Authenticated LSM Trees with Minimal Trust

Yuzhe (Richard) Tang Ju Chen Kai Li

Syracuse University, New York, USA

Abstract. In the age of user-generated contents, the workloads imposed on information-security infrastructures become increasingly write intensive. However, existing security protocols, specifically authenticated data structures (ADSs), are historically designed based on update-in-place data structures and incur overhead when serving write-intensive workloads.

In this work, we present LPAD (Log-structured Persistent Authenticated Directory), a new ADS protocol designed uniquely based on the log-structure merge trees (LSM trees) which recently gain popularity in the design of modern storage systems. On the write path, LPAD supports streaming, non-interactive updates with constant proof from trusted data owners. On the read path, LPAD supports point queries over the dynamic dataset with a polynomial proof. The key to enable this efficiency is a verifiable reorganization operation, called verifiable merge, in LPAD. Verifiable merge is secured by the execution in an enclave of trusted execution environments (TEE). To minimize the trusted computing base (TCB), LPAD places the code related to verifiable merge in enclave, and nothing else. Our implementation of LPAD on Google LevelDB codebase and on Intel SGX shows that the TCB is reduced by 20 times: The enclave size of LPAD is one thousand code lines out of more than twenty thousands code lines of a vanilla LevelDB. Under the YCSB workloads, LPAD improves the performance by an order of magnitude comparing with that of existing update-in-place ADSs.

1 Introduction

In the age of cloud computing, outsourcing data storage to the cloud is a common practice (e.g., Dropbox [6], Google drive [9], etc). When using the cloud storage to host security-critical infrastructures (e.g., Bitcoin like cryptocurrencies [3,2,18,7], Google's certificate transparency schemes [5,4,38], etc.), the lack of trust to the public clouds is real and becomes increasingly pressing, in the presence of the constant cloud-security incidents. It calls for security hardening of untrusted cloud storage. In particular, the authenticity of data storage is a fundamental security property critical to many information-security applications. To protect the data authenticity, a common approach is to instantiate an authenticated data structure (ADS) protocol with an untrusted cloud provider and trusted clients.

Many emerging security applications feature a write-intensive workload. For instance, in cryptocurrency, transactions are constantly generated. A public key directory (for certificate transparency) features an intensive stream of certificate-registration and revocation requests. To serve the write-intensive workload with data authenticity, existing ADS protocols present an ill-suited solution. Because most existing ADS protocols [49,36,42,43,54,30,40,34,41,52,53] are designed based on update-in-place data

structures, which incur multiple rounds of communications for serving an update (i.e., interactive update), incurring high overhead in write-intensive workloads.

In this work, we propose LPAD,[1] an authenticated data structure designed uniquely based on the log-structured merge tree (LSM tree) [39]. An LSM tree is an external-memory data structure that optimizes the write performance and is widely adopted in many modern storage systems such as Google LevelDB [11]/Big-table [27], Apache HBase [12], Apache Cassandra [10], etc. With the LSM tree's append-only write design, the LPAD supports streaming, non-interactive data updates from cloud clients. To support verifiable merge operation in an LSM tree, we assume a trusted third-party (TTP) in formulating the LPAD protocol. We believe this assumption is necessary, otherwise the protocol construction will require expensive protocols such as verifiable computations (VC) [45,23,22] – the state-of-the-art VC systems [44,46,25,50] cause multiple orders of magnitude performance slowdown comparing to an unsecured system (without VC).

This work aims at building a real LPAD system with minimal third-party trust. We propose to build the LPAD system by leveraging Intel SGX [15] which supports a trusted "enclave" on an otherwise untrusted platform. The proposed system design runs minimal functions inside SGX enclave, that is, the merge operation and timestamp management. Other than this, majority of the codebase of an LSM data store runs outside the enclave. By this means, it is promising to minimize the trusted computing base (TCB) in the enclave, which renders the system amenable to formal program verification. To authenticate the data storage outside the enclave, we design a digest structure that is aligned well with the LSM tree. The LPAD digest structure is implemented by co-locating the digests (Merkle trees [33]) with the data index, which is promising to save disk seeks when retrieving the query proof.

We evaluate our LPAD protocol and systems in terms of 1) security, 2) minimal TCB and 3) performance overhead. We analyze the protocol security by reducing the query authenticity to the hardness of finding collision in cryptographic hashes. We build a functional LPAD system based on Google LevelDB [11] and SGX SDK. In our LPAD implementation, the TCB size, namely the lines of code running in enclave, is reduced to 4.4% of the entire codebase. We evaluate the performance of our LPAD prototype extensively: Under the common YCSB workloads that are write intensive, LPAD improves the performance by an order of magnitude comparing existing update-in-place ADS.

In summary, the contributions of this work are:

New ADS protocol: This work addresses the authenticated storage of data updates in emerging security scenarios. We identify the performance problem of all existing ADS protocols: The update-in-place data structures existing ADSs rely on cause significant performance slowdown on the write path. To the best of our knowledge, we are the first to propose a log-structured ADS protocol, named LPAD, that allows for non-interactive updates from clients.

New system design: We materialize LPAD with a functional system built on Google LevelDB and Intel SGX. The system design of LPAD reduces the TCB size in enclave. This is done by placing only the computation-oriented code routine inside the enclave.

[1] LPAD stands for Log-structured Persistent Authenticated Dictionary which follows the naming of a common ADS protocol, PAD [32,20].

The system design of LPAD also collocates the digests with data, saving data-retrieve overhead.

2 Preliminaries

This section presents the preliminaries of related techniques to this work.

2.1 LSM Trees and Write-intensive Workloads

A log-structured merge tree (LSM) is designed to be a middle ground between classic B+tree like data structures that are read-optimized and the temporal log structures that are write-optimized. An LSM tree only causes sequential IO for writes, thus preserving write locality similarly to the pure log-structured storage. It avoids the full-disk scan per read by decomposing the storage into several sorted runs, each of which can be indexed and randomly accessed in sublinear time.

Target Workloads: The targeted workload of LSM tree is write-intensive workloads which become popular in serving user-generated contents in modern applications. The target workloads feature 1) an intensive stream of updates on individual data records and 2) data reads that result in random data accesses. In addition, 3) our workloads are issued from security-sensitive scenarios where the data integrity, membership and freshness needs to be guaranteed. Such application scenarios include Bitcoin-alike cryptocurrencies, data-transparency schemes, etc.

2.2 Authenticated Data Structures

An authenticated data structure (ADS) is a protocol that allows a data owner to outsource the data storage to a third-party host which will be queried later by data users. In a public-key setting, the data owner holding the secret key can initially sign and later update the dataset, and the user trusting the owner's public key can verify the query result (assuming an external PKI). An ADS protocol can be thought of as an extended digital signature scheme where the message is a dataset and new algorithms are added to support data-read/write queries. While there is recent ADS research [54,49,43] to support expressive queries and various data structures, in this work we consider the most foundational form of ADS, that is, an authenticated dictionary supporting set-membership queries [20].

Existing ADS constructions [34,48] are mainly based on update-in-place data structures. In the case of a Merkle tree, for instance, an update-in-place ADS requires the data owner (keeping a simple digest/signature) to issue read query first, modify the Merkle authentication proof, and then generate a new signature before writing it to the host. Variants of update-in-place ADSes are proposed, such as replicated ADSes [34,54] and cached ADSes [31]; they improve the update efficiency at the expense of a larger owner state. Update-in-place ADS constructions have been used to implement system prototypes, such as consistency-verified storage [35] and authenticated databases [34].

2.3 Intel Software Guard eXtension (SGX)

Intel SGX is a security-oriented x86-64 ISA extension on the Intel Skylake CPU, released in 2016. SGX provides a "security-isolated world" for trustworthy program execution on an otherwise untrusted hardware platform. At the hardware level, the SGX secure world includes a tamper-proof SGX CPU which automatically encrypts memory pages (in the so-called enclave region) upon cache-line write-back. Instructions executed outside the SGX secure world that attempt to read/write enclave pages only get to see the ciphertext and cannot succeed. SGX's trusted software includes only unprivileged program and excludes any OS kernel code, by explicitly prohibiting system services (e.g., system calls) inside an enclave.

To use the technology, a client initializes an enclave by uploading the in-enclave program and uses SGX's seal and attestation mechanism [21] to verify the correct setup of the execution environment (e.g., by a digest of enclave memory content). During the program execution, the enclave is entered and exited proactively (by SGX instructions, e.g., `EENTER` and `EEXIT`) or passively (by interrupts or traps). These world-switch events trigger the context saving/loading in both hardware and software levels. Comparing with prior TEE solutions [17,19,1,14], SGX uniquely support multi-core concurrent execution, dynamic paging, and interrupted execution.

3 System Overview and Motivation

In this section, we present the system architecture in terms of target applications and trust model. We also present our motivating observation, that is, placing existing update-in-place ADS construction over log-structured storage results in inefficiency.

3.1 System Model and Security Goals

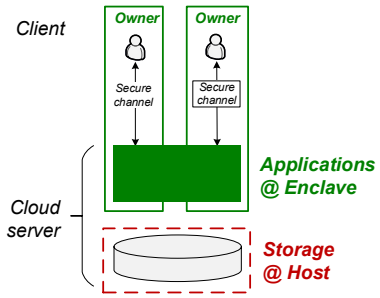


Fig. 1: System trust model: The box with solid lines (in green) means the trusted domain including the enclave and owners. The box with dotted lines (in red) means the untrusted host domain.

We consider the common cloud-storage scenario that cloud customers outsource their data storage to a third-party cloud platform. The cloud instance runs over an SGX machine and exposes an enclave to the customer. The server runs application and storage services. It persists data to its storage media through a key-value store. In this work,

we consider the LSM tree based key-value stores exposing a standard PUT/GET interface (will be elaborated soon). The use of LSM-based key-value store allows the system to ingest an intense stream of data writes, featured in our target applications.

The enclave is trusted by the cloud customer and we assume the standard techniques to establish such trust (e.g., software attestation mechanism [21] and key-exchange protocols [33]). The communication channel between the customer and the server enclave is thus secured by standard TLS protocols. In our system model, we assume the enclave and cloud customer are securely connected. Figure 1 illustrates the system architecture.

In the system, the enclave runs the server-side applications, and the host outside the enclave runs an LSM-tree based key-value store. The two interact through a standard key-value store API: Given key k, value v, timestamp ts, a write operation $\text{PUT}(k, v)$ returns an acknowledgment about committed timestamp ts, and a read operation $\text{GET}(k, ts_q)$ returns result record $\langle k, v, ts\rangle$ where ts_q denotes the timestamp at the invocation time of GET and ts denotes the timestamp of record returned. Formally,

$$\begin{aligned} ts &:= \text{PUT}(k, v) \\ \langle k, v, ts\rangle &:= \text{GET}(k, ts_q) \end{aligned} \tag{1}$$

Timestamp management: In our scheme, timestamp is managed inside enclave and is backed by a trusted storage service (e.g., TPM chip) to defend rollback attacks by the untrusted host. Upon PUT/GET requests, the timestamps are managed in the following manner: 1) For each PUT operation issued by the application, the enclave serializes the operation and monotonically increases the current timestamp to assign a unique timestamp for the operation. 2) For each GET operation issued by the application, it simply retrieves the current timestamp and uses it as t_q in the GET request. 3) The timestamp is periodically "flushed" to the trusted storage. The "flush" of timestamp counter can be set at a fixed rate or be coupled with the "flush" operation in the underlying LPAD. what does this mean?

Given a read operation, there are several properties associated: 1) Result integrity is about whether the result of a read, say $\langle k, v, ts\rangle$, is a valid data record (meaning the one written by a legitimate write before). The result integrity can be protected by a simple use of message authentication code (MAC), and it is not our main security goal. 2) Result membership is about whether a read result is fresh and complete in the dataset stored in the key-value store. The freshness states whether the result $\langle k, v, ts\rangle$ has the largest timestamp (or is the latest) among all records of the queried key k and with a timestamp smaller than ts_q. The completeness prevents a legitimate result from being omitted. The result membership in freshness and completeness can be authenticated using the LPAD scheme.

Threat model: The trust boundary in our system occurs between the enclave and the server host. The server host includes 1) hardware devices except for the SGX processor, and 2) software that is loaded and is executed in the untrusted memory, including the privileged operating system. The adversary can control the server host software stack and subvert the storage systems there by forging operation results (will be described next). The adversary in this work is assumed not to compromise the SGX hardware which is tamper resistant. In addition, this work does not address the following attacks: denial-of-service attacks, proven deletion under Sybil attacks, SGX side-channel attacks [51], design flaws or enclave program security.

A query-forging adversary can modify the result of a data read (or a write) returned from the untrusted server host. Given a read query, the adversary can present an incorrect result. Specifically, she can present a non-exist data record (breaking the result integrity), present a properly signed but stale result to the enclave (breaking the freshness), or present an empty result while omitting a matching record in the dataset (breaking the completeness).

Security goals: The security goal in this work is that the enclave issuing PUT/GET requests can verify the freshness of the query results. If the untrusted host forges any query results, the verification algorithm in the enclave cannot pass. Note that in this work, we don't address the mitigation of query-forging attack, that is, when the forging occurs, the enclave cannot recover the honest result from the forged result.

4 LPAD Protocol: Scheme and Constructions

In this section, we define the standard LPAD scheme. An LPAD is an ADS scheme tailored to the LSM tree structure. We first present a model of an LSM tree before describing the LPAD scheme and security.

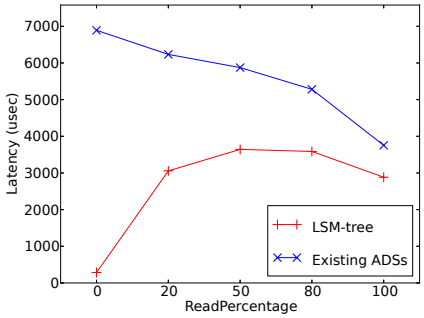
4.1 Design Motivation

To motivate our LPAD protocol, we present a strawman design that layers the update-in-place ADS over an LSM-tree based store.

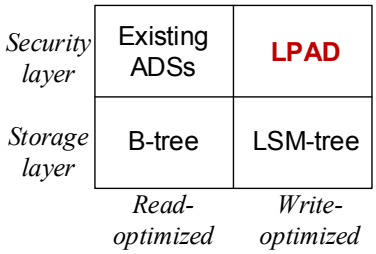
When placing an update-in-place ADS over an LSM tree, an immediate problem is that the authentication data has a different structure from the actual data. The structure mismatch creates extra engineering difficulty and performance overhead. As the motivation of our work, we conduct a performance study and show the slowdown. We consider an update-in-place ADS built on top of an imaginary binary search tree, which is mapped to the input domain of the underlying LSM store (e.g., through the encoding of in-order tree traversal). Each tree node is mapped to a key-value record in the LSM store. By this means, each read (write) is translated to a series of index-node lookups and the final data transfer. The performance result of this strawman comparing the ideal case (an LSM tree without any ADS) is illustrated in Figure 2a. It shows that with the presence of an update-in-place ADS, it adds a significant amount of overhead to the write performance. In addition, in the target workload featuring an intensive stream of writes, the slowdown is up to several orders of magnitudes.

4.2 Model of LSM Tree

An LSM tree represents a dataset m by a series of so-called levels, $l_0, l_1 ... l_{q-1}$. A level l_i is a list of ordered data records $l_i = b_1 b_2 ... b_j ....$ An LSM tree supports the basic data reads and writes, where a write only updates the first level l_0. A read may iterate through all levels to find a match. An LSM tree supports the MERGE operation that merges two adjacent levels (e.g., l_i and l_{i+1}) into one level. In the LSM tree, the first level l_0 resides in memory and all immutable levels $l_{\geq 1}$ reside on disk. Note this simple structure ensures writes are clustered into sequential storage access.



(a) Performance mismatch between data-level LSM tree and security-level ADSs



(b) Comparing the LPAD design with existing ADS: This figure considers random-read access and excludes the pure log-based design (e.g., log-structured file systems and hash-chain).

Fig. 2: Performance of update-in-place ADS over LSM and the design of LPAD

Table 1: Notations

b	key-value record	m	dataset
n	security parameter	ts	timestamp
a	answer	π	proof
l	LSM-tree level	q	number of levels

4.3 LPAD Scheme & Security

LPAD extends an ADS scheme with new algorithms to handle the MERGE operation interactively. Formally, consider a set-membership predicate: Given dataset m and record b, a set-membership predicate is $0, 1 := P(m, b)$ where 0/1 represent non-membership/membership of the record in dataset m. The scheme of an LPAD is the following:

LPAD scheme Π^{LPAD} consists of eight probabilistic polynomial-time (PPT) algorithms (GEN, SETUP, QUERY, VRFY, UPDATE, REFRESH, MERGE, SIGMERGE), where the first six are defined in a standard ADS scheme, and the last two are new algorithms in LPAD. Specifically, SETUP signs the initial dataset (m), (QUERY,VRFY) forms an interactive[a] sub-protocol for point-read (record b), and (UPDATE,REFRESH) forms an interactive point-write sub-protocol. The pair of new algorithms, (MERGE,SIGMERGE), interactively merge the levels.
For simplicity, consider the three-party ADS model[b] where a data owner writes to a server host and clients read ADS from the server. The owner holding secret key sk keeps a full copy of dataset m.

- $pk, sk \leftarrow \text{GEN}(1^n)$: A pair of public/private keys are generated with security parameter n.
- $s \leftarrow \text{SETUP}_{sk}(m)$: Owner signs the initial dataset m using secret key sk.

- $\pi, a \leftarrow$QUERY$_{pk,P(\cdot,\cdot)}(m, b)$: The host processes a set-membership query on record b against dataset m using public key pk. It returns the answer a of set-membership relation $P(\cdot,\cdot)$ and a proof π.
- $1, 0 :=$VRFY$_{pk}(\pi, a)$: The client receiving proof π and answer a verifies using public key pk whether the answer is authentic. 1 means the authentic answer.
- $l'_0, s'_0, upd \leftarrow$UPDATE$_{sk}(b, l_0)$: The owner adds a new record b to level-zero in the dataset l_0. It also generates update information upd.
- $l'_0, s'_0 :=$REFRESH$_{pk}(b, l_0, upd)$: The host receiving a new record to add b and update information upd (resulted from Algorithm UPDATE or SIGMERGE) refreshes the signature of level zero to be s'_0 using upd.
- $\varnothing, s'_i, l'_{i+1}, s'_{i+1}, upd :=$SIGMERGE$_{sk}(l_i, s_i, l_{i+1}, s_{i+1})$: The owner merges two adjacent levels (l_i, l_{i+1}) in the prior state to posterior state $(\varnothing, l'_{i+1})$. It generates the signatures of the two levels in the posterior state (s'_i, s'_{i+1}) with update information upd.
- $\varnothing, s'_i, l'_{i+1}, s'_{i+1} :=$MERGE$_{pk}(l_i, s_i, l_{i+1}, s_{i+1}, upd)$: The host merges two adjacent levels (l_i, l_{i+1}) to posterior state $(\varnothing, l'_{i+1})$ using public key pk and update information upd.

[a] Here, interactive means that the two algorithms (QUERY and VRFY) can be called multiple times.

[b] The extension from the three-party model to the two-party model is straightforward and can be found in related work [43].

The correctness of LPAD scheme is straightforward and similar to that of ADS [43]; informally, the correctness can be stated by that given any state resulted from calling UPDATE/REFRESH and MERGE/SIGMERGE, and given any correct QUERY on the state, running the verification algorithm (VRFY) will return 1. The security of LPAD scheme is defined in a game where an adversary can access public key pk (i.e. freely access VRFY,MERGE,REFRESH).

4.4 LPAD Construction by a Forest of Merkle Trees

This subsection presents a basic construction of LPAD and next subsection presents a read-optimized construction.

The basic LPAD construction authenticates each level by a standard ADS such as a Merkle tree. While this paper considers Merkle tree for construction, we stress that the LPAD is a paradigm that can work with other per-level ADS primitives (e.g., multi-set hash [28]). Concretely, GEN(1^n) runs the standard public-key generation algorithm, and SETUP$_{sk}(m)$ signs the initial dataset m using secret key sk before the owner uploads the digest to the server. On the read path, the untrusted server runs $\pi, a \leftarrow$QUERY$_{pk,P(\cdot,\cdot)}(m, b)$ that prepares a query proof by including the membership proof for the level that contains the answer and more importantly the non-membership proofs for the levels that don't contain the answer. Then the client receiving proof π and answer a verifies the answer integrity by running $1, 0 :=$VRFY$_{pk}(\pi, a)$, which further runs the verification algorithms of the per-level ADS against corresponding per-level proofs. It only accepts when *all* per-level verification accept.

membership proof

On the write path, the trusted owner updates the remote dataset by running the UPDATE algorithm of the first-level ADS, namely $l'_0, s'_0, upd \leftarrow \text{UPDATE}_{sk}(b, l_0)$. Then, the untrusted server refreshes the dataset based on the owner's update by running the first-level ADS's REFRESH algorithm, namely $m'_0, s'_0 := \text{REFRESH}_{pk}(b, m_0, upd)$.

Asynchronously, the server and owner interactively run algorithms to merge two adjacent levels: The owner locally updates the two adjacent levels (l_i, l_{i+1}) to posterior state $(\varnothing, l'_{i+1})$, with signatures and update information upd. The server then merges two adjacent levels (m_i, m_{i+1}) to $(\varnothing, m'_{i+1})$ and simply updates their signatures using update information upd. The $\text{SIGMERGE}(l_i, l_{i+1})$ is constructed by the owner retrieving from the host the two input levels, l_i and l_{i+1}, and linearly scanning them. This straightforward construction with linear cost may not be feasible in the traditional setting, but is practical in the case with TEE where the server is co-located with the enclave owner.

The correctness of LPAD construction is straightforward and we omit it in this and subsequent constructions.

Security analysis: The basic LPAD construction is secure as long as the per-level ADS constructions are secure. Because our security proof is based on the reduction to the security of per-level ADS. That is, if LPAD is insecure, it means at least one of per-level ADSes is insecure. Briefly, our formal security proof under the LPAD-forging game relies on the idea that MERGE can be "simulated" by a series of UPDATEs.

5 LPAD Systems

In this section, we present the engineering of LPAD protocol when building a functional storage system on Intel SGX. We build the system based on Google LevelDB [11], which is a representative and widely adopted storage system based on the LSM trees. In this paper, we use the term "LevelDB" to represent a broad class of log-structured key-value stores, such as Apache HBase [12], Apache Cassandra [10], Facebook RocksDB [8].

5.1 System Design and Implementation

System Overview Our LPAD schemes are built on public key cryptography. When instantiating the scheme on SGX, we naturally place the secret key inside the enclave. Note that we assumed a secure key-management component in enclave such as sgx-kms [26]. In addition, all LPAD algorithms that access the secret key are run inside enclave. Algorithms with the public key are run by the untrusted host, except for the VRFY algorithm whose return value is security critical.

Therefore, our system runs the following LPAD algorithms in enclave: 1) The algorithms involving secret keys, that is, GEN, SETUP, UPDATE, SIGMERGE (recall the LPAD scheme in § 4) are executed in an enclave. 2) The verification algorithm (i.e. VRFY) is placed in an enclave as the verification result is critical to the protocol security.

In particular, to support SIGMERGE, it runs a clone of MERGE computation inside enclave. The data buffered in memory is placed outside the enclave but we allow the in-enclave MERGE computation to directly access the buffer outside enclave. The data

buffer is placed outside enclave, because it is accessed by the enclave for once (i.e. no locality) and placing it in enclave does not save boundary-crossing overhead. During the MERGE, the inputs are read from the disk to buffer and enclave, and are authenticated in a deferred fashion by reconstructing the entire Merkle tree of the levels. By the end of MERGE, the newly generated level is signed by the enclave. Details about implementing verifiable MERGE is presented in § 5.1.

Digest Structures In LPAD, the data storage is hosted in the untrusted world and it entails to authenticate the data outside the enclave by building a digest structure. We design and implement a digest structure that is aligned well with the data layout in an LSM store, aiming to minimize the imposed IO overhead.

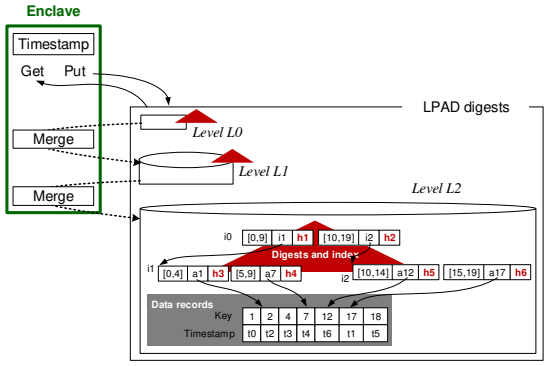


Fig. 3: LPAD system architecture with digest structures

The LPAD digest structure consists of Merkle trees built over the LSM tree dataset. Each Merkle tree digests an LSM-tree level. The data records are digested in their original order, that is, sorted first by data keys and then by timestamps. In an LSM store, data records are stored in a data file where records are indexed to facilitate the data lookup. The LPAD Merkle tree is stored by being embedded in the index. That is, each pointer in a tree index is augmented to store the hash of the Merkle tree.

An example: In Figure 3, there is an LSM tree with three levels. For the third level, it contains a list of key-value records, sorted first their data keys and then by timestamps. There exist an index structure that facilitates the lookup of a data key in the level. The index is a three-node B tree (of nodes $i0, i1, i2$). Each index node contains multiple entries, each with a data-key range and a pointer to the child index node. For instance, the index node $i1$ has an entry $([0, 4], a1)$, which is used to direct a search with data key in $[0, 4]$ to the data block starting with record $\langle 1, t0\rangle$. LPAD system augments each index entry with a Merkle-tree hash. For instance, the index entry $([0, 4], a1)$ is augmented with hash $h3$ where $h3 = H(\langle 1, t0\rangle \| \langle 2, t2\rangle \| \langle 4, t3\rangle)$. In the LPAD Merkle tree over level l_2, the Merkle tree is constructed by $h1 = H(h3 \| h4)$ and $h2 = H(h5 \| h6)$.

In LPAD system, the storage of LPAD Merkle trees is co-located with the index in an LSM store. One of the benefits for this design is that the co-located data and digest storage can reduce the data-access cost. That is, when storing the Merkle trees in a separate file, retrieving the Merkle proof involves multiple random accesses on disk,

incurring expensive disk seeks. In our co-located digest storage, the random-access to retrieve proof is piggybacked in the data access path, namely, the seek to look up the index is also used to prepare the hashes in a LPAD proof, saving extra seeks.

MERGE Implementation Our system runs three functionalities in enclave for verified MERGE: 1) the computation of MERGE, 2) the authentication of input data that comes into the enclave in a streaming fashion, 3) the signing of the output data. The code of these three functions run inside the enclave and the data buffer resides outside the enclave. More specifically, given multiple files at consecutive levels as input, the verified MERGE inside enclave loads the data files into data buffers outside enclave, merge-sort the data records, reconstructs the Merkle root hashes for all input files, and builds a Merkle tree over the merged output stream of records. If the reconstructed Merkle root hashes are identical to what are stored in enclave, the enclave make effective of this MERGE operation by updating the digest of the relevant levels with the reconstructed Merkle hash.

5.2 Security Analysis

In our system in SGX, an invariant is that on both write and read paths, *the in-enclave algorithm of LPAD (i.e.* UPDATE *and* VRFY*) occurs after the outside-enclave algorithm (i.e.* REFRESH *and* QUERY*)*. This invariant, with the promise of fully serialized execution,[2] allows the enclave to construct the execution order (of reads and writes) from the order these in-enclave algorithms are called. This execution order further allows to fully specify the execution history, based on which the membership can be authenticated by LPAD (e.g., freshness assumes the temporal order among reads/writes).

Concretely, we consider the freshness attack that the adversary from the untrusted host presents a correct but stale read result. The freshness property requires that a read result $\langle k, v, ts\rangle = \text{GET}(k, ts_q)$ is fresh as of timestamp ts_q. By definition, it can be authenticated by the membership of result record $\langle k, v, ts\rangle$ and the non-membership of a virtual record $\langle k, v', ts'\rangle$ that is "fresher" and with $ts' \in [ts, ts_q]$. Both the membership and non-membership can be further authenticated by the LPAD scheme underneath. Based on freshness authentication, any stale result returned from the untrusted host can be easily detected (by the failure of VRFY). Here, special notes should be taken that under the above invariant (i.e. in-enclave algorithms occur after the untrusted LPAD algorithms), the untrusted host is given the chance to store the data no older than the digest in enclave, and thus any deviation from the (non)-membership proven by the digest can be attributed to the misbehavior of the untrusted host.

The freshness attack can be extended to different forms: 1) The completeness attack is a special form of freshness attack where the untrusted host omits the result and falsely returns an empty result. In this case, the non-membership authentication (for the empty result) will not pass. 2) The forking attack [37] works by the untrusted host presenting different views to different reads. As our enclave under LPAD protocol fully specifies the operation history (without ambiguity), there is always only one legitimate result

[2] The untrusted host can break the promise of serialized execution, but will eventually be detected through the in-enclave checks.

that can be authenticated, thus eliminating the forking vulnerability. Note that we do not consider concurrency attacks.

6 Evaluation

In this section, we evaluate LPAD system with the goal of answering the following questions:

- What is the trusted code size (§ 6.1)?
- What is the performance of LPAD under IO-intensive workloads (§ 6.2)?

6.1 Implementation & Enclave Code Size

Original LevelDB codebase: The codebase of LevelDB consists of several code modules. Their profiles are the following: F1) A skip list that is accessed by reads and writes on level l_0 consists of $1.3K$ lines of code. F2) A write-ahead log (WAL) that is accessed by a write at level l_0 consists of $1K$ LoC to persist writes and recover state. F3) LRU data cache and Bloom filter that are accessed by reads on levels $l_{\geq 1}$ consists of $1K$ LoC. F4) Merge computation for compaction consists of $0.2K$ LoC. F5) Thread management code for compaction ($< 0.1K$ LoC), F6) Application-specific IO handling, e.g., for file parsing, file meta-data management, etc., consists of $4.7K$ LoC, and F7) miscellaneous utility code, e.g., for computing regular hash, consists of $4K$ LoC.

Code modification of LevelDB: The system implementation is based on the codebase of LevelDB with the following changes: 1. We add a program to hook our in-enclave program to the LevelDB running in the untrusted host. 2. We add a program to store and serve the Merkle trees for QUERY on the untrusted host; several LevelDB utilities are reused for the Merkle-tree persistence. 3. We modify the LevelDB codebase to make each PUT return its timestamp. The change is not significant and does not cause overhead.

For evaluating the trusted code size, we prepare a baseline realizing Haven-style partitioning. In particular, our baseline is based on the latest systems-level support in enclave, Panoply [47]. Comparing with Haven [24], Panoply considers the application logic is partitioned to several modules, with each loaded in a container with Panoply's rich systems interface (e.g., thread management). Despite its rich in-enclave functionalities and minimized systems-level TCB, we stress Panoply is not application partitioning scheme and cannot specify how to partition LevelDB. Thus, in our baseline, we map the entire codebase of LevelDB into enclave.

Recall that the LPAD places inside the enclave trusted LPAD algorithms (SIGMERGE,UPDATE,VRFY) and requires the enclave to include the code for Merkle proof and SHA computation. Additionally, the enclave runs some glue code generated by Intel SGX SDK [16]. The total number of code lines in enclave is around 900. Comparing with the baseline approach, trust-minimized partitioning reduces the application-level trusted code size by 20 times.

The result of enclave code size is presented in Table 2.

Table 2: Trusted code size with LPAD partitioning strategies

Partitioning scheme	Trusted code size (LoC)
LPAD	891
The Haven [24] approach (LevelDB in enclave)	$\sim$ 20000

6.2 Performance Evaluation

In this section, we present the performance of LPAD under Yahoo Cloud Serving Benchmark (YCSB) [29] which is a standard benchmark suite. We evaluate the performance under IO-intensive workloads. We start by describing the common experiment setup.

Experiment setup: We did all the experiments on two laptops with an Intel 8-core i7-6820HK CPU of 2.70GHz and 8MB cache, 32 GB Ram and 1TB Disk. This is one of the Skylake CPUs with SGX features.

We used the YCSB benchmark suite [29] that provides a workload generator and a multi-threaded execution platform for evaluating the performance of generic key-value stores. We leverage the LevelDB-YCSB adapter based on online projects.[3] In our experiment, we run the YCSB workload driver on one machine and the storage system on another machine; the two machines are connected in a high-speed LAN network.

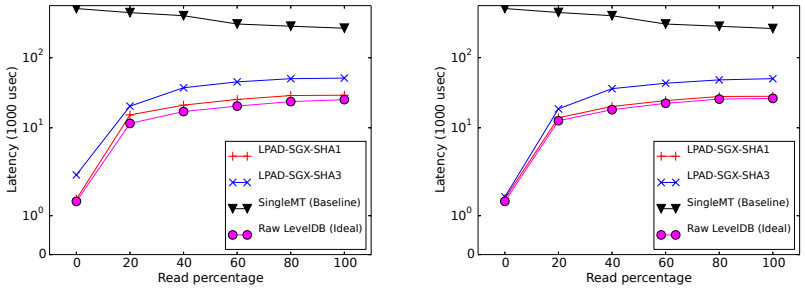
We use two datasets in this experiment: The large dataset contains 200 million records (which is 24GB without compression under 100-byte values), and the small dataset contains 1 million records (140MB without compression). The large dataset is intended to capture the IO-intensive workload where the working set is larger than memory and IO is constantly triggered during data serving. The small dataset captures the memory intensive workloads with the working set fully residing on memory; in this case memory references (or cache misses) are the bottleneck. Both datasets are generated with uniformly distributed keys, each key-value record contains a 16-byte key and a value that can take a size of 100 or 1000 bytes.[4] We used the SHA3 hash algorithm from the Crypto++ library [13].

IO-intensive workload In the experiment, we varied the read percentage from 0% (that is, a write-only workload), $20\%, 40\%, 60\%, 80\%$ to 100% and we tested 1 million queries. We consider both SHA1 and SHA3 algorithms. We vary record size (116 bytes and 1016 bytes). Our experiments are conducted with MERGE turned on and in a single thread. Each experiment is run at least three times. We report the latency per operation.

We compare the performance of our LPAD-SGX system against two baselines: 1) The first is a raw LevelDB instance running in the untrusted world. This is an unsecured solution, but its performance is ideal. We name this baseline by "Raw LevelDB (Ideal)". 2) The second baseline is a LevelDB protected by a single Merkle tree, which represents most existing work in the ADS literature. This baseline is named by "SingleMT (Baseline)".

[3] `https://github.com/jtsui/ycsb-leveldb`

[4] Note the smaller size a value is (e.g., 100 byte), the more challenging to serve for a storage system as small writes cause more random access IO.



(a) IO-intensive workloads (116-byte records, SingleMT stands for the base-line approach of a single Merkle tree for authentication)

(b) IO-intensive workloads (1016-byte records)

Fig. 4: LPAD-SGX performance

The performance result under the IO-intensive workload is presented in Figure 4a and Figure 4b. For both 1016-byte and 116-byte record sizes, the LPAD-SGX scheme matches well with the write-optimized characteristics of the original LevelDB – their latency increases as the workload becomes more read intensive. By contrast, the baseline of a single Merkle tree exhibits a read-optimized behavior. More over, with any read-write ratio, the LPAD-SGX systems' slowdown comparing the ideal performance is at most $2X$, which is much smaller than the 500X slowdown of the SingleMT baseline (the single Merkle tree). By using SHA1 instead of SHA3, the LPAD-SGX system further reduces the slowdown to 36% for the 116-byte records and 12% for the 1016-byte records. This result confirms the benefit of matching security protocol with the underlying storage system.

7 Conclusion

This work presents LPAD, an ADS protocol designed based on LSM trees to address the efficiency under write-intensive workloads. A functional system is built based on LPAD that is on top of Google LevelDB with Intel SGX. The system design of LPAD features three salient properties: 1) It supports a small enclave program by having around one thousand in-enclave code lines out of more than twenty thousands code lines of LevelDB. 2) It guarantees query authenticity in terms of data integrity and membership. 3) The performance slowdown of LPAD is less than 12%.

Acknowledgement

Yuzhe Tang's work is supported by National Science Foundation under Grant CNS1815814 and a gift from Intel.

References

1. ARM TrustZone, https://www.arm.com/products/security-on-arm/trustzone.
2. Bitcoin core: https://bitcoin.org/en/bitcoin-core/.
3. Bitcoin: https://bitcoin.org/en/.
4. Certificate transparency.
5. Certificate transparency, the internet standards.
6. Dropbox: `www.dropbox.com`.
7. Ethereum project: https://www.ethereum.org/.
8. Facebook RocksDB, http://rocksdb.org/.
9. Google drive: `https://www.google.com/drive/`.
10. http://cassandra.apache.org/.
11. http://code.google.com/p/leveldb/.
12. http://hbase.apache.org/.
13. http://www.cryptopp.com/benchmarks.html.
14. IBM SCPU, http://www-03.ibm.com/security/cryptocards/.
15. Intel corp. software guard extensions programming reference, 2014 no. 329298-002.
16. Intel software guard extensions (intel sgx) sdk.
17. Intel TXT, http://www.intel.com/technology/security/ downloads/trustedexec_overview.pdf.
18. Litecoin: https://litecoin.org/.
19. TPM, http://www.trustedcomputinggroup.org/tpm-main-specification/.
20. A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *Information Security ISC 2001*, pages 379–393, 2001.
21. I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for cpu based attestation and sealing.
22. S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998.
23. S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1):70–122, 1998.
24. A. Baumann, M. Peinado, and G. C. Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 267–283, 2014.
25. B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 341–357. ACM, 2013.
26. S. Chakrabarti, B. Baker, and M. Vij. Intel SGX enabled key manager service with openstack barbican. *CoRR*, abs/1712.07694, 2017.
27. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI*, pages 205–218, 2006.
28. D. E. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental multiset hash functions and their application to memory integrity checking. In C. Laih, editor, *Advances in Cryptology - ASIACRYPT 2003, 9th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, November 30 - December 4, 2003, Proceedings*, volume 2894 of *Lecture Notes in Computer Science*, pages 188–207. Springer, 2003.
29. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.
30. P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic data publication over the internet. *Journal of Computer Security*, 11:2003, 2003.

31. R. Elbaz, D. Champagne, C. H. Gebotys, R. B. Lee, N. R. Potlapally, and L. Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. *Trans. Computational Science*, 4:1–22, 2009.
32. M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, volume 2, pages 68–82. IEEE, 2001.
33. J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.
34. F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD Conference*, pages 121–132, 2006.
35. J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI*, pages 121–136, 2004.
36. C. U. Martel, G. Nuckolls, P. T. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.
37. D. Mazières and D. Shasha. Building secure file systems out of byantine storage. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 108–117, 2002.
38. M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: bringing key transparency to end users. In J. Jung and T. Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 383–398. USENIX Association, 2015.
39. P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.
40. H. Pang and K.-L. Tan. Authenticating query results in edge computing. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 560–, Washington, DC, USA, 2004. IEEE Computer Society.
41. S. Papadopoulos, Y. Yang, and D. Papadias. Cads: Continuous authentication on data streams. In *VLDB*, pages 135–146, 2007.
42. C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In P. Ning, P. F. Syverson, and S. Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 437–448. ACM, 2008.
43. C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, 74(2):664–712, 2016.
44. B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 238–252. IEEE Computer Society, 2013.
45. R. Rubinfeld and A. Shapira. Sublinear time algorithms. *SIAM J. Discrete Math.*, 25(4):1562–1588, 2011.
46. S. T. V. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 71–84, 2013.
47. S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26-March 1, 2017*, 2017.
48. E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: a scalable cloud file system with efficient integrity checks. In *ACSAC*, pages 229–238, 2012.
49. R. Tamassia. Authenticated data structures. In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, pages 2–5, 2003.

50. R. S. Wahby, S. T. V. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.
51. Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 640–656. IEEE Computer Society, 2015.
52. Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 5–18. ACM, 2009.
53. Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Authenticated indexing for outsourced spatial databases. *VLDB J.*, 18(3):631–648, 2009.
54. Y. Zhang, J. Katz, and C. Papamanthou. Integridb: Verifiable SQL for outsourced databases. In I. Ray, N. Li, and C. Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1480–1491. ACM, 2015.