Write-Optimized Consistency Verification in Cloud Storage with Minimal Trust

Yuzhe Tang Ju Chen

Dept. of EECS, Syracuse University, Syracuse, NY, USA
Email: {ytang100, jchen133@}syr.edu

***Abstract*—Today, data outsourcing to the clouds is a popular computing paradigm, and enabling efficient and trustworthy outsourcing becomes critically important as many emerging cloud applications are increasingly security-sensitive, such as healthcare, finance, etc. One of the promising techniques is authentication data structure (ADS). Most existing ADSs are not log-structured, yet cloud storage systems that work beneath the ADSs are log-structured – this structural mismatch leads to significant performance overhead.**

We propose log-structured ADSs for lightweight verification in cloud outsourcing. Our approach is leveraging recently available commercial TEE (trusted execution environment, such as Intel SGX). For security, only two functionalities are placed inside a TEE, that is, frontend consistency checking and backend maintenance computations, yielding a small TCB (trusted codebase). For performance efficiency, the ADS layer follows the log-structured design, resulting in small overhead. We implemented a working log-structured ADS system on LevelDB, and demonstrated a small TCB and small performance overhead ($6 \sim 12\%$ in IO-intensive workloads) through extensive performance studies.

I. Introduction

Today, outsourcing data storage to the public cloud becomes a popular computing paradigm, due to cost-effectiveness, efficiency, availability/accessibility, etc. This is evidenced by various cloud-storage services on the market [9], [1], [4], [3]. In the presence of potentially malicious clouds (e.g. driven by profit or compromised by external hackers), the trust to the cloud providers becomes crucial to the client's decision-making on cloud adoption. Ideally, cloud clients want the data outsourcing to be:

1) trustworthy in that all interactions to the outsourced cloud storage come with assurance (from some authority) that the cloud behaves honestly. Any operation properties that might be exploited must be verified in their correctness, such as query-result integrity, freshness, etc.

2) practically efficient in that the cloud's efficiency in processing a large volume of data is the main incentive for outsourcing and should not be sacrificed by the extra work to enforce security.

One of the most promising techniques to achieve these goals is the authenticated data structure (ADS), a cryptographic protocol that enables the verifiability essentially by computation hardness (e.g. hash collision resistance), yet is relatively efficient (in an asymptotic sense). Despite the extensive researches in this domain (e.g. tree-based ADS [77], [48], [60], [61], [90], [32], [57], [46], [59], [87], [88], signature-based ADS [52], [54], [53], [56], [30], [58]), existing ADS techniques fall short on close systems integration and enabling systems-level efficiency. This is especially the case when existing ADSs are update-in-place structures while the underlying storage systems follow a different philosophy – log-structured design with append-only updates (e.g. log-structured merge trees or LSM trees [55] used in various cloud storage systems [29], [8], [6]). When placing the update-in-place ADS over the log-structured storage substrates, the structural mismatch causes severe performance problems, such as slow-down by orders of magnitude (presented in § VI-B). Briefly, the cause of the slowdown is that the update-in-place ADS shifts the workload to be more read intensive (by adding reads to the write path), making the underlying log-structured storage less effective.

The lack of log-structured ADS is not by accident: The difficulty comes from supporting verifiable maintenance with practical performance. Deferred maintenance is observed by log-structured systems, and typically requires a linear or superlinear computation with the input of a large amount of the stored dataset (e.g. merging datasets in the case of an LSM tree). With a fully untrusted cloud, verifying the maintenance correctness efficiently and securely requires the client running a sublinear checking algorithm. And this is where existing theory-oriented approaches (e.g. proof-based verifiable computations [21], [62], [69], [28], [83], homomorphic digests/signatures [15], [27]) do not (yet) provide a practically efficient solution (e.g. with constant-sized data transfer between the cloud and client).[1]

We resort to systems-oriented solutions and relax the untrusted cloud model by considering a small trusted entity at the cloud side. The presence of a trusted entity in close proximity to the cloud is necessary, as it makes possible not only verifiable maintenance on the backend, but also the *immediate* verification of strong consistency on the frontend (detailed in § II-B). The cloud-side trust is make practically possible by the recent support of trusted execution environment (TEE) in commercial hardware, such as Intel Safe Guard Extension (SGX [11]). SGX allows a client to set up a security-isolated "world", called enclave, in the otherwise untrusted cloud; the client only trusts the CPU and trusted codebase (TCB) in the enclave.

We propose a protocol, called LPAD, for outsourcing log-structured storage with lightweight verification of data freshness. We model an LSM tree by multiple ordered lists supporting two operations, data reads/writes on the frontend and maintenance on the backend. The ordered lists are digested by a forest of Merkle trees [51], and data reads/writes are made verifiable by Merkle proofs. The maintenance is trustworthy

[1] In theory, the problem of efficient verifiable maintenance can be solved by a "merge"-homomorphic digest. For standard Merkle tree, such homomorphism is impossible [78], and for other digest structures, we believe the merge homomorphism is a theoretically open problem [78].

due to in-enclave execution. Our system architecture is designed with the goal of minimal user-space TCB and proven security; we demonstrate (in § VI-A) our system design has a TCB smaller than alternative state-of-the-art designs [25] by two orders of magnitude.

We build a working system materializing the LPAD design, on a real LSM storage system (Google's LevelDB [7]) and with Intel SGX CPU. At the systems level, the verifiable freshness is naturally extended to enable verifiable read-/write consistency under concurrent execution. To strongly consistent stores, such as LevelDB, we build a checker to verify the linearizability [39] in real-time. The consistency checker has lightweight overhead as its implementation has few synchronization points and is tailored to LevelDB's single-writer-multi-reader concurrency model. We explore the "code-partitioning" problem – data maintenance code is system-service intensive (due to the needs of data persistence) while SGX prohibits system services in its enclave [11]. We tackle the problem by "partitioning" the maintenance code-path at a place close to system calls while avoiding excessive world switches at runtime. This partitioning strategy improves performance efficiency by up to 6 times in our evaluation.

The contributions of this work consist of the following:

1) We identify the structural gap between existing update-in-place ADS protocols and log-structured storage systems underneath. This gap results in severe performance slow-down.

2) To bridge the gap, we propose LPAD, a formal protocol for log-structured ADS. We formally present a construction of LPAD using Merkle trees and hardware TEE features (trusted execution environments). We analyze the security and correctness of protocol construction. To the best of our knowledge, this is the first time to make Merkle tree scheme write-optimized by following an LSM design.

3) We build a working system of LPAD based on Intel SGX and atop LevelDB. We partition the codebase to minimize world-switches and runtime overhead. We build a lightweight consistency checker with minimal synchronization points. The built prototype system demonstrates a smaller *user-space* TCB[2] (by two orders of magnitude) than state-of-the-art SGX systems.

4) We conduct extensive performance evaluation and characterize the overhead of LPAD protocol and the consistency checker: In a disk IO intensive workload, the overhead is small and practically acceptable ($6\% \sim 12\%$ slowdown to an ideal non-secure system), which is significantly smaller than that of existing ADS implementation ($24X$ slowdown). This is the first time to demonstrate the *immediate consistency verification with practical overhead*.

II. Motivations

In this section, we present further details of the motivations for this work.

[2]We stress our software design minimizes the user-space TCB. Note the use of SGX keeps OS kernel codebase out of TCB, but does not necessarily mean a small TCB overall; an counterexample is Haven [25] which results in a large user-space TCB.

A. Why Combine ADS with LSM Trees

Preliminary: Merkle hash tree [51]: is a method of collectively authenticating a list of objects (that is, a set of objects with fixed ordering). We denote it by MHT. In an MHT, each leaf node is the hash of an object (or a key-value record), and an intermediate tree node is the hash digest of the concatenation of its direct children's hash digests. The root node digests the entire list and can further be signed. We call the root hash by Merkle hash. The authenticity of an object and its position in the tree can be verified by the digests of the siblings of the nodes that lie in the path from the root to the object's leaf node, a.k.a. the authentication path or Merkle proof.

Preliminary: ADSs: An ADS or authenticated data structure [77], [48], [60], [61], [90], [32], [57], [46], [59], [87], [88] is a protocol that formally describes interactions between a trusted verifier and untrusted prover. The goal of an ADS is to make certain properties (e.g. consistency) of the interaction verifiable to the verifier. A specific ADS can be characterized by the type of interactions supported. For instance, a hash-chain is an ADS that supports reads and writes only on the tail of the chain. A PAD or persistent authenticated dictionary [18], [37], using MHT, supports random access, that is, reads and writes at an arbitrary position of the dataset. Other ADSs support more complicated queries than point reads. In this work, we might use ADS to refer to PAD.

Update-in-place ADSs: All existing ADSs (PADs) [73], [36], [65] perform updates in place. That is, between a prover and a verifier, an ADS, say a remote MHT, updates itself by the verifier requesting a Merkle proof from the prover, modifying the proof and updating the MHT with a new root hash sent to the prover – In essence, this process updates the MHT "in-place." There are variants of ADSs, such as replicated ADS [46], [48], [51], [90] and cached ADS [33]; they replicate a certain part of the ADS on the verifier for better update performance. These optimizations do not change the nature of in-place updates. In addition, the update-in-place ADSs have been used in building verifiable systems, such as SUNDR [47] and MBTree [46] where a single MHT is used to digest the outsourced dataset and is updated in place. It is noteworthy that the client-synchronization based systems, including CloudProof [66], CONIKS [50], Caelus [43], etc., digest recent (after the last synchronization) updates by a log of hash-chain favoring write performance. However, the hash-chain is temporary and can not be used for immediate verification. Their permanent digest is still a single MHT updated in place.

The design of read-optimized ADS is fundamentally different from various write-optimized storage systems adopted well in the cloud: While read-optimized design features a single list updated in place, the write-optimized structure features append-only updates and multi-list data storage. This structural mismatch could result in severe performance slowdown when putting them into a single system (for verifiable storage). Our performance study demonstrates that the slowdown can be up to several orders of magnitude.

Initial performance observation: Our initial performance study (for motivation purposes) is designed to understand how much the storage IO is skewed by the presence of ADS? In write-only workloads, the storage IO would be skewed by the update-in-place ADS to include 50% reads and 50% writes (essentially every application-level write is translated into a read-modify-write sequence). In our study, we drive a write-only workload and half-read-half-write workload (representing the read-modify-write workload by existing ADSs) into an LSM store, LevelDB [7] from Google. Under certain experimental settings, the measured latency difference between the two workloads can be up to several orders of magnitude, for instance, 4.564 micro-second per operation for the write-only workload versus 604.302 for the half-read-half-write workload. More detailed and extensive studies are presented in § VI-B.

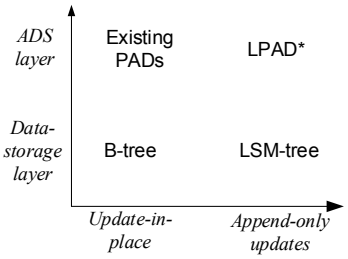


Fig. 1: Comparing ADS update protocols "*": note we only consider PAD or ADS protocols supporting random reads. Thus, hash-chain which does not support random reads is excluded, although a hash-chain is log-structured.

B. Global Consistency Verification w.o. Client Comm.

Consistency in a cloud storage is about whether storage writes can be serialized and whether reads return the latest writes on the serialized order. Existing researches on verifying strong multi-client consistency (strong in the sense of stronger than fork consistency) all rely on client communication to synchronize their views and to establish a consistency ground truth [43], [66], [50]. This paradigm requires all the clients to be available at scheduled time, which may render it unfeasible to many application scenarios where clients are asynchronous in nature and can not be coordinated to be available at the same time. The following is one example:

Cross-organizational Git repository: Git repository allows multiple developers to concurrently make changes to a shared project. Known consistency-oriented attacks (e.g. duplicated-effort attack [82]) on the git repository can be detected at the time of merging branches, by enforcing fork consistency [82], [47]. With strong consistency verification, it can *prevent* the attack in the first place (at the pull time). Existing strong-consistency solutions rely on client-view synchronization, which is feasible only to the case of a small repository with all the developers in one organization [43]. However, in a public repository (e.g. hosted in Github.com) where developers are organized in an ad-hoc fashion and from different organizations, it becomes unfeasible to schedule times for view synchronization. This becomes especially difficult as the repository grows large, which is the target application for our system.

There are many other use cases such as serving web app to ad-hoc mobile social users [41]. In general, our proposed systems achieve two features desirable to these new applications: 1) write-intensive workloads (e.g. social users constantly post new updates, and developers constantly push new commits), 2) strong consistency verification (e.g. needed to prevent duplicated effort attack instead of just detecting them).

The remainder of this paper is organized in the following way: We present a formal model of an LSM tree (§ III), formally describe the LPAD protocol and its construction (§ IV), and then present the LPAD system in an outsourced storage scenario (§ V). In each of these design layers, we describe the system frontend and backend. The paper organization for describing the LPAD technique is illustrated in Table I.

TABLE I: Paper organization describing the LPAD technique

Design layer	Frontend	Backend
LSM storage modeling	§ III	
LPAD protocol	§ IV-A	
LPAD construction	§ IV-B1	§ IV-B2
LPAD system	§ V-A	§ V-B

III. Modeling LSM Trees

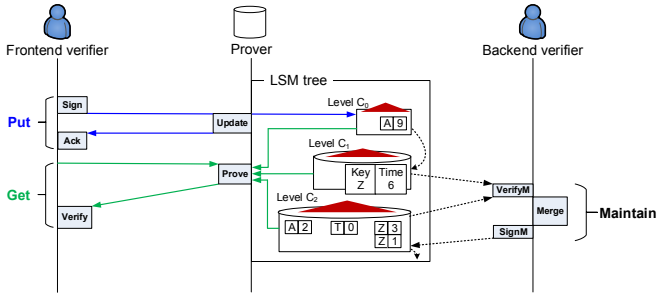


Fig. 2: LPAD construction with a three-level LSM tree

An LSM tree exposes two interfaces: A frontend interface for serving online reads/writes, and a backend interface for serving maintenance. An LSM tree supports key-value data model where each record consists of a key, a value and a timestamp; records are accessed by keys. The internal of an LSM tree is a series of key-ordered lists: All lists have their records sorted by the keys. To an online write or Put, only the very first list is updated-in-place, while all the other lists are immutable to the Put and are asynchronously updated in batch through a maintenance process, called compaction. Given multiple ordered lists, a compaction "merges" them into a single list; this improves future operation efficiency.

To an online read or Get, the LSM tree needs to check all the lists in the worst case, (although reading individual lists can be facilitated by binary search or primary index). For instance, in Figure 2, a Get needs to check on all three lists.

By this arrangement, an LSM tree has three performance characteristics: 1) optimized write performance because online writes do not cause random disk IO (the first list updated in-place resides in memory), 2) de-optimized read performance as a read checks multiple lists and causes multiple random disk

IO, 3) maintenance improves the performance of future writes as a compaction job merges a lower-numbered list (assuming the LSM tree's lists are numbered by their arrival order as will soon be described) into a higher-numbered list, clearing the way for future compaction.

Formally, an LSM tree is specified by the following invariant:

Invariant 3.1 (Intra-level key-ordering): An LSM tree lays out its data storage in multiple lists (i.e., so-called levels[3]), $C_0, C_1, \ldots, C_n$. In each list, records are sorted by key.

For instance, Figure 2 illustrates an LSM tree of three levels: C_0, C_1 and C_2, and at any level, say C_2, records are sorted by key, say from A to T to Z.

A compaction is formulated in the following format:

Invariant 3.2 (Two-level compaction): A compaction takes as inputs two key-complete lists[4] in two consecutive levels, say C_i, C_{i+1}, and produces as output a merged list C'_{i+1} replacing the higher-numbered input list C_{i+1}, and an empty list replacing C_i.

For an LSM tree with Invariant 3.1 and 3.2, we have the property that records of the same key across different levels are sorted by timestamp. For instance, in Figure 2, an older record of key A, with smaller timestamp 2, resides on higher-numbered level C_2, while a more recent record with timestamp 9 resides on a lower-numbered level C_0. Formally,

Theorem 3.3 (Inter-level time-ordering): In an LSM tree under compaction (specified by Invariant 3.2), given any two records of the same key, the one in the lower-numbered list, say $\langle k, ts\rangle$ in list C_i, must be younger than the one in the higher-numbered list, say $\langle k, ts'\rangle$ in list C_j. That is, given $\langle k, ts\rangle @ C_i$ and $\langle k, ts'\rangle @ C_j$, $i > j \Rightarrow ts < ts'$.[5]

Proof (Sketch) We present the proof sketch and leave the full proof in Appendix A. Theorem 3.3 is implied by Invariant 3.2. The state of the overall system can only be mutated by two operations: 1) a Put that only mutates the first list, 2) a compaction that, due to Invariant 3.2, only moves records from a lower-numbered list to a higher-numbered list. Consider an initial system state satisfying Theorem 3.3. The two state-mutating operations do not violate the theorem in the end state: Operation 1) inserts the latest record to C_0 so that previous older records still reside on levels higher than level 0. Operation 2) moves a key-complete range of records from a lower-numbered level to a higher-numbered one. Such a range does not create any non-consecutive range of records in one of these levels.

Now, we specify record freshness in an LSM tree. A record is fresh w.r.t. its key if and only if its timestamp is the largest among all records of the same key. Formally,

Definition 3.4: A record, $\langle k, v, ts_w\rangle$, is key-fresh w.r.t. timestamp ts_r in a level C_i, if and only if there is no record $\langle k, v', ts'_w\rangle \in C_i \wedge ts_r \leq ts'_w < ts_w$.

[3]In this paper, we use lists and levels interchangeably.

[4]A key-complete list in a level cover either all or none of the versions of a given key.

[5]Here, it assumes timestamps increase along with time, and a younger record has a larger timestamp.

Definition 3.5: A record, $\langle k, v, ts_w\rangle$, is key-fresh w.r.t. timestamp ts_r in an LSM tree C, if and only if there is no record $\langle k, v', ts'_w\rangle \in C \cup ts_r \leq ts'_w < ts_w$.

Theorem 3.6: A record at level C_i, say $\langle k, v, ts_w\rangle @ C_i$, is key-fresh w.r.t. timestamp ts_r in an n-level LSM tree $C = C_0 \cup C_1 \ldots C_{n-1}$, if and only if $\langle k, v, ts_w\rangle @ C_i$ is fresh in a level C_j, $\forall j \in [0, n)$.

Proof We prove if $\langle k, v, ts_w\rangle @ C_i$ is fresh in a level C_j, then it is fresh in C. We prove by contradiction. Assume if $\langle k, v, ts_w\rangle @ C_i$ is fresh $\forall C_i$, $\langle k, v, ts_w\rangle$ is not fresh. By definition, there must exist $\langle k, v', ts'_w\rangle \in C \wedge ts_r \leq ts'_w < ts_w$. Because $C = C_0 \cup C_1 \ldots C_{n-1}$, there must exist C_j s.t. $\langle k, v', ts'_w\rangle \in C_j$. By definition, $\langle k, v, ts_w\rangle$ is not fresh in C_j. Thus contradiction.

It can be similarly proved that if $\langle k, v, ts_w\rangle$ is fresh in C, it is fresh in $\forall C_i \in C$.

IV. LPAD Protocol & Construction

A. LPAD *Protocol*

Based on the LSM tree model, we describe our LPAD protocol. In the universe of an LPAD, there are three parties: a prover p, a frontend verifier v, and a backend verifier v'. The prover is untrusted (playing the role of untrusted LSM-based storage) while both verifiers are trusted. By convention, all parties are assumed to be reliable under system failures (or assuming no failures). Note our setting is slightly different from the standard ADS setting in that it considers an extra verifier v' for modeling the backend procedure of an LSM tree.

In this setting, an LPAD protocol formally describes the interactions between the prover and both verifiers, including the frontend interface (between p and v), and backend interface (between p and v').

On the frontend for writes, the verifier submits request vPut (k, v) and receives from the verifier a timestamp ts_w associated with the record written. vPut is a verifiable variant of regular Put request. As formally described in Figure 3, it produces an attestation that helps keep the record of this vPut operation. Timestamp ts_w dictates the position of the record in the global operation history and is useful to specify freshness. Note in this section, we only consider serial execution of Put/Get without concurrency, that is, the frontend verifier does not submit another operation until the current operation completes its execution.

On the frontend for reads, a verifier submits a read request, vGet(k, ts_r) with properly assigned timestamp ts_r and receives the result $\langle v, ts_{rw}\rangle$ from the prover. The correctness properties she wants to verify are two: 1) result integrity, which requires that record $\langle k, v, ts_{rw}\rangle$ is indeed a record written by a legitimate vPut before, 2) result freshness, which requires that record $\langle k, v, ts_{rw}\rangle$ is the latest among all matching records of key k and with timestamp before ts_r. Integrity can be verified easily by attaching each record a digest (e.g. a hash) and thus this work focuses on the freshness verification. As will be discussed in systems building (§ V), freshness verification is crucial to consistency verification (e.g.

staleness-based consistency [31], [22], linearizability [39]) and can be naturally extended to verify key-completeness [46] of range search in a multi-key setting.

On the backend, a maintenance job merges two lists C_i, C_{i+1} into one, C'_{i+1}. In the verifiable maintenance, the prover p and verifier v' interactively run vMerge that takes as p's input two ordered lists C_i, C_{i+1} and as v''s input the digests of the two lists δ_i, δ_{i+1}. At the end, the output state is that prover and backend verifier have the merged result, $\mathsf{p}.C'_{i+1}$ and $\mathsf{v'}.\delta'_{i+1}$.

The detailed LPAD protocol is formally presented in Figure 3.

In the following, we describe a construction of the LPAD protocol using Merkle trees and TEE.

B. LPAD *Construction by Merkle Tree & TEE*

An LSM tree, held by the prover, is digested by a *forest of Merkle hash trees (MHTs)*, each digesting one list in the LSM tree. Figure 2 illustrates a three-level LSM tree digested by a forest of three per-level MHTs (in red triangles). Given an LSM tree of dataset $C = \{C_0, C_1 \dots\}$, the LPAD construction converts it into a digested dataset $\widetilde{C} = \{\widetilde{C_0}, \widetilde{C_1}, \dots\}$ (where $\widetilde{C_i}$ is a Merkle tree) and a digest consisting of Merkle hashes at different levels, $\delta = \{\delta_0, \delta_1, \dots\}$ (where δ_i is the root hash of Merkle tree $\widetilde{C_i}$). Digested dataset $\widetilde{C}$ is stored by the prover and digest δ is stored and shared by both verifiers.

In the following, we describe the construction of frontend and backend procedures as formulated in Figure 3.

1) Frontend Construction: On the write path, v.Sign runs on standard public-key encryption algorithms, and p.Update verifies the record $\langle k, v\rangle$ using signature s and inserts it to $\widetilde{C}$. It uses C_0's Merkle proof on key k to instantiate $\mathcal{P}_U(C_0, k)$. Then v.Ack verifies the Merkle proof $\mathcal{P}_U(C_0, k)$, and updates digest δ by the hash of ts_w concatenated with $\langle k, v\rangle$.

On the read path, the major design issue is about the construction of freshness proof $\mathcal{P}[F]$. Here, we present two types of proofs with the same security, yet with different performance traits.

Definition 4.1 (All-level proof): Given $\mathsf{vGet}(k, ts_r) \to \langle v, ts_{rw}\rangle$, an all-level proof for freshness, $\mathcal{P}_1[F]$, consists of Merkle proofs for key k at all levels in an LSM tree.

Definition 4.2 (Selected-level proof): Given $\mathsf{vGet}(k, ts_r) \to \langle v, ts_{rw}\rangle @ C_i$ (that is, the result resides in level C_i), a selected-level proof for freshness, $\mathcal{P}_2[F]$, consists of Merkle proofs for key k at levels $C_0, C_1, \dots, C_i$ in an LSM tree.

For instance, in Figure 2, $\mathsf{vGet}(Z, 10) \to \langle v, 6\rangle$ and the result resides on level C_1. $\mathcal{P}_1[F]$ consists of Merkle proofs on all three levels, while $\mathcal{P}_2[F]$ consists of those on levels C_0 and C_1, excluding C_2.

A selected-level proof results in smaller-sized proofs yet an all-level proof enables parallel processing. In many real LSM storage systems (e.g. LevelDB), the selected-level proof matches their natural way of processing a Get request, that is, it checks levels from $C_0, C_1 \dots$ in order, *until* it reaches the level of a record matching key k.

- $\mathsf{v.Init}(1^\lambda) \to sk, pk$: Init, run by v, takes as input security parameter 1^λ, and outputs a secret key sk and a public key pk. The public key is implicitly used in all the algorithms below.
- $\mathsf{v.Setup}(C, sk) \to \delta, \widetilde{C}$: Setup, run by v, takes as input dataset C and the secret key sk, and outputs a digest δ and authenticated dataset $\widetilde{C}$.
- vPut $(k, v) \to ts_w, \mathcal{ATT}(ts_w)$: vPut submitted by verifier v takes as input a record $\langle k, v\rangle$ and produces a timestamp ts_w and attestation $\mathcal{ATT}(ts_w)$. The attestation enables the logging of this vPut operation at timestamp ts_w.
  - $\mathsf{v.Sign}_{sk}(\langle k, v\rangle) \to s$: Verifier v, using its secret key sk, Sign the record $\langle k, v\rangle$ and produces signature s as output.
  - $\mathsf{p.Update}(s, \langle k, v\rangle, \widetilde{C}) \to \{0,1\}, \widetilde{C'}, ts_w, \mathcal{P}_U(C_0, k)$: Prover p takes as input record $\langle k, v\rangle$, signature s, and its authenticated dataset $\widetilde{C}$, and produces a binary indicating if the update is executed successfully, a timestamp ts_w associated with the record, updated authenticated dataset $\widetilde{C'}$, and a proof about the pre-Update state of C_0 on key k, namely $\mathcal{P}_U(C_0, k)$. Note only level C_0 is mutable.
  - $\mathsf{v.Ack}(ts_w, \langle k, v\rangle, \mathcal{P}_U(C_0, k), \delta) \to \delta', ts_w$: The verifier associates the received timestamp ts_w with $\langle k, v\rangle$, and updates the digest δ based on $\mathcal{P}_U(C_0, k)$.
- $\mathsf{vGet}(k, ts_r) \to \langle v, ts_{rw}\rangle, \mathcal{C}[F](ts_r, ts_{rw})$: vGet, submitted by verifier v, takes as input queried key k and read timestamp ts_r, and produces as output result record with value v and its own timestamp ts_{rw}, as well as a certificate for freshness $\mathcal{C}[F](k, v, ts_r, ts_{rw})$.
  - $\mathsf{p.Prove}\ (\widetilde{C}, \langle k, v, ts_{rw}\rangle) \to \mathcal{P}[F]$: p.Prove takes as input authenticated dataset $\widetilde{C}$ and result record $\langle k, v, ts_{rw}$ and produces as output the freshness proof $\mathcal{P}[F]$.
  - $\mathsf{v.Verify}\ \ (\delta, \langle k, v, ts_{rw}\rangle, \mathcal{P}[F](ts_r, ts_{rw})) \to \{0,1\}$: v.Verify run by verifier v takes as input verifier's digest δ, result record $\langle k, v, ts_{rw}\rangle$, read timestamp ts_r and freshness proof $\mathcal{P}[F]$ and produces a binary indicating whether the verification passes.
- vMerge $(\mathsf{p}.C_i, \mathsf{p}.C_j, \mathsf{v'}.\delta_i, \mathsf{v'}.\delta_j) \to \{0,1\}, \mathsf{p}.\widetilde{C'_i}, \mathsf{p}.\widetilde{C'_j}, \mathsf{v'}.\delta'_i, \mathsf{v'}.\delta'_j$ (assuming $j > i$): vMerge takes as input the prover's two lists C_i, C_j and backend verifier's digests δ_i, δ_j, and produces as output a binary indicating if the vMerge is successfully executed and the final state $\mathsf{p}.C'_i, \mathsf{p}.C'_j, v'.\mathsf{v'}.\delta'_i, \mathsf{v'}.\delta'_j$. If the binary is 1, then $\mathsf{p}.C'_i = \emptyset, \mathsf{v'}.\delta'_i = \emptyset$ and $\mathsf{p}.C'_j$ is the merged list from the two input lists, and $\mathsf{v'}.\delta'_j$ is the digest of the merged list. This internally runs as an interactive process between prover and backend verifier:
  - $\mathsf{v'.CheckSel}(C_i, C_j) \to \{0,1\}$: CheckSel takes as input the two lists C_i, C_j and produces a binary indicating if the selected two lists conform to Invariant 3.2.
  - $\mathsf{v'.VerkfyM}(C_l, \delta_l) \to \{0,1\}, (l \in \{i, j\})$: VerifyM takes as input one of the two lists C_l ($l = i$ or $l = j$) and its digest δ_l, and produces a binary indicating if the content of the list matches the digest.
  - $\mathsf{v'.MergeSign}_{sk}(C_i, C_j) \to C'_j, \delta'_j, s'$: v'.MergeSign takes as input the two lists C_i, C_j and produces merged list C'_j, its digest δ'_j, and a signature s'.
  - $\mathsf{p.UpdateM}(C'_i, C'_j, s') \to \{0,1\}, \widetilde{C'_i}, \widetilde{C'_j}$: p.UpdateM takes as input two lists C'_j and C'_j, and a signature s'. It produces as output a binary indicating if the execution is success, the digested version of the two lists, $\widetilde{C'_i}$ and $\widetilde{C'_j}$.

Fig. 3: LPAD protocol

Correctness: The correctness of LPAD is about wether the proof can be used to correctly verify a Get result is fresh, as defined in Definition 3.5. To proof $\mathcal{P}_1[F]$, the correctness is straightforward, as Merkle proofs at all levels are included and each Merkle proof in a key-ordered list can prove the non-membership and freshness of a record at that level (Theorem 3.6). To proof $\mathcal{P}_2[F]$, the correctness is similarly proved except that the excluded Merkle proofs in $C_{i+1}, \ldots, C_n$ can be implied by Theorem 3.3 – all levels higher-numbered than C_i cannot have records fresher/younger than the records in C_i.

Figure 2 illustrates the intuition: Record $\langle Z, 6\rangle$ from level C_1 is the freshest (with the largest timestamp 7) in the entire dataset, and this can be proved by three facts: *F1*) The record is the freshest in its resident level C_1, *F2*) there is no record of key Z in level C_0, and *F3*) the record is fresher than any record of Z in the higher levels C_2, i,e., $\langle Z, 3\rangle$ and $\langle Z, 1\rangle$.

Security & Unforgeability: We analyse the security of frontend construction by considering stale-read attacks from the untrusted prover. In the attack, prover presents a stale Get result and tries to forge a freshness proof. The attack succeeds if the forged freshness proof can pass v.Verify by the verifier. For instance, in an operation sequence, $\mathsf{vPut}(k, v_1) \rightarrow ts_w = 11, \mathsf{vPut}(k, v_2) \rightarrow 12, \mathsf{vGet}(k, 14) \rightarrow \langle v_1, 11\rangle$, the vGet result is stale. The attack aims at forging $\mathcal{P}[F](k, v_1, 14, 11)$ to pass the verification.

The unforgeability of the freshness proof is provided by the collision-resistance of hashes in the Merkle proofs. The unforgeability of per-level Merkle proofs can be naturally extended to that of an overall freshness proof. For instance, in the previous example in Figure 2, *F1* is unforgeable due to the Merkle proof on C_1 and the key ordering in C_1 implied by Invariant 3.1. *F2* is unforgeable because of similar reasons (Merkle proof on C_0). *F3* is unforgeable by Invariant 3.3. Generalizing this case gives us a formal proof presented in the technical report [14].

2) Backend Construction in TCB: We construct the verifiable merge with linear (and theoretically optimal) cost.

In the basic construction for vMerge (formulated in Figure 3), v'.CheckSel(C_i, C_j) checks if $j = i + 1$ or $j = i - 1$ in a way to check the satisfiability of Invariant 3.2. v'.VerifyM(C_l, δ_l) reconstructs the Merkle tree of C_l and its root hash, and compares it with δ_l to result in the binary output. p.UpdateM updates the prover's LSM-tree layout from $\widetilde{C_i}, \widetilde{C_j}$ to $\widetilde{C'_i} = \emptyset, \widetilde{C'_j}$, if the update is successful.

Partitioned LSM Tree: In real LSM stores, the data storage in a level is partitioned into *sublists* and a compaction occurs at the fine granularity of the sublists. A sublist is an arbitrary, consecutive range of records in a list.

In vMerge on sublists, the input selection is parameterized by a policy that dictates what input sublists are allowed. In addition to requiring that sublists must reside on two consecutive levels (Invariant 3.2), another requirement is key-completeness, described as below:

Invariant 4.3 (Key completeness): Given a compaction with two sublists, the key-range of the lower-numbered sublist must be fully covered by the range of the higher-numbered sublist. In other words, given levels C_i and C_{i+1}, the lower-numbered sublist on C_i must not overlap in key ranges any part of C_{i+1} that is not in the selected higher-numbered sublist.

For instance, it is illegal to merge sublist $\{\langle Z, 6\rangle\}$ with sublist $\{\langle A, 2\rangle\}$ in Figure 2, but legal to merge $\{\langle Z, 6\rangle\}$ with sublist $\{\langle Z, 1\rangle, \langle Z, 3\rangle\}$.

For compaction under Invariant 3.2 and 4.3, the property of inter-level time ordering (Invariant 3.3) still holds. Because for a single key, the data migration between levels is still from lower-numbered to higher-numbered. We formally prove it in the technical report [14].

Under partitioned LSM tree, the CheckSel is slightly modified to be CheckSel$_p(sublist_1@C_i, sublist_2@C_j)$, and the checking logic is based on Invariant 3.2 and Invariant 4.3.

Verifiable-compaction security: There are two possible attacks by the prover, that is, selecting wrong lists/sublists that are not supposed to be compacted, and providing list/sublist contents that are modified from the original records. Both attacks are mitigated because of the Invariant enforcement (v'.CheckSel) and the unforgeability of MHT (v'.VerifyM).

V. LPAD Systems

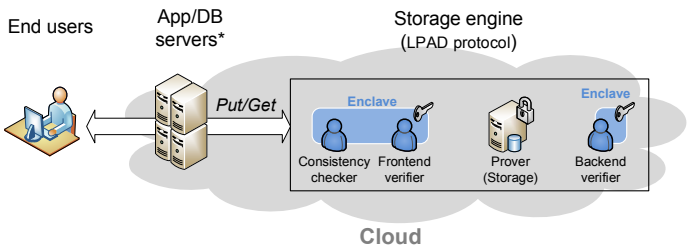


Fig. 4: LPAD systems in cloud data-outsourcing: means that application and database servers can be optionally added to the user-storage interaction.

This section describes an overall data-outsourcing system where the LPAD can be deployed.

We consider data-outsourcing to the public cloud. A data owner, for instance, a small-business company in a rapid growth (increasing customer base yet with limited computing budget), wants to outsource its customer data-storage to the cloud for cost effectiveness. The public cloud provisions machine instances through an infrastructure-as-a-service model (IaaS). The owner deploys the key-value store software on the instance to serve its data user. The deployed system architecture is illustrated in Figure 4. In particular, the data user can interact with the cloud store, either directly or through intermediate layers (e.g. application servers and database servers). What is exposed by the storage server is a trustworthy Put/Get interface:

$$\mathsf{vPut}_c(\text{key } k, \text{value } v) \rightarrow \text{Attestation } \mathcal{ATT}_{\text{put}} \quad (1)$$

$$\mathsf{vGet}_c(k) \rightarrow \langle v\rangle, \text{Certificate } \mathcal{CRT}_{\text{get}} \quad (2)$$

vPut_c/vGet_c is a variant of vPut/vGet that allows for concurrent invocation and offers certified consistency (by SGX authority described below). In this work, we consider the strong consistency level, Linearizability [39]. To a vPut_c call,

attestation $\mathcal{ATT}_{\text{put}}$ states that the store has committed the storage of the record and serialized it at a fixed position (that will not change later on). To a vGet_c call, certificate $\mathcal{CRT}_{\text{get}}$ states the result is correct in the sense of integrity and freshness (on the serialized total-order). Below, we introduce our trust model based on SGX.

Preliminary: SGX: We assume a cloud server is equipped with Intel SGX. Intel SGX is a security-oriented extension for x86-64 ISA 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 can not succeed. SGX's software TCB includes only user-space program and excludes any OS kernel code, by explicitly prohibiting system services (e.g. system calls) inside enclave.

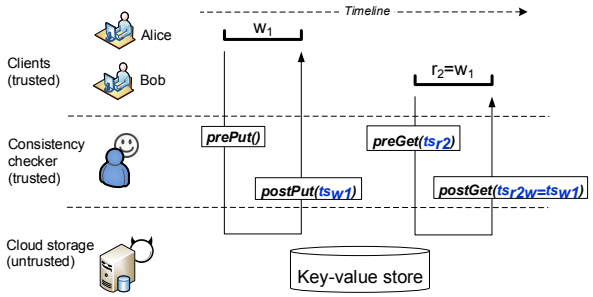
To use the technology, a client initializes an enclave by uploading the in-enclave program and uses SGX's seal and attestation mechanism [19] to verify the correct setup of the execution environment (e.g. by a digest of enclave memory content). During the program execution, the enclave is entered and exited proactively (by SGX instructions, e.g. `EENTER` and `EEXIT`) or passively (by interruptions or traps). These world-switch events trigger the context saving/loading in both hardware and software levels. Comparing prior TEE solutions [12], [13], [2], [10], SGX is unique in supporting multi-core concurrent execution and dynamic paging for runtime efficiency.

Trust model: We assume clients are trusted, and there is mutual trust among them (e.g. owner-and-user, and user-and-user). A client normally does not need to communicate with other clients, except for initial key-exchange during the setup. A client trusts nothing on a cloud machine except for the initialized enclave; the trusted computing base includes at the hardware level, the SGX CPU, and at the software level, the enclave program. The untrusted part on the server includes hardware, such as the non-secure memory pages outside enclave and all the peripherals, and software, such as operating systems and user-space processes running outside enclave. The untrusted host can mount a replay attack and answer to a Get operation with properly signed but stale version. We assume a secure channel is established over the untrusted Internet between the client and cloud server and is resilient to standard network attacks (e.g. man-in-the-middle attacks, etc. [42]).

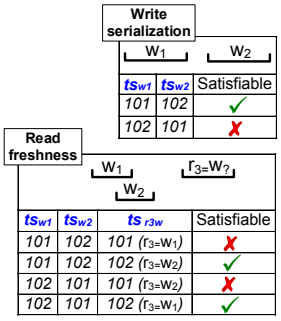
Scope: This work does not address out-of-scope issues which are addressed by orthogonal work, including denial-of-service attacks, proven deletion [34] under Sybil attacks [84], TEE state-continuity under replay or rollback attacks [74], enclave side-channel attacks [85], and design flaws of an enclave program.

A. Frontend Consistency Checking

Preliminary: Linearizability specification: We focus on the strong consistency level, linearizability. Conceptually, the



(a) Consistency checker thru. pre-/post- Put/Get hook: $r_2 = w_1$ means read r_2 returns the record written by w_1.



(b) Consistency specification

Fig. 5: Consistency checker: **Write serialization**: the total-order must be consistent with the real-time order of w_1 and w_2 (in the right figure), due to Linearizability requirement on real-time. For instance, write serialization does not allow the timestamp assignment of $102, 101$ to w_1, w_2 because this would place w_1 after w_2 which is different from the real-time order that w_1 executes before w_2. **Read freshness**: the read must return the latest write on the total-order dictated by ts_{w1} and ts_{w2}, due to Linearizability requirement on freshness. $r_3 = w_2$ means read r_3 returns the record written by w_2.

linearizability [39] is about the existence of a serialization total-order that conforms to two conditions: real-time requirement (denoted by $\mathbb{L}1$) and freshness (by $\mathbb{L}2$): $\mathbb{L}1$ requires that the total-order reflects the real-time partial-order of the reads/writes execution ("operation A completes before the beginning of operation B requires that A is placed before B on the total order"). Freshness $\mathbb{L}2$ requires that any read returns the latest write on the total-order. $\mathbb{L}2$ is specified in Definition 3.5.

Figure 5b illustrates the consistency specification in a (incomplete) list of different cases. In this work, a key assumption we make is that the untrusted store that claims to have strong consistency (linearizability) promises to provide a globally unique timestamp that represents the total-order, i.e., ts_w. In many real consistent-store systems, this assumed timestamp is already present. Note that this assumption considers that both violation of strong consistency and failure to present correct timestamp as malicious behavior that should be detected. Under this assumption, the consistency verification is reduced to checking whether conditions $\mathbb{L}1$ and $\mathbb{L}2$ are met on the total-order represented by the timestamp.

Following the precise specification above, we propose an

algorithm to *immediately* verify the linearizability. The algorithm considers concurrent operations that may complete out of order, that is, operation completion order is different from the assigned total-order by ts_w. This is illustrated by dark boxes in Figure 5b. Unlike the previous work based on periodical scheduling of verification [43], [66], the proposed algorithm achieves the theoretic lower bound in the delay between the verification time and completion time. Briefly, the technique proposed is to consider any completed operation be in one of two states, serialized and non-serialized,[6] and to verify linearizability only on the serialized operations.

The frontend system architecture is illustrated in Figure 4 where a consistency check is added in front of the frontend verifier. The consistency checker takes as input the concurrent execution of reads/writes and outputs a binary satisfiability decision upon the completion of each read (or write). Figure 5b illustrates the details of consistency checker which takes actions to respond to four events: pre-Get, pre-Put, post-Get, and post-Put. Internally, the checker transits an operation (Put or Get) among three states: pending (operation started and not yet completed), completed, and serialized (an operation is serialized when all operations with write timestamp smaller/older than the operation are completed).

A key design is the interaction between the consistency checker and frontend verifier. The consistency checker relies on the frontend verifier to present the freshness verification, and it does so *only when a completed operation transitions to the serialized state*. In our implementation, a synchronization point is confined within the scope of a single event; no locks are held across events. Consistency conditions (e.g. serialized writes and read freshness as illustrated in Figure 5b) are checked upon post-Put/Get events. The pre-Put/Get events initiate the bookkeeping of operation states.

The state maintained by consistency checker may need to ensure state-continuity upon system crash; and we assume a reliable monotonic counter (e.g. Memoir [63] and Ariadne [75]) exists in enclave.

Note our assumption that untrusted store returns a global timestamp can be realized in strongly consistent stores, such as LevelDB. A strongly consistent store takes a single writer[7] and persists the written record in a log before completing the write – The write timestamp ts_w is assigned in the logging process and is used to dictate the position of the written record in the log.[8] The pseudo-code of the frontend consistency verifier is in the technical report [14].

Frontend security: The untrusted prover interacts with the frontend verifier through vPut/vGet interface. The prover can forge a vGet result that is stale or incorrect. The verification by v would not pass as these results do not match the current digest held by the verifier.

[6]An operation occupying the serialized state means all the operations before this one in the assigned total-order have completed. An operation occupying the non-serialized state means there are some operations before this operation in the total-order that are not completed.

[7]Currently, our implementation is specific to this strongly consistent and single-writer stores. Extensions for concurrent writers in key-value stores are addressed in related work, such as cLSM [35].

[8]We don't consider the semantics of transactional isolation in this work.

The untrusted store interacting with the end users (or upper-layer systems) might mount consistency/concurrency attacks, of which the replay attack (presented earlier in § III) can be treated as a special case. In the concurrency attack, the store equivocate on concurrent Put/Get calls. For instance, Figure 5b illustrates a replay attack on concurrent writes, that is, read r_3 returns the data written by w_2 whose execution overlaps w_1. This is a violation of freshness, because the untrusted store has claimed w_1 happens before w_2 ($ts_{w1} < ts_{w2}$), even though they are concurrent in real execution. The replay attack is mitigated because of the LPAD's freshness and the consistency checker. Similarly, the untrusted store can mount realtime-order attack where the claimed write timestamps of two writes are inconsistent with their real-time order. For instance, in Figure 5b, w_1 occurs before w_2 in real-time, yet their assigned timestamps are in the reversed order, $ts_{w1} > ts_{w2}$. The realtime-order attack is mitigated by the bookkeeping of the consistency checker (i.e. the checker maintains the real-time ordering in pending and completed operations).

Note global linearizability is stronger than fork consistency [49], [47], thus our security layer mitigates forking attacks.

B. Backend Maintenance

```
1  void vCompact(C_i,C_j){//assume i<j
2    if(!CheckSel(C_i,C_j)) abort();
3    do{
4      if(record_i < record_j){
5           output = record_i;
6           rebuilt_digest_Ci.update(output);
7           digest_mergedCj.update(output);
8           eof = inputKV(C_i.next(), record_i);
9      }else{
10          output = record_j;
11          rebuilt_digest_Cj.update(output);
12          digest_mergedCj.update(output);
13          eof = inputKV(C_j.next(), record_j);
14     }
15     if (filterout_policy(output) == FALSE)
16         outputKV(output);
17   } while (!eof);
18   //output pending list
19   if(VerifyM(rebuilt_digest_Ci, delta_i)
20     && VerifyM(rebuilt_digest_Cj, delta_j)){
21     s = Sign(mergedCj);
22     return s;
23   } else abort();
24 }
25 }
```

Listing 1: One-pass program for verifiable compaction in Enclave

One naive approach to implementing the LPAD's verifiable compaction is to run the related constructs separately, that is, implementing v'.CheckSel, v'.VerifyM, and v'.MergeSign as separate iterations over the merging lists.

A more efficient implementation is a one-pass algorithm that iterates through all the lists without any repeated access. The algorithm is illustrated in List 1 which embeds all the related constructs in a single merge-sort-style loop. In each iteration of the loop, it might trigger two cross-boundary calls to switch the execution out of enclave: `inputKV` which reads one record into the enclave from the untrusted world (memory or disk), and `outputKV` which stores the output data to the untrusted world. By the end of the loop, the enclave endorses

(by signing) the merged list only when both v'.CheckSel and v'.VerifyM are satisfied. Figure 6 illustrates the system workflow of compacting two sublists or files[9] in two levels with an enclave. In particular, we address three systems-level issues below:

Memory-efficient MHT construction: Recall the input verification works by reconstructing the MHT from `inputKV()`, and checking its root hash (upon completion) against the previous digest. Our system limits the Merkle root hash construction by consuming $\log n$ memory footprint (n is the number of MHT leaf nodes or the number of records in a file), because it only needs to maintain a tree "frontier" (i.e. the path from the current leaf node to the root) bounded by the tree height [79]. The output stream is digested by similarly constructing an MHT upon `outputKV()` calls.

Versioning policies: Applications may have different policies in managing versions. For instance, an application may explicitly require the "update" semantic for its write, which states the write should overwrite all previous versions and requires the system to maintain a single, latest version for the written key. Other applications may prefer treat the update as an insert, allowing for multi-versioned data. A common policy is to keep the latest k (e.g. $k = 3$) versions and delete any version older than them.

Policies can be implemented as a filter plugin on the one-pass `vCompact` program in enclave. For instance, retaining the latest k versions is implemented by maintaining a single per-key counter counting the number of versions of the current key visited. One thing noteworthy about the one-pass compaction is that the data records are emitted in the key order (tie broken by timestamps) so that different versions of the same key are visited together and a newer version is always visited before an older version. This order allows the versioning policies to be implemented as an add-on filter on the pass.

Handling delete: Similar to original LSM stores (e.g. LevelDB [7]), we treat a delete request as a special data record, a.k.a, tombstone write. The semantic of the delete record w.r.t. key k is to delete all the versions of k preceding (in arrival time or timestamp) the delete record. We implement this semantic in `vCompact` by the deleting policy. The deleting policy is very similar to the overwriting policy with the only exception that the delete record itself will be dropped if the compaction reaches the last/highest-numbered level (that is when it can assure all possible data records before it are deleted).

Implementation with SGX: By the SGX hardware design, system services are prohibited inside enclave and have to be executed outside. The compaction interacts with input/output data resident on disk. It is thus essential to coordinate the scheduling of in-enclave merge computation and outside-enclave disk-accesses.

From programming perspective, the problem boils down to "partitioning" the code path of interface functions, `inputKV()/outputKV()`, to the parts that are run in and outside enclave. A naive partitioning is directly on the interface function, which results in a world-switch (i.e. the switch of program execution between the enclave and untrusted host) upon every individual call to `inputKV()`/`outputKV()`. That is, a world switch is triggered to read and write each key-value record. This design results in significant runtime overhead in our experiments. A better design is to result in less frequent world-switches. We partition the code at a level close to the system-calls so that a world-switch is triggered only when it becomes necessary. Specifically, we maintain a buffer in the untrusted world to hold input/output data, and when the buffer runs out of space, our partitioned code starts to switch out enclave and performs system calls to read/write files in the untrusted world.

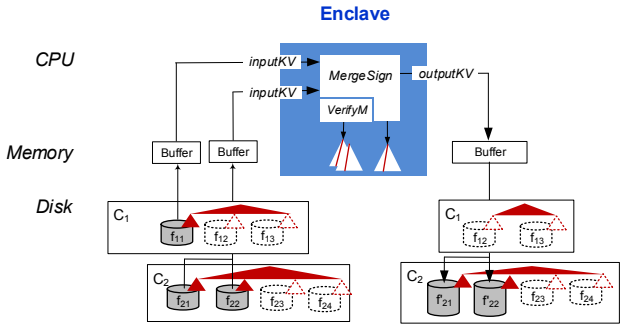


Fig. 6: In-enclave compaction: three selected files or sublists, f_{11}, f_{12}, f_{21}, are merged into two or more files in C_2, e.g., f'_{21}, f'_{22}. It shows the workflow of in-enclave compaction; the white triangles inside the enclave refer to the frontier built to construct input/output MHT root hash.

In our implementation, we maintain the Merkle root hashes of all the per-level MHTs in enclave to simulate the "signing". Because in-enclave signing by digital signatures would unnecessarily increase the enclave codebase. Normally, an LSM tree does not have too many levels (e.g. fewer than 20), making it feasible for storing Merkle root hashes.

1) Security Analysis: A local attacker is a party who fully controls the server's software/hardware stacks except that the attacker can not physically break into the enclave world: The SGX CPU is tamper resistant and the enclave memory pages are encrypted and protected under a computationally bounded attacker.

By design, an enclave execution allows two possible channels for boundary-crossing, 1) switching control out of enclave (through SGX instruction, EEXIT [11]), and 2) direct untrusted-memory access from enclave. To a local attacker, this constitutes the attack surface of an enclave program execution.[10]

Through the attack surface, we consider two attacks by 1) injecting incorrect input data (data exploit), and 2) exploiting in-enclave program integrity for control-flow hijacking attacks. The first attack has been analyzed in § IV. To attack 2), we assume a trustworthy enclave program with program integrity.

[9]In a partitioned LSM tree, a sublist is materialized as a file. Hence, we use file and sublist interchangeably hereafter.

[10]We don't consider the denial of service attacks mounted by malicious OS or the side-channel attacks [85] targeting on confidentiality of an enclave.

In our implementation, we leverage Control-Pointer Integrity (CPI [44]) in the recent LLVM compiler that ensures (in a certain degree) the program integrity.

VI. Evaluation

In this section, we evaluate our design to answer the following questions:

- What is the TCB size? (§ VI-A)
- What is overall performance benefit of LPAD comparing existing ADSs (§ VI-B)?
- What is the detailed performance overhead of LPAD on the frontend (§ VI-D) and backend (§ VI-C)?

A. Implementation & Enclave Size

We implemented our LPAD systems design and built the system of LPAD, a trustworthy key-value store based on LevelDB, Intel SGX SDK and Crypto++ SHA code. The system implementation involves writing programs for the execution in two worlds.

The program in the untrusted world is taken out of the LevelDB codebase [7] with several changes: 1. hooking our enclave program (described below) to the LevelDB operations, 2. implementation for storing and serving MHT digests; we here reuse several LevelDB persistence utilities and format the MHT digests in a key-value format for that purpose. We also implemented the proof construction (p.Prove) for processing Get in the untrusted world. We modified the LevelDB codebase to return the write timestamp upon Put. This change is not significant (e.g. leaving the original codebase as it is) and does not cause high overhead.

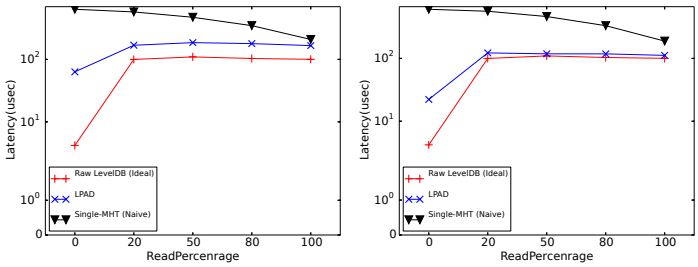
The enclave program consists of four modules, 1) merge, which performs the compaction computation, 2) MHT operations, which include MHT construction and Merkle proof verification, 3) SHA which is taken from Crypto++ library with the modification to get rid of system calls for the use in enclave, 4) consistency checker, and 5) misc. functionality which includes the world-switch glue code generated by Intel SGX SDK (alpha on Linux), various condition checking, thread synchronization support, etc. The four modules enable enclave-entry points for all the constructs from verifiers (v or v') in Figure 3. The enclave program is written in C.

We report the size (by lines of code, LoC) of each enclave module in Table II. The total enclave code line is 1025. We compare it with the Haven [25] approach which would put into the enclave the entire codebase of LevelDB, among other facilities (hence TCB size is estimated to be larger than LevelDB's number of codelines, 19567 LoC). By comparison, our design results in a TCB size reduction by at least 20 times.

Layered implementation While we did modify the codebase of LevelDB to hook LPAD, it does not have to be the case. Depending on the API exposed by the storage system, our implementation might be incremental, that is, without changing the original storage codebase. For instance, in HBase, it already exposes hooks for pre-Put/Get/Compact, post-Put/Get/Compact, as in its CoProcessor API [5].

TABLE II: TCB size

	TrustKV					All in enclave [25]
LoC	1025					> 19567
Module	merge	SHA	MHT	checker	misc.	LevelDB
LoC	118	339	40	134	394	19567



(a) SHA-3 (b) SHA-1

Fig. 8: Memory intensive workloads

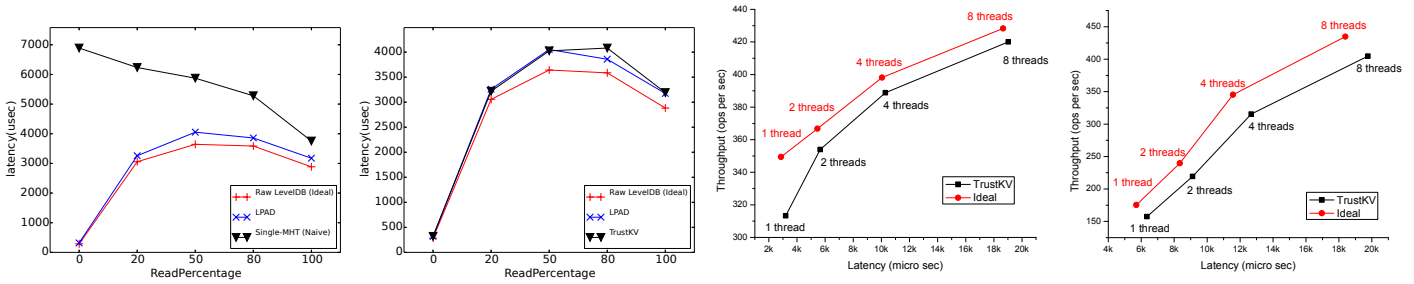
B. Overall Performance

1) LPAD *vs ADS:* This set of experiment aims at studying the performance advantage (or disadvantage) of an LPAD over the existing ADS when running on an LSM storage. We choose a single MHT over the entire dataset to represent the update-in-place ADS. Note the single-Merkle tree approach is used in many verifiable storage systems including SUNDR [47], and is representative.

The experimental setup is on a laptop 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 on. We generated a base dataset with uniformly distributed keys; our base dataset includes 200 million records (22GB without compression) with uniformly distributed keys;the keys are generated sequentially in order. The overall number of read/write requests are less than 1% of the entire dataset. For a key-value record, the key size is 16 bytes and value size is 100 bytes.

We performed the experiment by running LevelDB's built-in benchmark with the modification to drive workloads of different read-write ratios to the target system. We varied the read percentage from 0% (i.e. write-only workload), $20\%, 50\%, 80\%$ to 100%. We tested different storage system settings, such as compaction turned on/off, different record sizes, use of different hash algorithms (e.g. SHA1 or SHA3). We run each experiment at least three times, and report the results in two metrics. As LPAD is designed for single-threaded case, the experiments are conducted under single-threaded workloads.

The performance result is presented in Figure 7a. The performance difference between LPAD and the single MHT can be up to three orders of magnitude, and it is clear that LPAD's curve is very similar to the "Ideal" performance where the raw LevelDB is tested. In both LPAD and raw LevelDB, the latency increases as the workload moves from write-only to more read-intensive, both reflecting the write-optimized performance nature of LSM storage design. By contrast, in the single-MHT, the write-only workloads result in the largest latency. The performance result can be explained



(a) LPAD vs ideal (single-threaded) (b) LPAD vs TRUSTKV (single-threaded) (c) Concurrent readers (d) Concurrent single-writer-multi-readers (number of threads in the figures is that of readers)

Fig. 7: Disk IO intensive workloads

by the following: In our setting, most of the read requests are cold and served from disks, rendering the disk seek the dominant factor. The LPAD design introduces no extra disk seeks, while the naive Single-MHT design incurs a lot of extra disk seeks due to its update-in-place nature.

We perform similar experiments with a much smaller dataset (with all records fit in memory). In this memory-intensive workload, as disk IO is stripped away from the critical path, the overhead of LPAD becomes more significant: as in Figure 8, the LPAD overhead is up to 50% of the raw LevelDB when SHA3 is used to construct the Merkle trees. When SHA1 is used, the overhead reduces significantly, despite its lower security.

2) Cost of Consistency Checker: This experiment aims at studying how much performance overhead is caused by our front-end consistency checker. We add the consistency checker to the LPAD and call the overall system by TRUSTKV. We first conduct experiments in a single-thread setting. The number of queries for each workload is one million. We report the execution latency in Figure 7b.

Both the LPAD and TRUSTKV system preserves the performance of ideal LSM storage, with small performance overhead (less than 12 percent). Specifically, the performance trend is a hill-shaped curve; the latency reaches the highest point for the read-write workload mixed at certain rate. This is consistent with LSM storage's performance as it is not as good serving read-write workloads as serving write-only (or read-only) workloads.

Multi-Threaded Execution: This experiment studies the performance of TRUSTKV under multi-threaded workloads. Note LPAD can only run for single-threaded execution, and is thus excluded here. We run LevelDB's built-in workloads (read-random and read-while-writing) on top of both variants. The number of queries for each workload is one million, which are evenly distributed among read threads. We report the latency and throughput in Figure 7c and Figure 7d. From both figures, the TRUSTKV reduces raw LevelDB's throughput by about 10%. As target throughput increases, the latency increases (almost linearly) with the throughput.

C. Compaction Performance

1) Micro-benchmark: In our implementation of in-enclave compaction, there are two options: 1) the level of code-partitioning that decides the frequency of context switches, as discussed in § V, and 2) whether the context switch copies the untrusted buffer to enclave; when the buffer is not copied, it is the pointer to the buffer that is passed into the enclave for direct access. We design the experiments to understand the performance impact of these implementation options. We consider 5 variants with different combinations of these options: copy, copy (no MHT), non-copy, non-copy (no MHT) and naive-partition. The first four variants perform one context-switch per syscall, while the last one, naive-partition, incurs the most context switches (one per read and write). non-copy does not make data copy upon context switch; and no MHT means MHT was disabled in experiments. We also consider the baseline (unsecured) , which measures the original LevelDB performance without any security.

In experiments, we change compaction configurations, including file size, the number of input files, buffer size, and record size. We measure the execution time; for each result, we conducted three runs of experiments and report average result. We use a buffer size to hold about 1000 records and measure the execution time under different settings: We use 5 input files and vary the file size from 4 million records to 12 million, and report the result in Figure 9a. We fix at 31.5 million records and evenly distribute them to varying number of files from 3 to 9. We report the result in Figure 9b. From these two figures, it can be seen: 1. the execution time grows linearly with the number of records, and is insensitive to number of files, 2. the memory copies do not cause significant overhead, 3. with a large buffer, the context switch overhead is negligible (except for naive-partition), 4. the hash computation (by SHA) incurs about $45\%-115\%$ more execution time, 5. the naive-partition approach results in the highest execution time, caused mainly by an excessive amount of context switches.

The impact of the context switches can be seen more clearly from Figure 9c where we vary the buffer size. With a large buffer size (e.g. $>$ 2 KB), the overhead of context switches is small. When the buffer size is smaller than 2 KB, the performance overhead is significantly increased by context switches.

In the last experiment, we vary the record size, more specifically, the size of value in a key-value record. We report

the experiment result in Figure 9d. The difference in execution time can be attributed to the disk IO costs: A larger record costs longer disk-memory transfer time and all the series converge at a large record size in Figure 9d.

2) LevelDB benchmark: We used the built-in benchmark of LevelDB to study the impact of longer compactions to other store operations. The built-in benchmark tests multiple workloads in a sequence: "fillseq, fillsync, fillrandom, readrandom, readrandom, compact, readrandom." In particular, "fillseq", "fillsync", and "fillrandom" are write-only workloads, and "readrandom" is a read-only workload; they both may trigger the execution of compaction. In addition, "compact" is a workload where compactions are explicitly triggered. "fillseq" and "fillsync" would clear the storage so that the next workload can start from a freshly new store. We consider 3 variants in this experiment, copy (no MHT), copy and naive-partition. Other settings are the same to our micro-benchmark. We report the result in Table III; in addition to the raw latency readings, we also report the normalized latencies by the baseline approach. We highlight the largest normalized latencies: They all belong to the "compact" workload. The slowdowns by copy(no MHT), copy and naive-partition are respectively $1.3, 3$ and 9.3, which are consistent with the micro-benchmark results, except for that the absolute performance difference is smaller due to the interference of front-end query operations.

TABLE III: Compaction latencies with online queries (the unit is micro seconds except for "compact" with seconds, and the number in the parenthesis is normalized latencies by baseline)

	baseline	copy(no MHT)	copy	naive-partition
fillseq	29.2	29(1.00)	30.9(1.06)	28.8(0.99)
fillsync	378951	39854(1.05)	38330(1.01)	38454(1.01)
fillrandom	71.3	85.0(1.19)	127(1.79)	257(3.61)
readrandom	9.55	11.6(1.22)	14.5(1.52)	17.8(1.86)
readrandom	5.02	5.73(1.14)	7.08(1.41)	10.3(2.05)
compact($\times 10^6$)	5.02	6.82(**1.36**)	15.43(**3.07**)	47.06(**9.37**)
readrandom	3.599	3.553(0.99)	3.611(1)	3.647(1.01)

D. Frontend Performance

In this section, we characterize the performance of front-end query verification. One factor affecting performance is the size of the proof. We consider the use of selected-level proof (in Definition 4.2) as default in LPAD, and compare the performance using it with that using the all-level proof (in Definition 4.1). We also evaluate the performance of front-end query processing without running any digest computations. It allows studying the performance impact of world-switches to online query performance. We use the unsecured LevelDB as the same baseline. We re-run the LevelDB built-in benchmark under the same configuration, and report the result in Table IV.

The trends are similar: All-Level always has the highest normalized latency in all workloads, and the last workload ("readrandom") has the largest normalized latency in all tested approaches. Compare approaches No MHT and LPAD: In No MHT whose overhead is only introduced by SGX has a less than 3X slowdown, while LPAD has a slowdown up to 15X. We suspect the use of MHT/SHA is the main culprit of performance slowdown.

The result also shows the effectiveness of using the inter-level time ordering in saving read overhead. Without the time ordering, the slowdown of All-Level is up to 51.44; it is more than 3 times larger than that of LPAD.

TABLE IV: Latencies of online queries (the unit is micro seconds except for "compact" with seconds, and the number in the parenthesis is normalized latencies by baseline)

	baseline	LPAD	All-Level	No MHT
fillseq	29.2	63.9(2.19)	64.1(2.20)	30.3(1.04)
fillsync	37895	34792(0.92)	38149(1.01)	35493(0.94)
fillrandom	71.3	130(1.83)	134(1.88)	129(1.82)
readrandom	9.55	88.2(9.24)	266(27.8)	22.7(2.38)
readrandom	5.02	77.1(15.3)	260.6(51.8)	11.9(2.37)
compact($\times 10^6$)	5.02	12.20(2.43)	12.51(2.49)	12.21(2.43)

We conclude our performance study of LPAD: Comparing update-in-place ADS, LPAD improves the performance by up to two orders of magnitude. LPAD becomes less disk-IO intensive. Among various LPAD configuration options, the use of MHT/SHA stands out to be the most significant in performance impact. The SGX/Enclave hardware, if properly used, can be lightweight with less than $3X$ slowdown for online query processing, and less than $1.4X$ slowdown for compaction jobs. MHT/SHA causes relatively high runtime overhead, but it comes with the benefit of higher security levels. In practice, clients who are more concerned about performance should use more efficient but less secure hash primitives (e.g. SHA-256). We leave it to the future work to study the performance optimization problem.

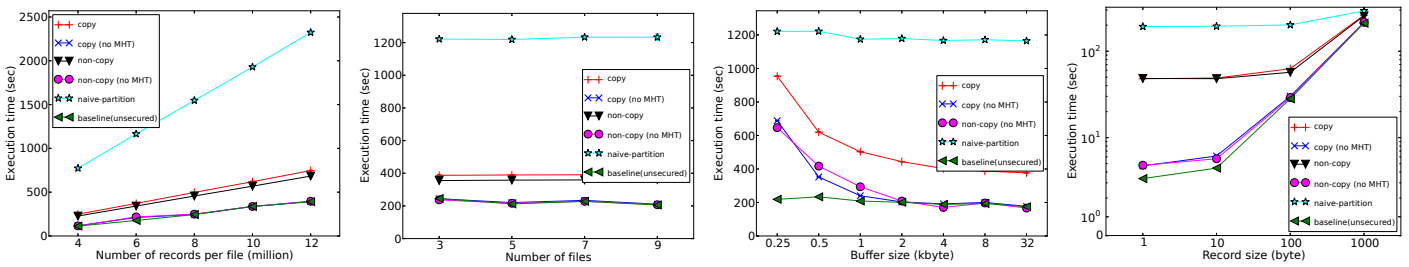
VII. Related work

A. Systems & Databases on TEE

Existing TEE solutions include Intel SGX, TXT [12]/TPM [13], ARM TrustZone [2], IBM SCPU [10], etc. Prior to SGX, there are TEE-based software systems for database systems [24], [23], [20], key-management [76], etc. Intel SGX adds architectural supports for more generic execution in the secure world; these new features include dynamical memory allocation, paging, and multi-core execution. There are recently software systems built on SGX, for big-data analytics [67], supporting legacy applications [25], network management [70], distributed multi-party computations [38]. A formal verification technique is proposed to strengthen the security of enclave programs [72].

In particular, Haven [25] supports generic legacy applications using SGX. The support is done by loading the entire application software stack into enclave, and redirecting system calls to the untrusted world. This design, while enabling software compatibility, increases the in-enclave codebase to a size (hundreds of thousands or millions of codelines) that it can not be formally verified in an efficient way.

VC3 [67] supports MapReduce style big-data analytics on SGX. It partitions the Hadoop software stack and places only the user-defined mapper/reducer functions inside enclave, thus resulting in a small and trusted codebase. However, this approach to partitioning is specific to the MapReduce framework and is not readily applicable to partitioning a key-value store system. Furthermore, VC3's in-enclave verification



(a) 5-way compaction and 8k buffer (b) 63 million records and 8k buffer (c) 5-way compaction and 10 million records (d) 5-way compaction and 600M data size

Fig. 9: LPAD execution time

works mainly for stateless batched computation, while our work complements the VC3's approach in supporting stateful computations with storage.

CorrectDB [23] is a trusted database system that enables correctness-verifiable SQL processing on IBM SCPU [10]. It partitions an SQL query plan to the two worlds; the query executed in the non-secure world is protected by using MHT, and the query executed inside the secure world involves multi-dimensional data where the MHT can not handle efficiently. Our TRUSTKV shares the similar goal with CorrectDB in placing code to the secure world only when it is necessary. However, the target applications and systems are different, and the techniques are orthogonal.

B. Storage Consistency Verification

Our system uses the notion of strong consistency to specify the correctness of storage queries. Given the limited space, we cannot completely survey the extensive body of consistency research; instead, we focus on cloud storage consistency and verification.

In the context of cloud storage, query linearizability [39] requires the total order among reads and writes on a single object (or a single key in a key-value store). Transactional serializability [26] requires the same but for reads and writes on different objects. Relaxed consistency levels [80], [64], [81] and eventual consistency [31], have been proposed for large-scale distributed storage in the cloud. In the eventual consistency, the read can return an arbitrarily stale write result as long as it "eventually" returns fresh data. In the presence of multiple clients, a forking attack is the one that the cloud can present different results to different clients, hence "forking" client views. Fork consistency [49] specifies that a client's view can only be forked once. Without client-to-client communications, the fork consistency is the strongest consistency level achievable in theory.

There is a body of research work on secure verification of storage consistency under various specifications [66], [73], [36], [43]. CloudProof [66] verifies the strong consistency in both write-write linearizability and read-after-write freshness. Caelus [43] verifies the weaker consistency models, such as time-bounded eventual consistency. To verify the consistency specification, the commonly adopted mechanism is by logging the operation history by the hash chain and periodically auditing it. While it is effective in amortizing the verification cost, it does not detect consistency violation in real-time. In addition, it also assumes a trusted, third-party auditor collecting the operation history from both the clients and the cloud. This requires the client availability for auditing which might not be realistic, particularly for ad-hoc mobile users. By contrast, our work enables real-time consistency verification, does not assume a client to be online other than the query time, and can securely verify query consistency with efficiency.

Alternatively, read-after-write freshness can be verified by MHTs [36], [73], [79] (with a trusted timestamp oracle). In these approaches, the owner stores the root hash digesting the remote MHT in the cloud [36], [73]. Upon data writes, the owner needs to update the MHT by reading back relevant authenticated information from the cloud before producing the new root hash. Our work avoids the inefficiency of reading digest upon writes, and inspired by the log-structured system we apply the idea of append-only writes to the updating-MHT problem.

Prior work based on trusted hardware [86], [45] addresses the freshness verification problem. They tackle reliability under faulty hardware and their approaches are complementary to our work.

Venus [71] supports verifiable causal and eventual (strong) consistency on untrusted storage. Venus assumes honest clients, and for the purpose of setting up the ground truth of consistency verification, a subset of clients that are always online. The put/get operations are concurrent and non-blocking, with online causal consistency; an asynchronous callback is needed to verify the (eventually) strong consistency through client-to-client communication.

C. Log-Structured Merge Storage Systems

Given the recently renewed interest in LSM storage systems, there is a body of researches on improving and applying the LSM structures. To improve read performance on log-structured stores, bLSM [68] systematically model the LSM tree performance, and organizes data into row-based storage for serving row-based queries with less disk access. The compaction process is decomposed at finer granularity and is run with costs carefully amortized to each write. Prior work [40] partitions the LSM tree storage by keys to accommodate the skewed key access popularity. The level sizes follow an exponentially growing sequence, in a way to minimize write amplification. In distributed key-value stores, compaction jobs among multiple nodes are scheduled and coordinated for better performance [16]. The LSM tree structure is applied

beyond disk-based storage: With a clear separation between mutable and immutable structures, an LSM tree minimizes storage overhead of dynamic data. This advantage enables the memory-efficient design of LSM tree based main-memory databases [89]. The LSM tree has also been applied to spatial databases in the AsterixDB project [17]. cLSM [35] is an LSM store supporting concurrent executions of put, get, snapshot-scan and conditional-updates. Running with concurrent merge, it supports non-blocking get and minimizes the blocking on put using readers-write block. It is implemented as an add-on to an LSM store by hooking among the three typical components of the store (i.e. in-memory, on-disk and merge components).

VIII. Conclusion

We build a trustworthy key-value store with pragmatic performance, for the data outsourcing to the public cloud. We specify the correctness of an LSM-tree based storage system by strong consistency on the frontend and the compaction specification on the backend for data maintenance. Against the local attacks in the cloud, these properties are made securely verifiable by the combined use of Merkle hash tree and Intel SGX. The Merkle hash tree is used for verifiable freshness and strong-consistency of query serving. Intel SGX, the first commodity hardware for trusted execution, is used for the verifiable data-maintenance jobs with close proximity to data. The use of trusted execution environment for verifiable maintenance is necessary, and we made our trusted codebase small and likely to be minimal. We analyze the security of our design and implement it on LevelDB with SHA1/3 for Merkle hash tree. We evaluate the performance overhead and demonstrate near-practical efficiency.

Acknowledgement

The authors would like to thank Dr. Heng Yin, Scott D. Constable, Amin Fallahi for the helpful discussion to this work.

References

[1] Apple icloud: www.apple.com/icloud/.
[2] Arm trustzone, http://www.arm.com/products/processors/technologies/trustzone/.
[3] Dropbox: www.dropbox.com.
[4] Google cloud storage: cloud.google.com/storage.
[5] Hbase coprocessor: blogs.apache.org/hbase/entry/coprocessor_introduction.
[6] http://cassandra.apache.org/.
[7] http://code.google.com/p/leveldb/.
[8] http://hbase.apache.org/.
[9] https://aws.amazon.com/s3/.
[10] Ibm scpu, http://www-03.ibm.com/security/cryptocards/.
[11] Intel corp. software guard extensions programming reference, 2014 no. 329298-002.
[12] Intel txt, http://www.intel.com/technology/security/ downloads/trust-edexec_overview.pdf.
[13] Tpm, http://www.trustedcomputinggroup.org/tpm-main-specification/.
[14] Write-optimized consistency verification in cloud storage with minimal trust, full version, https://drive.google.com/open?id=0B749HX0RkgQHLVRSczA0dEtvRUk.
[15] S. Agrawal and D. Boneh. Homomorphic macs: Mac-based integrity for network coding. In *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*, pages 292–305, 2009.
[16] M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. *PVLDB*, 8(8):850–861, 2015.
[17] 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.
[18] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *Information Security ISC 2001*, pages 379–393, 2001.
[19] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for cpu based attestation and sealing.
[20] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.
[21] 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.
[22] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *PVLDB*, 5(8):776–787, 2012.
[23] S. Bajaj and R. Sion. Correctdb: SQL engine with practical query authentication. *PVLDB*, 6(7):529–540, 2013.
[24] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Trans. Knowl. Data Eng.*, 26(3):752–765, 2014.
[25] 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.
[26] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
[27] D. Boneh and D. M. Freeman. Homomorphic signatures for polynomial functions. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, pages 149–168, 2011.
[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] 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.
[30] W. Cheng, H. Pang, and K.-L. Tan. Authenticating multi-dimensional query results in data publishing. In *Proceedings of the 20th IFIP WG 11.3 Working Conference on Data and Applications Security*, DBSEC'06, pages 60–73, Berlin, Heidelberg, 2006. Springer-Verlag.
[31] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220, 2007.
[32] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic data publication over the internet. *Journal of Computer Security*, 11:2003, 2003.
[33] 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.
[34] R. Geambasu, T. Kohno, A. A. Levy, and H. M. Levy. Vanish: Increasing data privacy with self-destructing data. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 299–316, 2009.
[35] 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.
[36] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *ISC*, pages 80–96, 2008.
[37] 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.

[38] D. Gupta, B. Mood, J. Feigenbaum, K. Butler, and P. Traynor. Using intel software guard extensions for efficient two-party secure function evaluation. In *Financial Cryptography and Data Security*, 2016.

[39] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

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

[41] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for web applications. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 895–913, 2016.

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

[43] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 880–896, 2015.

[44] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 147–163, 2014.

[45] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 1–14, 2009.

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

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

[48] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, Jan. 2004.

[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] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: bringing key transparency to end users. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 383–398, 2015.

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

[52] E. Mykletun, M. Narasimha, and G. Tsudik. Signature bouquets: Immutability for aggregated/condensed signatures. In *Computer Security - ESORICS 2004, 9th European Symposium on Research Computer Security, Sophia Antipolis, France, September 13-15, 2004, Proceedings*, pages 160–176, 2004.

[53] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2(2):107–138, May 2006.

[54] M. Narasimha and G. Tsudik. Dsac: integrity for outsourced databases with signature aggregation and chaining. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, CIKM 2005, pages 235–236, New York, NY, USA, 2005. ACM.

[55] 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.

[56] H. Pang, A. Jain, K. Ramamritham, and K. Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 407–418, 2005.

[57] 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.

[58] H. Pang, J. Zhang, and K. Mouratidis. Scalable verification for outsourced dynamic databases. *PVLDB*, 2(1):802–813, 2009.

[59] S. Papadopoulos, Y. Yang, and D. Papadias. Cads: Continuous authentication on data streams. In *VLDB*, pages 135–146, 2007.

[60] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 437–448, 2008.

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

[62] 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.

[63] B. Parno, J. R. Lorch, J. R. Douceur, J. W. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 379–394, 2011.

[64] K. Petersen, M. Spreitzer, D. B. Terry, M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 288–301, 1997.

[65] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 463–477, 2013.

[66] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 31–31, Berkeley, CA, USA, 2011. USENIX Association.

[67] 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, 2015.

[68] 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.

[69] 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.

[70] M. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: securing NFV states by using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, SDN-NFV@CODASPY 2016, New Orleans, LA, USA, March 11, 2016*, pages 45–48, 2016.

[71] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: verification for untrusted cloud storage. In *Proceedings of the 2nd ACM Cloud Computing Security Workshop, CCSW 2010, Chicago, IL, USA, October 8, 2010*, pages 19–30, 2010.

[72] R. Sinha, S. K. Rajamani, S. A. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1169–1184, 2015.

[73] 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.

[74] R. Strackx, B. Jacobs, and F. Piessens. ICE: a passive, high-speed, state-continuity scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 106–115, 2014.

[75] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 875–892, 2016.

[76] H. Sun, K. Sun, Y. Wang, and J. Jing. Trustotp: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 976–988, 2015.

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

[78] Y. Tang. On the impossibility of merkle merge homomorphism. *IACR Cryptology ePrint Archive*, 2016:617, 2016.

[79] Y. Tang, T. Wang, L. Liu, X. Hu, and J. Jang. Lightweight authentication of freshness in outsourced key-value stores. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 176–185, 2014.

[80] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 309–324, 2013.

[81] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 172–183, 1995.

[82] S. Torres-Arias, A. K. Ammula, R. Curtmola, and J. Cappos. On omitting commits and committing omissions: Preventing git metadata tampering that (re)introduces software vulnerabilities. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 379–395, 2016.

[83] 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.

[84] S. Wolchok, O. S. Hofmann, N. Heninger, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel. Defeating vanish with low-cost sybil attacks against large dhts. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*, 2010.

[85] 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, 2015.

[86] H.-J. Yang, V. Costan, N. Zeldovich, and S. Devadas. Authenticated storage using small trusted hardware. In *CCSW*, pages 35–46, 2013.

[87] 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.

[88] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Authenticated indexing for outsourced spatial databases. *VLDB J.*, 18(3):631–648, 2009.

[89] 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.

[90] Y. Zhang, J. Katz, and C. Papamanthou. Integridb: Verifiable SQL for outsourced databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1480–1491, 2015.

APPENDIX A
PROOF OF THEOREM

We present the proof for Theorem 3.3.

Proof Consider the initial system state satisfying Theorem 3.3, that is, $\forall\langle k, ts\rangle@C_i$[11] and $\langle k, ts'\rangle@C_j$ with $i < j$, then $ts > ts'$. The system state can only be mutated by one of the two operations: 1) a Put that mutates list C_0; and 2) a compaction that moves records from a lower-numberd list to a higher-numbered one. Since the theorem is about records of a single key, all the records we consider is of the same key k.

Operation 1) does not violate the invariant in Theorem 3.3, because of the following: After the Put, assume the new record inserted is $\langle k, v, ts''\rangle@C_0$. Given it is the newest record with largest timestamp ts'', for any record picked from non-zero level, say $\langle k, v, ts'''\rangle@C_{i>0}$, it will hold: $i > 0$ and $ts'' > ts'''$. Theorem 3.3 holds after Operation 1).

[11]We use $\langle k, ts\rangle@C_i$ to denote record $\langle k, ts\rangle$ resides in level C_i.

Operation 2) does not violate the invariant in Theorem 3.3. As by Invariant 3.2, a compaction moves records in a consecutive time range from a lower-numbered level to a higher-numbered one. W.l.o.g., consider the records (of key k) being moved are $\langle k, v_0, ts\rangle, \langle k, v_1, ts\rangle, \ldots, \langle k, v_l, ts\rangle, \ldots$. They are moved from level C_i to level C_{i+1}. Now $\forall\langle k, v, ts\rangle@C_j$ from the overall dataset but the moved records, and $\forall\langle k, v, ts'\rangle@C_i$ in the prestate. Consider two cases: A) $j = i$. In this case, the record $\langle k, v, ts\rangle$ will stay after in C_i after the compaction only when its timestamp falls outside the *consecutive* range of those moved records, that is, ts smaller than the timestamp of any moving records, hence $ts < ts'$. B) $j \neq i$. In this case, if $j = i + 1$, then it is irrelevant to Theorem 3.3 after the compaction. If $j < i$ or $j > i + 1$, then the moving has no effect on their level ordering before/after compaction, that is, if $j < i$ before compaction, then $j < i + 1$ after compaction. If $j > i + 1 > i$ before compaction, then $j > i + 1$ after compaction. Therefore, the theorem always holds.

APPENDIX B
CONSISTENCY CHECKING

```
class store_wrapper{
  Store store;
  Att Put(key,val){
    prePut(key,val);
    att(tsw)=store.dPut(key,val);
    return postPut(key,val,att(tsw));
  }
  Crt Get(key){
    preGet(<key>);
    <key,val>,pf(tsrw,tsr*)=store.dGet(key);
    return postGet(<key>,pf(tsrw,tsr*));
  }

  mutex State pending_wr, completed_wr,history_w;

  void prePut(<key,val>){
    pending_wr.add(<key,val,start_rt=now()>);
  }
  boolean postPut(<key,val>,att(tsw)){
    <key,val,start_rt>=pending_wr.remove(key,tsr);
    completed_wr.addW(<key,val,start_rt,end_rt=now(),tsw>);
    ac1 = completed_wr.tryTruncate();
    if(ac1 != NULL){
       assertC(acl);
       if(assertOrdered(acl,history_w))
         history_w.merge(acl);
    }
  }
  void preGet(key){
    pending_wr.add(<key,start_rt=now()>);
  }
  boolean postGet(r<key,val,tsrw,tsr,pf(tsrw,tsr*)>){
    r<key,start_rt>=pending_wr.remove();
    if(r.tsr <= history_w.latest());
      assertL2(r<key,val,tsr,tsrw,pf(tsrw,tsr*)>,history_w)
     ;
    else
      completed_wr.addR(<key,val,start_rt,end_rt=now(),tsr,
     tsrw>);
  }
}
```

Listing 2: Interfaces of verified and verifiable Put/Get

Our linearizability checking algorithm works by adaptively finding the operations that can form a consecutive segment with serialized operations, checking the consistency conditions on this segment, and then merging the segment into the serialized operations. The correctness of our algorithm depends on the following intuition: Given two consecutive operation

sets, if $\mathbb{L}1$ holds on both of them, then $\mathbb{L}$ holds on the merged segment from them. Formally,

Definition B.1: Real-time partial-order $\prec$ is a relation in a set of operations l. An operation, say o, has two attributes: invocation time $inv(o)$ and response time $resp(o) > inv(o)$. Given two operations $o_1, o_2 \in l$, $o_1 \prec o_2$ when $resp(o_1) < inv(o_2)$.

Definition B.2 (Linearizability): Given a set of operations, l, and real-time partial order $\prec$, if there exist a total order $<$ among the operations such that:

1. Real-time $\mathbb{L}1$: the real-time partial order is consistent with total-order, that is, $\forall o_1, o_2 \in l$, if $o_1 \prec o_2$, then $o_1 < o_2$. We denote it by $\mathbb{L}1(l, \prec) = TRUE$

2. Freshness (or legality as in [39]) $\mathbb{L}2$: any read operation returns the latest write in the total order.

Definition B.3 (Ordered operation sets): Assuming all operations are defined on a total-order $<$. Two operation sets, say l_1 and l_2, are ordered when all the operations of l_1 are larger than those of l_2, or when all the operations of l_1 are smaller than those of l_2.

Operation set, say l_1, is smaller than operation set l_2, denoted by $l_1 <' l_2$, if and only if the smallest operation in l_2 (based on the total order $<$ of l_2) is larger than the largest operation in l_1.

Theorem B.4: Given two ordered operation sets, if $\mathbb{L}1$ holds separately on them, then $\mathbb{L}$ holds on the merged set from them. That is, if $\mathbb{L}1(l_1) = TURE \&\& \mathbb{L}1(l_2) = TURE$, then $\mathbb{L}1(l_1 \cup l_2) = TURE$.

$\forall o_1, o_2 \in l_1 \cup l_2$, if $o_1 \prec o_2$, then $o_1 < o_2$.

Proof We consider the non-trivial case that $\forall o_1 \in l_1$, and $\forall o_2 \in l_2$. Without loss of generality, assume $o_1 \prec o_2$.

Then $resp(o_1) < inv(o_2) < resp(o_2)$. If $l_1 < l_2$, it requires $resp(o_1) > resp(o_2)$ which contradicts $o_1 \prec o_2$. Thus, $l_1 <' l_2$.

Because $l_1 <' l_2$, $o_1 \in l_1$, and $o_2 \in l_2$, we have $o_1 < o_2$. That is, $\forall o_1 \prec o_2$, we have o_1, o_2. Thus the theorem holds.

```
1  void assertC(ac1,history_w){
2    //L1
3    o0<key,val,ts> = history_w.latest();
4    o1<key,val,ts> = ac1.oldest();
5    do {
6      assertL1pairwise(o0<key,val,ts>,o1<key,val,ts>);
7      o0=o1; o1=o1.next(ac1);
8    } while (o1 != NULL)
9    //L2
10   for (read r in ac1)
11     assertL2(r<key,val,tsr,tsrw,pf(tsrw,tsr*)>,history_w);
12 }
13 void assertL2(r<key,val,tsr,tsrw,pf(tsrw,tsr*)>,history_w){
14   assert(r.tsr <= history_w.latest());
15   assert(verify(pf(tsrw,tsr*))==true);
16   assert(tsr*==tsr);
17 }
```

Listing 3: Linearizability checking