LPAD: Building Secure Enclave Storage using Authenticated Log-Structured Merge Trees

Yuzhe (Richard) Tang, Ju Chen
Department of EECS, Syracuse University

Abstract

With the advent of commercial trusted execution environments (e.g., Intel Software Guard eXtension or SGX), an important research task is building trustworthy software systems based on the TEE, which will enable a wide range of security applications on the third-party cloud.

This work aims at building secure and high-performance storage systems for safe data outsourcing. It considers as storage substrate modern key-value stores, such as Google LevelDB, that adopt the design of log-structured merge trees (LSM). We propose Log-structured Persistent Authenticated Dictionary (LPAD), a security protocol that specifies the workflow of an LSM tree for the Intel SGX architecture. We build a secure storage system following the LPAD protocol and based on Google LevelDB. When building the system, we study a range of software-partitioning strategies that make the tradeoff between performance overhead and the size of trusted computing base.

We evaluate the LPAD storage for three salient features: formal security in terms of strong data authenticity, low performance overhead and small trusted computing base (TCB). On the latter two aspects, our evaluation shows that 1) the LPAD-based system has a small trusted program. 2) The performance overhead is low with a typical 12% $\sim$ 40% slowdown.

1 Introduction

Hosting applications on a third-party platform becomes an increasingly popular computing paradigm in the age of cloud. Due to the lack of trust (to the cloud), supporting security-sensitive applications poses challenges. The commercial trusted execution environments (TEE) such as recently released Intel Software Guard eXtension or SGX [12] and ARM TrustZone [4, 1] allows for setting up an enclave for running applications on a remote host and are promising to solve the lack-of-trust problem. With the availability of new hardware, it is an important research problem to build trustworthy and secure software systems leveraging the TEE features [75, 22, 64, 52, 39, 60, 22].

This work focuses on building an important class of software systems on Intel SGX — Secure and high-performance key-value stores. Concretely, the target data systems include Google BigTable [31]/LevelDB [9], Apache HBase [3], Apache Cassandra [2], Facebook RocksDB [6], and many others. These storage systems are designed based on Log-structured Merge Tree (LSM) [53], which is an external memory data structure that supports append-only writes and random-access reads, with balanced performance (§ 2.1). The LSM-based key-value stores have been or can be used in many emerging security-sensitive application scenarios (See § 3.1 for a list of such applications).

When building a TEE-based system, the key design problem is "software partitioning." That is, the software system needs to be partitioned into two pieces, run respectively in the trusted world (i.e., enclave) and the untrusted world of TEE. To partition an LSM-based key-value store, the first baseline (B1) is to do so at the storage API level. That is, the trusted enclave runs an application program sending storage read/write requests and the untrusted host runs the LSM key-value store that serves the requests. To secure the trust-boundary crossing (between the enclave and host), security protocols, such as Authenticated Data Structures or ADS [74, 68, 54, 20], are applicable. Existing ADS protocols entail maintaining an update-in-place structure of digests on the untrusted host, which has very different performance characteristic than LSM trees and causes high overhead (See § 3.4 for a detailed explanation). The second baseline (B2) is to do the partitioning at the systems level (e.g., system calls). That is, the trusted enclave runs an unmodified storage system using Library operating-system supports [26, 70, 65], and the untrusted host runs the native operating systems to serve the system calls. This partitioning scheme may result in a large trusted computing base (TCB) at the application level[1], which renders it hard for verifying the program security.

In this work, we propose a low-overhead and small-TCB partitioning scheme tailored to an LSM-based storage system. The key idea is to partition the key-value stores, not as high as the API level (i.e., in B1) or as low as system calls (i.e., in B2), but rather at a middle level with awareness of the LSM-tree abstraction. This middle-ground solution is advantageous in saving the TCB by placing only necessary computation inside enclave, as well as in low overhead by closely matching the security-layer structure with underlying storage systems. Note that existing work for automatic partitioning such as Glamdring [47] and [59] is ineffective in our context due to the lack of knowledge on LSM-specific semantics.

To instantiate the LSM-storage partitioning, we first propose a formal security protocol, LPAD or Log-structured Persistent Authenticated Directory. The purpose of this formalization is for provable security. A challenge in the protocol design is to avoid under-specification (losing security) or over-specification (losing systems-level flexibility). LPAD formalizes the protocol at an appropriate level that provides formal security and admits a wide variety of data-maintenance policies in an LSM tree. In addition, LPAD can be constructed efficiently without relying on expensive cryptographic primitives (e.g., verifiable computation [24, 23, 55, 62, 28, 71]). It places minimal computation in its trusted party, rendering it amendable for a small TCB.

At the systems level, we then study the placement of the memory objects in the storage system. We explore the design space of placing different objects inside/outside enclaves, and propose a series of strategies that make trade-off between performance and the TCB size.

We implement our design based on the codebase of Google LevelDB [9] and result in a security-enhanced key-value store functioning on Intel SGX. Based on the implementation, we evaluate the LPAD system in three aspects: 1) The LPAD system on LevelDB has a small application TCB of one thousand codelines in enclave, comparing the twenty thousands code lines of original LevelDB. 2) The security is formally analyzed in the language of security games. 3) The performance overhead can be as low as 12% slowdown under the IO-intensive workloads and around 40% slowdown under the memory-intensive workloads (in § 6).

In summary, this work contributes the following techniques:

1. We identify the gap between the design of existing security protocols (i.e., ADS) and that of LSM-based storage systems. The gap results in a significant performance problem, as demonstrated through analysis and performance study. 2. We propose LPAD, the first security protocol that marries the ADS design with LSM trees. LPAD achieves efficiency and small TCB by having consistent data structures across layers. 3. We instantiate LPAD with a series of data-placement strategies, that make the tradeoff between performance and TCB size. 4. We implement a secure-storage prototype based on Google LevelDB that is functioning on Intel SGX. We demonstrate the system is strongly secure, has a small application-level TCB and low performance overhead.

We present the rest of paper in the following order: preliminaries in § 2, research formulation in § 3, the proposed LPAD protocol and constructions in § 4, LPAD instantiation on Intel SGX in § 5, the system evaluation in § 6, the related work in § 7 and the conclusion is in § 8.

2 Preliminaries

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

2.1 LSM Tree-based Storage Systems

A log-structured merge tree (LSM) [53] is a data structure that serves random-access reads and writes with a unique performance characteristic. At the high level, an LSM tree is a forest of sub-trees, where each sub-tree's leaf nodes store a sorted run of records by data keys (each record is a key-value pair). A subtree in an LSM tree is also called a level. A write operation to the tree only update the first level, making this level mutable and all other levels immutable.[2] A read operation may iterate through all levels (in the worst case). An offline "merge" operation[3] migrates records from lower levels to a higher level.

LSM trees have been recently adopted in the design of many modern storage systems, including Google BigTable [31]/LevelDB [9], Apache HBase [3], Apache Cassandra [2], Facebook RocksDB [6], etc. In these systems, an LSM tree is treated as an external memory data structure and used for managing storage IOs. Specifically, the first level of an LSM tree resides on memory and all other levels, which are immutable, reside on disk; by this means, only sequential disk writes occur (random-access writes are contained in memory by the first level).

In terms of performance, the LSM-tree based storage design represents a middle ground between the two classic designs, that is, the read-optimized update-in-place storage (e.g., B+ tree and many database indices [38, 66])

[1]Be aware that the code size of modern storage systems is usually at the scale of tens or hundreds of thousands of codelines (e.g., HBase [3] and LevelDB [9]).

[2]To be more precise, immutable in this work means immutable to random-access writes.

[3]In an LSM tree, a merge operation is also called compaction; we use the two words interchangeably.

and the write-optimized log-structured storage (e.g., log-structured file systems [58]). On the one hand, an LSM tree (in an external memory model) serves a data write in an append-only fashion, in a way similar to log-structured file systems. On the other hand, it supports random-access reads without scanning the entire dataset, which is similar to update-in-place style B+ trees. An LSM tree reaps the benefits from both worlds, at the expense of assuming some offline hours to do the batched merge operation. We will formally describe an LSM tree in § 4.1.

2.2 Authenticated Data Structures (ADS)

An authenticated data structure (ADS) is a security protocol in a client-server setting that allows a data-owner client to outsource her data storage to a third-party host (server) and to allow a data-user client to query it. In a public-key setting, a data owner holding a secret key can initially sign and later update the dataset, and a user trusting the owner's public key (e.g., through an external PKI [43]) can verify the authenticity of query result. While there is recent research [74, 68, 54] to design ADS for expressive queries, we consider in this work the most foundational form of ADS, that is, an authenticated dictionary supporting set-membership queries [20], which is well suited for a key-value store system.

Existing ADS constructions [45, 67] 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 [45, 74] and cached ADSes [37]; 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 [46] and authenticated databases [45].

2.3 Intel Software Guard eXtension (SGX)

Intel SGX is a security-oriented x86-64 ISA extension on the Intel Skylake CPU, released in 2015. SGX provides a "security-isolated world" for trustworthy program execution on an otherwise untrusted hardware platform. At the hardware level, the SGX's trusted world or enclave includes a tamper-proof SGX CPU which automatically encrypts memory accesses upon cache write-backs. Programs executed outside the enclave trying to access enclave memory only get to see the ciphertext and cannot succeed. At the software level, the SGX enclave includes only some unprivileged program and excludes any OS kernel code, by explicitly prohibiting system services (e.g., system calls) inside the enclave.

To use the technology, a client initializes an enclave by uploading her program to the server host and uses SGX's seal and attestation mechanism [21] to verify the correct setup of the enclave environment (i.e., the binding between the client's program and a genuine SGX CPU). 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/reloading in both hardware and software levels. Comparing prior TEE solutions [14, 17, 4, 11], SGX uniquely support multi-core concurrent execution, dynamic paging, and interrupted execution.

3 Research Formulation

In this section, we formulate the research by presenting the application scenarios, system and threat models, and a baseline design to introduce the proposed approach.

3.1 Application Scenarios

The target of this work is security-sensitive applications that require random-access data reads and writes. A conventional example is serving the public key directory in PKI [43], which requires the security of data authenticity (C1) that revoked keys should not be served and random-access queries (C2) where specific person's public key is queried. Modern security workloads are more dynamic and may feature an intensive stream of data updates (C3). The following is a list of real-world relevant applications.

- In health-care information exchange [16, 8] (A1), hospitals may want to outsource the storage of their patient medical records (EMR) to a federated third-party cloud for easy EMR discovery.
- In community-based code development (A2), a Github-alike service may benefit from outsourcing its open-source code repository to the public cloud for lower cost and higher availability.
- In enterprise computing (A3), a Foursquare [7] alike startup may want to outsource its customer's social dataset to the cloud for its cost-effectiveness.
- In smart-city (A4), an open government may want to outsource its big-data dataset about citizens (e.g., homeowner information, house electricity readings, etc.) to the public cloud with high availability. This practice increases the government transparency.

These application scenarios have all the three characteristics. For instance, in A1, intensive data updates are generated as any patient checks in a clinical place (C3).

A specific patient's records may be queried about during her clinical visit (C2). And missing a patient's record (C1) will lead to incomplete knowledge and uninformed diagnostic decisions by her doctor. In general, for big-data applications, data updates are intensively generated as the big-data is continuously produced and collected (C3). The data users are usually only interested in a limited part of the data, and such interest is addressed by random-access queries (C2) addresses this need. The data-authenticity(C1), especially the guarantee of membership authenticity as will be described, is of critical importance to the application security. We note that in some of these applications, data confidentiality is optional, as they involve open datasets.

3.2 System Model

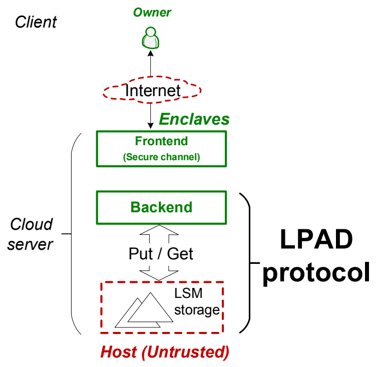


Figure 1: LPAD protocol in an overall system: The boxes with solid lines (in green) mean trusted domains including the enclave and owners. The shapes with dotted lines (in red) mean untrusted host domains including Internet and storage host. This work focuses on the secure interaction on the storage backend, formally the security protocol of LPAD, that occurs between enclaves and untrusted host.

We abstract the applications described above and present our system model: The owner of a security-sensitive dataset outsources the data storage to a third-party host. The owner then submits read and write requests to the outsourced dataset. For simplicity, we consider it is the data owner who submits read requests. In practice, the data owner can share her public key with another party to grant to her the data-read permission.

The host runs SGX enclaves established by the owner. An enclave includes the SGX CPU at the hardware layer and the program uploaded by the owner at the software layer. The other parts of the host are untrusted by the owner.[4] In particular, the storage medium such as the disk is in the untrusted domain.

[4]In the rest of this paper, we use the term "host" to mean the untrusted parts of a host.

The host may launch multiple enclaves: On the frontend, the host communicates with the data owner through a secure Internet channel established by an enclave; this enclave runs software attestation and TLS handshake protocol. On the backend, another enclave receives data-access requests from the application and interacts with the untrusted host storage. The host runs an LSM-based storage system (e.g., Google LevelDB) to persist the intensive update streams and to serve random-access reads. As mentioned, the LSM-based storage lends itself to this type of modern workload. Figure 1 illustrates the overall system architecture and the focus of this work (as will be explained soon).

The storage interface supported in enclave follows the standard key-value store API: Given data key k, value v, timestamp ts, a write operation $\mathsf{Put}(k,v)$ sends a data-update request to the outsourced key-value store and returns an acknowledgment about committed timestamp ts. A read operation $\mathsf{Get}(k,ts_q)$ sends a data-read request to the key-value store and returns result record $\langle k,v,ts\rangle$. Formally,

$$\begin{aligned} ts &:= \mathsf{Put}(k,v) \\ \langle k,v,ts\rangle &:= \mathsf{Get}(k,ts_q) \end{aligned} \tag{1}$$

Scope of this work: This work focuses on the storage backend of data outsourcing. We particularly aim at the secure interaction between the backend enclave and LSM-based storage host.

The frontend design and security are out of the scope of this work. For completeness, we do assume external mechanisms are in place to secure the frontend; existing information-security (InfoSec) mechanisms used in practice can help establish secure-channels including various techniques of key-distribution [43] and software attestation [21].

3.3 Threat Model

The adversary in this work attacks the storage backend of the host. The adversary can manifest in the forms of an untrusted operating system, a malicious user-space program or even hardware components in the untrusted domain. The attack goal is to compromise the membership authenticity. For instance, given a Get request, a malicious host can forge a result (breaking query-result integrity), or present a stale record (violating query freshness), or skip a legitimate record (violating query completeness). Given a Put request, the malicious host can send acknowledgment without actually persisting the data.

Formally, result integrity is about whether read-result $\langle k,v,ts\rangle$ is a key-value record written by a legitimate write request before. This is a special case of message integrity (in the classic secure channel [43]), and the integrity can be easily protected by message authentication code (MAC). Query membership in a key-value store is about whether a read result is fresh and complete. 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.

Non-goals of our research work include rollback attack and security, denial-of-service attacks, SGX side-channel attack and security [72], enclave program security and memory safety [36], hardware attacks to break SGX CPU tamper-resistance [35].

For applications that data confidentiality is a concern, we assume the key-value records are deterministically encrypted [43], such that the ciphertext domain has an ordering and normal key-value store operations can be processed on the ciphertext.

3.4 Baseline and Observation

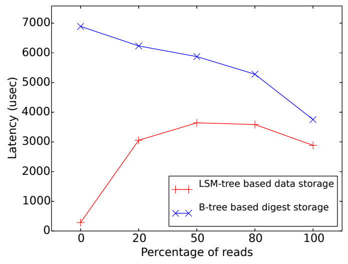


Figure 2: Performance of LSM-tree based data storage and B-tree based digest storage (i.e., existing ADS)

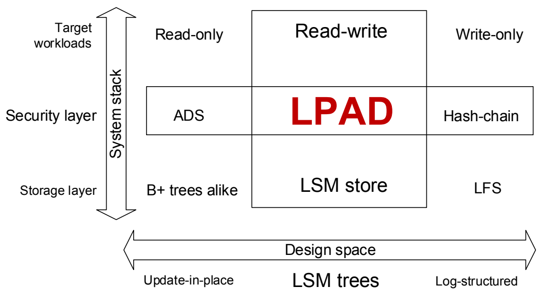


Figure 3: LPAD and alternative designs: (LFS stands for log-structured file systems).

Our security goal is to ensure the data authenticity, especially the membership authenticity under the threats mentioned above when the enclave interacts with the storage host on the backend.

A baseline approach is to use the existing protocols of Authenticated Data Structures (ADS) to specify and formalize the enclave-host interaction. While the conventional setting of ADS is the client-server architecture, an ADS protocol can be easily adapted to fit in the enclave-host architecture (with the enclave being the data owner and host being the server). As mentioned in § 2.2, existing ADS constructions rely on update-in-place data structures, such as a B-tree alike Merkle tree [50]. This means at the security layer, the data reads and writes follow an update-in-place structure while at the data-persistence layer (in the untrusted domain), the data reads and writes follow an LSM-based structure.

This data-structural mismatch will create the performance problems. First, the presence of update-in-place structures adds random disk accesses to the data-write path which offsets the optimization by the underlying LSM store. Second, the mismatch between the structures at the security and data-persistence layers renders it impossible to co-locate data and digest physically, which increase the number of disk "seeks".

To demonstrate the performance, we implemented the ADS in an B-tree format [45] and the data storage in an LSM tree. These two components are invoked synchronously in overall authenticated data storage. Key-value records are 100 bytes while the digests are SHA256 (i.e., 256 bits). We measure the average latency of data accesses separately on these two components and report the results in Figure 2; The two data structures have very different performance profiles. On the write-intensive workloads (i.e., with a small percentage of reads), the LSM tree has a clear performance advantage (lower latency) over the B-tree based ADS. Coupling two data structures together in an overall system would offset this advantage.

Research statement: Based on the observation, we present the research statement of this work: Building membership-authenticated log-structured data storage on Intel SGX enclaves with three salient features: 1) formal security, 2) low overhead, 3) small application-level TCB.

4 LPAD Protocol

In this section, we present the proposed LPAD technique from the *protocol*'s perspective. The purpose is to describe the enclave-host interaction at an abstract level, such that the protocol can be presented formally and precisely for the security purpose. The systems-oriented description is left the following sections.

An LPAD protocol is essentially an ADS tailored to the LSM tree structure. We thus first present the formal model of an LSM tree and then describe the LPAD protocol and its variants.

4.1 Model of an 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 $l_i = b_1 b_2 ... b_j ...$. In the beginning, all data reside on level l_0 and as new records are added, data are migrated to higher levels through the MERGE operation. An LSM tree supports the data reads and writes, where a write only updates the first level l_0, and a read may iterate through all levels until a match or non-match is resolved.

The MERGE operation migrates data at the units r across levels. It merges ranges from one or two adjacent levels, $r_0, r_1, ...$, into one range at one level, that is, $\varnothing, ..., \varnothing, r'_{kk} := \text{MERGE}(r_0, ..., r_{kk-1}, r_{kk})$. It is up to the application to specify how big the migration data-unit it is. An example policy is leveled merge/compaction [9] where the data unit is an entire level, and it always merge a lower level with the overlapping part of a higher level. For another example, a size-tiered compaction policy [5] organizes each level by multiple files in the time order, where each file is a sorted run by key. The timed compaction is to merge different files at the same level and to flush the result to the higher level. Some notations used in this paper are in Table 1.

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

4.2 LPAD Protocol

An LPAD protocol runs in an ecosystem consisting of two parties: a trusted enclave and an untrusted host for data storage. Recall that before the LPAD protocol, we assume a secure channel is established between the client-side data owner and the server-side enclave. Through the channel, the enclave obtains the original copy of dataset sent from the owner. Thus, the LPAD protocol starts with the initial state that the enclave has the original owner's dataset. In addition, the server-side enclave in LPAD also represents the data owner.[5]

The purpose of the LPAD protocol is to store the owner's dataset on the untrusted host and to serve the enclave's data-access queries with security in membership authentication. For simplicity, we consider a single owner/enclave in describing the LPAD protocol.

Formally, given dataset m and record b, the target query is a set-membership predicate, $0, 1 = P(m, b)$ which returns 0/1 representing the non-membership/membership of the record in dataset m. The protocol runs in four interactive rounds, described as below.

[5]We use the terms "owner" and "enclave" interchangeably.

1. The enclave generates a pair of public/private keys based on security parameter n (i.e., algorithm $pk, sk = \text{GEN}(1^n)$), and then signs the initial dataset m[6] by secret key sk (i.e., algorithm $s = \text{SETUP}_{sk}(m)$[7]). The enclave then stores the signed dataset (s) in the host, along with public key pk.

2. On the read path, the untrusted host runs an authenticated query-processing routine (i.e., algorithm $\pi, a = \text{QUERY}_{pk}(m, b)$) that, in addition to finding the result, prepares a query proof π by including a membership proof for the LSM-tree level that contains the answer, and more importantly, by including non-membership proofs for the tree levels that don't contain the answer. In this work, we use the standard Merkle tree [50, 51] to construct the LPAD protocol.[8] Under this construction, the proof of both membership and non-membership is in the form of the so-called authentication path [50]: An authentication path of a Merkle leaf node is the list of tree-node hashes that are neighbors to the path from the root to the leaf.

Then, the enclave receiving proof π and answer a verifies the integrity of the answer by iterating through all levels, recomputing the root hash for each level, and checking the equality between the recomputed roots and the dataset digest from signature s.

3. On the write path, the enclave, on behalf of the client owner, updates the dataset. The enclave updates the first level l_0 with new record b, resulting in a new dataset signature s' (i.e., algorithm $l'_0, s', upd = \text{UPDATE}_{sk}(b, l_0)$). The algorithm may retrieve an authentication path from the host, perform the update, and obtain the new image of first level l'_0 and signature s'.

Then, the untrusted host refreshes the first level based on the new record b and update information upd (i.e., algorithm $l'_0 = \text{REFRESH}_{pk}(b, l_0, upd)$).

4. Periodically, the enclave and host interactively run algorithms to migrate data across levels. The LPAD protocol specifies the MERGE mechanism at the granularity of levels and leave it open to various policies to specify the actual data units r used. The enclave updates the two adjacent levels (l_i, l_{i+1}) to posterior state (l'_i, l'_{i+1}), with signatures and update information upd (i.e., algorithm $l'_i, l'_{i+1}, s', upd := \text{SIGMERGE}_{sk}(l_i, l_{i+1}, s)$). The host server then merges data in two adjacent levels (l_i, l_{i+1}) to (l'_i, l'_{i+1}) and simply updates their signatures using update information upd (i.e., algorithm $l'_i, l'_{i+1}, s' := \text{MERGE}_{pk}(l_i, l_{i+1}, s, upd)$).

Specifically, $\text{SIGMERGE}(l_i, l_{i+1})$ is constructed by the enclave retrieving from the host the two input levels, l_i

[6]Recall that the enclave obtains the genuine copy of dataset m from the owner through the secure channel.

[7]We use the subscription to refer to the public/private keys used in the algorithm.

[8]Despite the Merkle tree construction in this work, we stress the LPAD scheme is generic and can be constructed by other ADS primitives (e.g., multi-set hash [33]).

and l_{i+1}, and linearly scanning them. This straightforward construction with linear cost may not be feasible in the traditional client-server setting, but is practical in the TEE case where the server host is co-located with the enclave.

The above formalization supports MERGE policies with different data units in a straightforward way.

The correctness of LPAD scheme is straightforward and similar to that of ADS [54]; informally, the correctness can be stated by that in any state resulted from calling UPDATE/REFRESH and MERGE/SIGMERGE, given any query against the state, verifying the query result will accept. The security of LPAD scheme, informally, requires that any adversary who compromises the host cannot forge invalid query results and trick the VRFY algorithm to acccept. The formal definition and security/-correctness of the protocol construction are described in the technical report [15].

4.3 Read-optimized Protocol

The basic LPAD construction above needs to read all levels upon QUERY (we denote this construction by LPAD-AllLevel). In the real-world LSM tree, one of the optimizations is to stop the query processing as early as the level the first hit is found. To incorporate this read optimization, we extend our model by richer semantics and design the next construction, LPAD-R.[9]

In LPAD-R, each record is associated with a timestamp, denoted by (b, ts). The value of a timestamp is assigned by an ideal "clock" which monotonically increases the newest timestamp. The read query is extended to return the latest record where the "latest" record is defined by having the largest timestamp among records of the same value. Note that by assuming a global clock, we assume all the writes that generate timestamp are synchronized, which as we will see is the case for Google LevelDB [9].

In LPAD-R, processing QUERY(m, b) stops when the first level, say l_i, is found to have (b, ts). In this case, $\pi = \pi_0, \ldots \pi_i$, where $\pi_0, \ldots \pi_{i-1}$ are proofs of non-membership and π_i is proof of membership. When $i = q$, it means $b \notin m$ and it is the same proof with that used in the basic construction.

Before analyzing the correctness and security, we present a property about the temporal order of records in different levels in LPAD.

Theorem 4.1 *In an LPAD, a record stored in a lower-numbered level has a larger timestamp than that of a record in a higher-numbered level. That is, given records (b, ts) and (b', ts') respectively residing in levels l_i, l_j, if $i < j$, $ts > ts'$.*

[9]LPAD-R is used as the standard construction and it will be used interchangeably with LPAD.

For instance, Figure 4 illustrates an LSM tree of three levels: l_0, l_1 and l_2. At any level, say l_2, records are sorted, say from key A to T to Z. Theorem 4.1 requires that in Figure 4, an older record A with timestamp 2, is stored on a higher-numbered level l_2 than the level a newer record $(A, 9)$ is stored (which is level l_0).

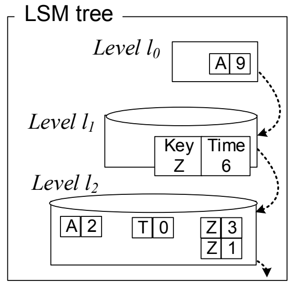


Figure 4: An example LSM tree

The proof sketch of Theorem 4.1 is that LPAD only allows to move records from lower-numbered levels to higher-numbered ones, but not in the reversed order. We formally prove the property in the technical report [15].

Correctness: The correctness of this optimization means that when VRFY$(\pi, a) == 1$, (b, ts) must be the latest record in the dataset. It can be proven that VRFY$(\pi, a = 1) == 1$ would require $\exists i$ s.t. $a_{l_i} = 1$. The correctness of ADS(l_i) in membership queries implies that b must be present in l_i, thus present in m. In addition, the correctness of ADS$(l_{<i})$ in non-membership queries implies that b must be absent in these levels. Due to Theorem 4.1, any records in $l_{>i}$ will be older than record (b, ts) in level l_i.

Security: The security of the optimized construction means that an adversary cannot forge a proof for record that is not the freshest in the system. The security comes the security of ADS in both membership and non-membership queries. Given proof π about record b at level i, it is unfeasible for an adversary to forge a proof of membership of b in $l_{<i}$ or to forge a proof of non-membership of b in l_i.

5 LPAD-based Storage on SGX

In this section, we present the engineering of LPAD protocol on the architecture of Intel SGX, and study the software-partitioning problem.

We build our system on Google LevelDB [9], which is a representative and widely adopted storage system based on the LSM trees. We stress that in this paper we abuse the term "LevelDB" to represent a broad class of log-structured key-value stores, such as Apache HBase [3], Apache Cassandra [2], Facebook RocksDB [6]. In addition to following the LSM design, these stores are similar to LevelDB in terms of many software-engineering patterns, including using memory buffers to facilitate data reads on non-zero levels, persisting level zero using write-ahead log. These features will be discussed in the context of LevelDB but readers should be aware that they are generic to this class of key-value stores.

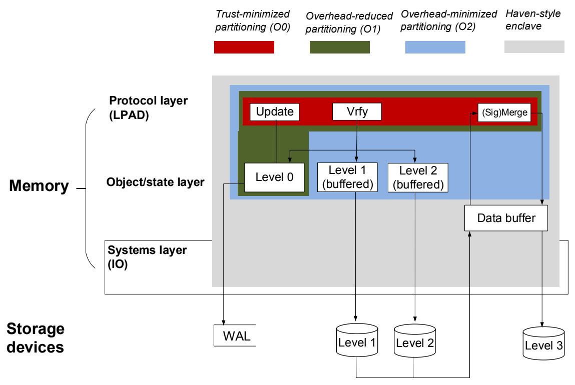


Figure 5: Partitioning the system at protocol and memory-object layers: Strategy **O0** places inside enclave the necessary functionality (the red rectangle), that is, the trusted algorithms of LPAD schemes (it omits GEN and SETUP in the Figure). Strategy **O2** places inside enclave as many memory "objects" as possible including those accessing data in levels l_0 and $l_{\geq 1}$ (the dark blue rectangle). Strategy **O1** is a middle ground between O0 and O2 which placing the object of l_0 in enclave while leaving others out (the light blue rectangle).

For simplicity, we consider the single-threaded setting of LevelDB; we leave our multi-threaded extension to the technical report [15]. Reads and writes are submitted to LevelDB by a single thread and are thus fully serialized. The security definition naturally follows the description in § 3.3. We present an abstraction of LevelDB and then a variety of software-partitioning strategies are devised.

LevelDB abstraction: The codebase in the LevelDB follows the model of LSM tree (§ 4.1) and can be split into three types of "memory objects" serving reads/writes. Here, we use the term "memory object" to refer to a memory region that stores closely coupled code and data and that encapsulates certain functionality. The three memory objects in a LevelDB are:

J1 is the object relevant to reading/writing data stored in level l_0. In this object, the primary data is at level l_0 and resides in memory when serving reads and writes. Periodically, it also logs data to disk for failure tolerance through a data structure called Write-Ahead Log (WAL).

J2 is the object relevant to reading immutable levels $l_{\geq 1}$. In this object, the data is stored on disk and served through IO. There is a memory buffer that improves performance by leveraging locality.

J3 is the object relevant to merging the immutable levels. In this object, similar to J2, data is stored primarily on disk and IO occurs upon data access.

Figure 5 illustrates this abstract view of LevelDB. Note that each object involves the disk-IO access that must be placed outside an enclave.

The problem of software partitioning in LevelDB is to partition the *application* program and to runs the two partitions in the two worlds of SGX. In this section, we study the partitioning strategies of LevelDB by placing its objects inside/outside an enclave; these strategies make the tradeoff between minimal trust and overhead.

5.1 Trust-minimized Partitioning (O0)

An intuitive instantiation of the LPAD protocol follows the conventional rules that only secrets should be confined inside the enclave [44, 29, 56]. In an LPAD protocol, the secrets are the secret keys and the dependent operations. This results in the trust-minimized partitioning scheme:

Partitioning scheme O0 places five LPAD algorithms (and nothing else) inside the enclave, that is, GEN,SETUP,UPDATE,SIGMERGE, VRFY— the first four have a dependence on secret keys and the last one produces security-critical results. The algorithms in enclave are dependent only on the states of sublinear sizes, such as the proof in VRFY. Figure 5 illustrates the O0 partitioning scheme.

Concretely, for Object J1, the memory data at level l_0 is stored outside enclave. On the write path, Algorithm REFRESH is run outside the enclave that updates Level l_0. It then notifies the enclave to run Algorithm UPDATE. On the read path, the data at level l_0 (in Object J1) and all other levels (in Object J2) are stored outside enclave. Algorithm QUERY is run to read all levels outside the enclave. After that, the enclave runs Algorithm VRFY to authenticate the read result. For Object J3, we place inside enclave the actual computation of MERGE. An MERGE computation reads multiple data streams from disk, store them on a buffer, performs in-memory computation and writes the result to disk. In O0, we place the MERGE data buffer outside enclave, as it has no locality.

MERGE implementation: We implement J3 by three threads: The first thread residing outside enclave read data from disk to an input buffer. The second thread residing inside the enclave performs the merge computation. And the third thread residing outside the enclave persists the result data from the buffer to disk. The three threads share data buffers and synchronize their execution by checking the buffered data size. The second thread performs three functionalities: 1) the computation of MERGE, 2) the authentication of the input data streams, 3) signing the output data. We implement them in one data pass by exploiting incremental Merkle trees [69]. Our implementation also retains the LevelDB functionality, including versioning policies, tombstone delete, etc. Details about implementing verifiable MERGE are in the technical report [15].

Security analysis of O0: Suppose reads and writes are fully serialized. An invariant of partitioning strategy O0 is that on both write and read paths, *the LPAD algorithms inside enclave (i.e.,* UPDATE *and* VRFY*) occur after the algorithms running outside enclave (i.e.,* REFRESH *and* QUERY*)*. This invariant, with the promise of fully serialized execution,[10] allow 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 $\langle k, v, ts \rangle = \mathsf{Get}(k, ts_q)$ is fresh as of timestamp ts_q. By definition, it can be authenticated by the membership of the result, $\langle k, v, ts \rangle$ and the non-membership of any record, say $\langle k, v', ts' \rangle$ that is "fresher" than the result and with $ts' \in [ts, ts_q]$. Both the membership and non-membership can be further authenticated by the LPAD scheme. Based on the freshness authentication, any stale result returned from the untrusted host can be easily detected (by the failure of VRFY).

The freshness attack can be extended to different variants: 1) In a query-completeness attack, 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 [49] 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 that can be authenticated, thus eliminating the forking vulnerability.

5.2 Overhead-minimized Partitioning (O1,O2)

The partitioning schemes presented in this subsection aims at minimizing the overhead of boundary crossing and do so by placing as many objects into enclave as possible. This design is motivated by an observation of two-fold: 1) The more data is placed in an enclave, the less it needs to access data on the untrusted host, saving the verification costs (at software level). 2) The more code is placed in an enclave, the less it needs to switch the execution to outside the enclave, saving the world-switching overhead. Based on this observation, we propose two partitioning strategies:

Partitioning scheme O1 places only the memory object of J1 inside enclave. In particular, the disk-IO part of J1 (for accessing WAL) is placed outside enclave, and the memory part of J1 for accessing data stored in level l_0 is placed inside enclave. In O1, accessing data in level l_0 does not cause boundary crossing, thus saving overhead. For partitioning Object J3, O1 is the same with O0.

Partitioning scheme O2 places the memory objects of J1 and J2 inside enclave J1 is similarly placed in enclave with O1. In addition, the disk-IO part of J2 for accessing data stored in levels $l_{\geq 1}$ is placed outside enclave, and the memory part of J2 for accessing memory buffers is placed inside enclave. Thus, the world switches caused by accessing the buffers of higher-numbered levels $l_{\geq 1}$ can be saved. Particularly, if the workload of data accesses exhibit a high level of locality, O2 is effective in saving overhead as the memory buffer in J2 will be accessed repeatedly without boundary crossing. For partitioning Object J3, O1 is the same with O0.

Both O1 and O2 increase the trusted code size comparing O0. The details of these two overhead-reduced partitioning strategies are illustrated in Figure 5.

Security analysis of O1,O2: Both O1 and O2 preserve the invariant 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). As analyzed in O0, the invariant associated with the promise of serialized execution allows the enclave to fully specify the read/write execution history and establish a total-order on which the freshness can be authenticated.

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 and memory-intensive workloads (§ 6.2)?

We also conducted micro-benchmark experiments to study the impact of LPAD design at a fine granularity. Due to space limits, the micro-benchmark results are in the technical report [15].

6.1 Implementation & Enclave Code Size

We implement LPAD partitioning schemes by modifying LevelDB with the following changes: 1. We wrap the enclave programs by ECalls provided in the Intel SGX SDK [13], where the ECall arguments are passed by

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

pointers. 2. For MERGE, we run the enclave algorithm in a separate thread (described in § 5.1) that share a memory with the untrusted threads. We implement the thread synchronization by enforcing mutex on these shared memories. 3. We implemented the LPAD algorithms using the SHA3 hash algorithm from the Crypto++ library [10], and map the algorithms to the enclave-host architecture based on specific partitioning schemes. In particular, we store the Merkle trees in the untrusted LevelDB storage by encoding the hash digests in the LevelDB data records and in-file index. 4. We modify LevelDB to make each Put return its timestamp for records serialization. The change is not significant and does not cause overhead.

For evaluating the trusted code size, we prepare a baseline by loading the unmodified LevelDB code into an in-enclave library OS, Panoply [65]. We stress Panoply is not an 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 trust-minimized partitioning (O0) place inside the enclave trusted LPAD algorithms (SIGMERGE,UPDATE,VRFY) and require the enclave to include the code for Merkle proof and SHA computation, in addition to the glue code generated by Intel SGX SDK [13]. The total number of code lines in enclave is around 900. Comparing the baseline approach, trust-minimized partitioning reduces the application-level trusted code size by 20 times.

In the overhead-reduced partitioning (O1), level l_0 and associated code are included in an enclave. The level l_0 is stored in a skip list, which increases the trusted code size from 900 (of O0) to around 2500 — there are 400 lines for data access in a skip list and 1600 for the extra glue code.

The overhead-minimized partitioning further adds the higher-numbered levels into an enclave, which increases the trusted code to 4100 lines. Nevertheless, it is 5 times smaller than the Haven-style solutions. 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)
Trust-minimized (O0)	891
Overhead-reduced (O1)	$891 + 1588$
Overhead-minimized (O2)	$891 + 1588 + 1591$
Haven/Panoply-style (Baseline)	~ 20000

6.2 Performance Evaluation

In this section, we present the performance of LPAD under Yahoo Cloud Serving Benchmark (YCSB) [34] which is a standard benchmark suite. We evaluate the performance under IO-intensive workloads and memory-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 [34] as a workload generator evaluating the performance of generic key-value stores. We leverage the LevelDB-YCSB adapter based on online projects.[11] In our experiment, we run the YCSB workload driver on one machine and the storage system on another machine; the two machines are connected to 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.[12]

6.2.1 IO-intensive workload

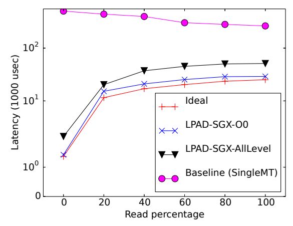
In the experiment, we varied the read percentage from 0% (that is, a write-only workload), 20%,40%,60%,80% to 100% and we tested 1 million queries. In the benchmark, we turned on the MERGE, used different record sizes (116-byte and 1016-byte records). We used the SHA3 hash algorithm. We run the workload in a single thread. Each experiment is run at least three times. The result in latency is reported.

We test LPAD-SGX with different partitioning schemes and compare their performance with 1) the baseline of running a single Merkle tree over the dataset, and 2) the ideal solution (in the sense of ideal performance) of running the original, insecure LevelDB system.

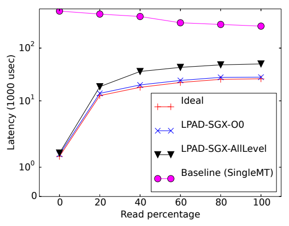
The performance result under the IO-intensive workload is presented in Figure 6a and Figure 6b. For both 1016-byte and 116-byte record sizes, the LPAD-SGX scheme matches well with the write-optimized characteristics of the original LevelDB – their latency increases as the workload becomes more read intensive. By contrast, the baseline of a single Merkle tree exhibits a

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

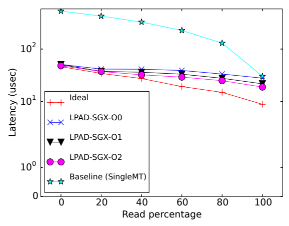
[12]Note the smaller size a value is (e.g., 100 byte), the more challenging to serve for a storage system as small writes cause more random access IO.



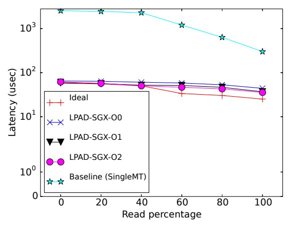
(a) IO-intensive workloads (116-byte records, SingleMT stands for the baseline approach of a single Merkle tree for authentication)



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



(c) Memory-intensive workloads (116-byte records)



(d) Memory-intensive workloads (1016-byte records)

Figure 6: LPAD-SGX performance

read-optimized behavior. Moreover, 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 baseline (the single Merkle tree). Our LPAD-SGX leverage the read-optimized construction (LPAD-R) that 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 an underlying storage system. Note the LSM tree and LPAD protocol are about optimizing storage IOs, and IO-intensive workloads are the best setting to show the effectiveness of LPAD.

6.2.2 Memory-intensive Workload

We use the small dataset to test the performance under the memory-intensive workload. When the memory access dominates the performance, the placement of objects in/outside an enclave (or the partitioning) becomes relevant. In this experiment, we thus evaluate the performance of different LPAD-SGX partitioning strategies in the presence of memory-intensive workloads. This experiment is run in a single thread.

The performance result is presented in Figures 6c and 6d. With a small record size (116 bytes), partitioning strategy O2 has a performance advantage over O0 by up to 33%, while causing 125% slowdown comparing the ideal performance. With a larger record size (1016 bytes), O2 improves the performance of O0 by up to 20% and has a slowdown of 40% to the ideal performance.

7 Related Work

This section presents a short version of related work. We leave the full version in the technical report [15] due to the space limitation.

Software Systems on SGX: There are general-purpose system supports in enclave, including Haven [26], Graphene-SGX [70], SCONE [22], Panoply [64], which exports a library OS interface and runs legacy programs in enclaves.The side-channel vulnerabilities have been discovered on Intel SGX [72, 30, 41, 27]. There are defense mechanisms proposed [63, 32, 63]. A series of special-purpose systems are proposed and built in enclaves, such as VC3 [60], Opaque [75], oblivious machine-learning [52], CorrectDB [25], etc; these systems achieves advanced side-channel security in a way tailored to specific applications. In particular, HardIDX [39] is a secure storage system with enclave which seals data using authenticated encryption [43]. LPAD is a storage system with authenticated membership that follows the LSM tree design.

LSM Storage Systems: bLSM [61] optimizes the LSM tree performance by row-based data storage and fine-grained compaction. Prior work [42] minimizes the write amplification under the skewed key access pattern. Pebble [57] reduces the write amplification by organizing storage layout in skip lists and avoiding data rewriting in the same level. Distributed compaction management is proposed for better performance [18]. Beyond disk storage, the LSM tree has been applied for main-memory databases with high compression rate [73], on non-volatile memory [48], and for spatial databases in the AsterixDB project [19]. Concurrency of the LSM tree is studied in cLSM [40] that supports snapshot scan, conditional update, and concurrent merge operations.

8 Conclusion

This work presents membership-authenticated log-structured storage on hardware enclaves. The proposed techniques include the LPAD protocol that achieves low overhead and proven security, a software system built based on LPAD that supports various software-partitioning strategies to optimize between minimal TCB and lower overhead. With real implementation and extensive performance studies, we demonstrate the LPAD system has a small enclave program, achieves low overhead and provide security in membership authenticity.

References

[1] AMD SP, http://www.amd.com/en-us/innovations/software-technologies/security.

[2] Apache Cassandra, http://cassandra.apache.org/.

[3] Apache HBase, http://hbase.apache.org/.

[4] ARM TrustZone, https://www.arm.com/products/security-on-arm/trustzone.

[5] Compaction in Apache Cassandra, https://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra.

[6] Facebook RocksDB, http://rocksdb.org/.

[7] Foursquare, https://foursquare.com/.

[8] Gahin: http://www.gahin.org/.

[9] Google LevelDB, http://code.google.com/p/leveldb/.

[10] http://www.cryptopp.com/benchmarks.html.

[11] IBM SCPU, http://www-03.ibm.com/security/cryptocards/.

[12] Intel sgx programming reference, 2014 no. 329298-002, `https://goo.gl/n9DgHX`.

[13] Intel software guard extensions (intel sgx) sdk.

[14] Intel TXT, http://www.intel.com/technology/security/downloads/trustedexec_overview.pdf.

[15] Lpad: Membership-authenticated log-structured storage with hardware enclaves, full version, `https://goo.gl/zFma56`.

[16] Nhin: http://www.hhs.gov/healthit/healthnetwork.

[17] TPM, http://www.trustedcomputinggroup.org/tpm-main-specification/.

[18] M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. *PVLDB*, 8(8):850–861, 2015.

[19] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in asterixdb. *PVLDB*, 7(10):841–852, 2014.

[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. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer. SCONE: secure linux containers with intel SGX. In K. Keeton and T. Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 689–703. USENIX Association, 2016.

[23] 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.

[24] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1):70–122, 1998.

[25] S. Bajaj and R. Sion. Correctdb: SQL engine with practical query authentication. *PVLDB*, 6(7):529–540, 2013.

[26] 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.

[27] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: SGX cache attacks are practical. *CoRR*, abs/1702.07521, 2017.

[28] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 341–357, 2013.

[29] D. Brumley and D. X. Song. Privtrans: Automatically partitioning programs for privilege separation. In M. Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 57–72. USENIX, 2004.

[30] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In E. Kirda and T. Ristenpart, editors, *26th USENIX Security Symposium, USENIX*

Security 2017, Vancouver, BC, Canada, August 16-18, 2017., pages 1041–1056. USENIX Association, 2017.

[31] 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.

[32] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 7–18. ACM, 2017.

[33] 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.

[34] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.

[35] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[36] D. Dhurjati, S. Kowshik, V. S. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *ACM Trans. Embedded Comput. Syst.*, 4(1):73–111, 2005.

[37] 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.

[38] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 2nd Edition*. Benjamin/Cummings, 1994.

[39] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A. Sadeghi. Hardidx: Practical and secure index with SGX. In G. Livraga and S. Zhu, editors, *Data and Applications Security and Privacy XXXI - 31st Annual IFIP WG 11.3 Conference, DBSec 2017, Philadelphia, PA, USA, July 19-21, 2017, Proceedings*, volume 10359 of *Lecture Notes in Computer Science*, pages 386–408. Springer, 2017.

[40] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 32:1–32:14, 2015.

[41] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 299–312. USENIX Association, 2017.

[42] C. M. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *VLDB J.*, 16(4):417–437, 2007.

[43] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[44] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.

[45] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD Conference*, pages 121–132, 2006.

[46] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI*, pages 121–136, 2004.

[47] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. M. Eyers, R. Kapitza, C. Fetzer, and P. R. Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 285–298. USENIX Association, 2017.

[48] L. Mármol, S. Sundararaman, N. Talagala, and R. Rangaswami. NVMKV: A scalable, lightweight, ftl-aware key-value store. In S. Lu and E. Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 207–219. USENIX Association, 2015.

[49] 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.

[50] R. C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[51] R. C. Merkle. A certified digital signature. In *Proceedings on Advances in Cryptology*, CRYPTO '89, 1989.

[52] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 619–636. USENIX Association, 2016.

[53] 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.

[54] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, 74(2):664–712, 2016.

[55] 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, 2013.

[56] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.

[57] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 497–514. ACM, 2017.

[58] M. Rosenblum. *The Design and Implementation of a Log-Structured File-System*. Kluwer, 1995.

[59] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury. Automated partitioning of android applications for trusted execution environments. In L. K. Dillon, W. Visser, and L. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 923–934. ACM, 2016.

[60] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 38–54. IEEE Computer Society, 2015.

[61] R. Sears and R. Ramakrishnan. blsm: a general purpose log structured merge tree. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 217–228. ACM, 2012.

[62] 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.

[63] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In X. Chen, X. Wang, and X. Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 317–328. ACM, 2016.

[64] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. Panoply: Low-tcb linux applications with sgx enclaves. 2017.

[65] 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.

[66] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill Book Company, 2005.

[67] 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.

[68] R. Tamassia. Authenticated data structures. In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, pages 2–5, 2003.

[69] Y. Tang, T. Wang, L. Liu, X. Hu, and J. Jang. Lightweight authentication of freshness in outsourced key-value stores. In *ACSAC*, pages 176–185. ACM, 2014.

[70] C. Tsai, D. E. Porter, and M. Vij. 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.*, pages 645–658. USENIX Association, 2017.

[71] 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.

[72] 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.

[73] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1567–1581, 2016.

[74] 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.

[75] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 283–298, 2017.