 2004. Authenticating Query Results in Edge Computing. In Proceedings of the 20th International Conference on Data Engineering (ICDE '04). IEEE Computer Society, Washington, DC, USA, 560–. http://dl.acm.org/citation.cfm?id=977401.978163

[29] Stavros Papadopoulos, Yin Yang, and Dimitris Papadias. 2007. CADS: Continuous Authentication on Data Streams. In VLDB. 135–146.

[30] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. 2013. Streaming Authenticated Data Structures. In Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings (Lecture Notes in Computer Science), Thomas Johansson and Phong Q. Nguyen (Eds.), Vol. 7881. Springer, 353–370. https://doi.org/10.1007/978-3-642-38348-9_22

[31] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2008. Authenticated hash tables. In Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008, Peng Ning, Paul F. Syverson, and Somesh Jha (Eds.). ACM, 437–448. https://doi.org/10.1145/1455770.1455826

[32] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013. 238–252. https://doi.org/10.1109/SP.2013.47

[33] Yi Qian, Yupeng Zhang, Xi Chen, and Charalampos Papamanthou. 2014. Streaming Authenticated Data Structures: Abstraction and Implementation. In Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014, Gail-Joon Ahn, Alina Oprea, and Reihaneh Safavi-Naini (Eds.). ACM, 129–139. https://doi.org/10.1145/2664168.2664177

[34] Srinath T. V. Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. 2013. Resolving the conflict between generality and plausibility in verified computation. In Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013. 71–84. https://doi.org/10.1145/2465351.2465359

[35] Roberto Tamassia. 2003. Authenticated Data Structures. In Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings. 2–5. https://doi.org/10.1007/

978-3-540-39658-1_2

[36] Yuzhe Tang, Kai Li, Jianliang Xu, Qi Zhang, and Ju Chen. 2019. Authenticated Key-Value Stores with Hardware Enclaves. CoRR abs/1904.12068 (2019). arXiv:1904.12068

[37] Yuzhe Tang, Ting Wang, Ling Liu, Xin Hu, and Jiyong Jang. 2014. Lightweight authentication of freshness in outsourced key-value stores. In Proceedings of the 30th ACSAC 2014, New Orleans, LA, USA, Charles N. Payne Jr., Adam Hahn, Kevin R. B. Butler, and Micah Sherr (Eds.). ACM, 176–185. https://doi.org/10.1145/2664243.2664244

[38] Yuzhe Richard Tang, Kai Li, Yibo Wang, and Sencer Burak Somuncuoglu. 2020. Scalable Log Auditing on Private Blockchains via Lightweight Log-Fork Prevention. In Proceedings of the 4th Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers, SERIAL@Middleware 2020, Delft, The Netherlands, December 07-11, 2020. ACM, 1–4. https://doi.org/10.1145/3429884.3430032

[39] Yuzhe Richard Tang, Zihao Xing, Cheng Xu, Ju Chen, and Jianliang Xu. 2018. Lightweight Blockchain Logging for Data-Intensive Applications. In Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers (Lecture Notes in Computer Science), Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala (Eds.), Vol. 10958. Springer, 308–324. https://doi.org/10.1007/978-3-662-58820-8_21

[40] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017. USENIX Association, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[41] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation. In 22nd NDSS 2015. http://www.internetsociety.org/doc/efficient-ram-and-control-flow-verifiable-outsourced-computation

[42] Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. 2009. Authenticated join processing in outsourced databases. In ACM SIGMOD 2009. 5–18. https://doi.org/10.1145/1559845.1559849

[43] Yin Yang, Stavros Papadopoulos, Dimitris Papadias, and George Kollios. 2009. Authenticated indexing for outsourced spatial databases. VLDB J. 18, 3 (2009), 631–648. https://doi.org/10.1007/s00778-008-0113-2