FastVer: Making Data Integrity a Commodity

Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann
Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi
Srinath Setty, Nikhil Swamy, Alexander van Renen*, Min Xu§
arvinda@microsoft.com,badrishc@microsoft.com,johannes@microsoft.com,esha.ghosh@microsoft.com
donaldk@microsoft.com,protz@microsoft.com,ravirama@microsoft.com,taramana@microsoft.com
aseemr@microsoft.com,srinath@microsoft.com,nswamy@microsoft.com,renen@in.tum.de,xum@cs.uchicago.edu
Microsoft Research, Technical University of Munich*, University of Chicago§

ABSTRACT

We present FastVer, a high-performance key-value store with strong data integrity guarantees. FastVer is built as an extension of FASTER, an open-source, high-performance key-value store. It offers the same key-value API as FASTER plus an additional *verify()* method that detects if an unauthorized attacker tampered with the database and checks whether results of all read operations are consistent with historical updates. FastVer is based on a novel approach that combines the advantages of Merkle trees and deferred memory verification. We show that this approach achieves one to two orders of magnitudes higher throughputs than traditional approaches based on either Merkle trees or memory verification. We have formally proven the correctness of our approach in a proof assistant, ensuring that *verify()* detects any inconsistencies, except if a collision can be found on a cryptographic hash.

ACM Reference Format:
Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, and Srinath Setty, Nikhil Swamy, Alexander van Renen*, Min Xu§. 2021. FastVer: Making Data Integrity a Commodity. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457312

1 INTRODUCTION

Data integrity is an important property of many software systems. It is particularly important in the cloud as customers may want to monitor their data—independently from guarantees given by the cloud provider. As an example, consider a cloud service that authenticates users using passwords. Such a system would maintain a table with usernames and hashes of users' passwords. If an administrator can tamper with this table and change passwords without being detected, they can login to the service masquerading as another user.

The main principles of data integrity are known and have been explored for decades (e.g.,[7, 19–21]). The basic idea is to store a cryptographic hash of the data (e.g., a hash of the username-password table) at a secure and trusted location. Every update to the data (e.g., registering a new user) is authorized by a processor at this trusted location and results in an update of the hash stored at the trusted location. With every access to the data (e.g., validating a password), the processor at the trusted location compares its hash with the current state of the data used for the access. If the hash does not match, the access is rejected as it indicates that the data has been tampered with in an unauthorized way (e.g., by a rogue administrator).

The classic approach to manage hashes at the trusted location is to use *Merkle trees* [21]. Merkle trees implement a hierarchy of hashes organized as a tree and store the root hash at the trusted location. To update a record, the hashes along the path from the record to the root are updated, and the new root hash is stored at the trusted location. To verify a read of a record, the processor at the trusted location receives the record and the hashes along the record-to-root path and checks whether these hashes are consistent and match the root hash stored at the trusted location. The per-operation cost is proportional to the height of the tree, so logarithmic in the database size. (We review Merkle trees in detail in Section 4.)

While the theory of Merkle trees is well explored and understood, adoption of this technology has been limited in practice to low-throughput scenarios such as protection of passwords and public blockchains. The problem is that the verification with Merkle trees is expensive and does not scale well because all update operations update the root hash and need to be serialized. To overcome the limitations of Merkle trees, *deferred memory verification*[1] has been proposed as an alternative for data integrity [3, 7, 27]. The key idea is to defer verification and verify the integrity of a batch of operations rather than each operation individually. While this approach improves concurrency [3], it incurs a high cost during verification since it involves a scan of the entire database. This might result in delaying the commit of a transaction beyond the latency that is tolerated by the application.

An important consideration for data integrity in the cloud is the choice of the *trusted location*. If it is outside of the cloud, the communication cost incurred by the roundtrips to the trusted location can be prohibitive. This issue alone has made it impractical to deploy data integrity at scale in the cloud so that there was no need to address the limitations of Merkle trees and deferred memory verification. Fortunately, the *trusted location* issue in the cloud has been addressed with the latest generation of server architectures.

[1] Also referred to as offline memory checking.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457312

These servers provide so-called *enclaves*, which are a game-changer to implement security features in the cloud. Prior work has shown how enclaves can help implement *confidentiality* efficiently in the cloud [1, 2, 4, 31]. This paper shows how enclave technology can help to commoditize data integrity in the cloud.

Enclaves or Trusted Execution Environments (TEE) are supported by most CPU families today: Intel calls its feature SGX, AMD calls it SEV-SNP, and ARM, Trustzone. While there are differences, the basic idea and value proposition are the same. An enclave is a protected region of the virtual memory space of a process containing code and data with a well-defined interface. The CPU protects the execution of code and the state within the enclave from even privileged (e.g., host OS kernel) code and, therefore, also from administrators with root privileges. Since enclaves are embedded into cloud servers, the interactions between the untrusted cloud server and the enclave are efficient.

Enclaves solve an important piece of the data integrity puzzle, the trusted location. However, this throws into sharper relief the limitations of the current state-of-the-art. Even with enclaves, Merkle trees achieve at best tens of thousands of operations per second. Deferred memory verification involves scanning the entire database in the trusted location when verifying a batch of operations which can take dozens of seconds. The goal of this paper is to overcome these deficiencies and deploy data integrity at scale with affordable cost and performance for any data service in the cloud. Consider, for instance, a database of bank accounts that are updated and accessed with millions of updates per second. There is a substantial economic incentive to tamper with such a database, yet there are also high performance and operational requirements.

Our technique to achieve strong integrity guarantees with extremely high throughput and latency guarantees is a combination of two simple yet effective ideas:

Caching within the enclave: We exploit state within the enclave to cache verification data. Rather than storing only the root of the Merkle tree our approach allows caching arbitrary nodes within the enclave. Such caching reduces cost since only the path from a record to the first cached ancestor needs to be verified. Furthermore, caching removes the root as the single point of contention since the root is no longer referenced by every operation, thereby improving concurrency. While prior work [29] has considered Merkle tree caching, our approach is simpler, more general, and more performant (Section 8).

Hybridizing Merkle and deferred memory verification: We combine Merkle trees and deferred memory verification into a new hybrid scheme that has the low latency of the Merkle trees approach and the high throughput of deferred memory verification. The core idea is to treat database records and Merkle tree nodes uniformly for integrity checking. For example, instead of always checking the integrity of a Merkle node by comparing it with the hash stored at its parent node, the hybrid scheme provides the flexibility to defer this check to a later verification scan. More generally, we use the flexibility of the hybrid scheme to organize data into a *verification memory hierarchy*: The integrity of the *cold* database records and Merkle tree nodes is checked by checking their hash against the hash at a parent node as in the Merkle trees approach. This approach is expensive since it leads to a logarithmic chain of hash checking,

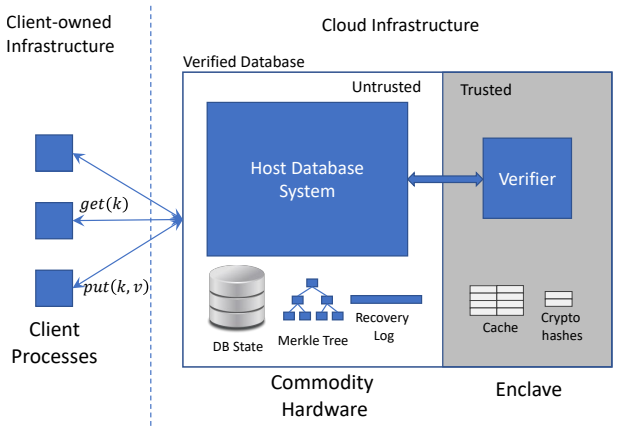


Figure 1: Architecture of a Verified Database System.

but the cost is tolerable for cold data. The *hottest* records and Merkle nodes are cached within the enclave; since these are stored within trusted memory, integrity checking is elided for such data. The integrity of *warm* records and Merkle nodes is batch-verified using the deferred memory verification approach. Our organization of data for integrity checking is analogous to traditional memory hierarchy with Merkle-tree based integrity checking playing the role of *disks*, deferred memory verification, the role of *main memory*, and enclave caching, that of CPU cache.

We have integrated our data integrity techniques into a state-of-the-art, open-source, high-performance key-value store called FASTER [8], and we call the resulting system FastVer. Our experiments show that FastVer can process more than 50 million key-value ops/sec with sub-second verification latency for a range of database sizes and workloads. Our performance numbers are competitive with Faster without data integrity (within a factor of 2) and better than commercial key-value systems such as Redis [26]. It is two orders of magnitude more performant than Merkle trees and an order of magnitude better than Concerto [3], which is the best known system based on deferred memory verification both in terms of throughput and latency.

As a unique contribution, we have formalized our hybrid scheme for data integrity checking using the F* proof assistant [30], and proven it correct. Our hybrid scheme is quite subtle and a machine verified proof ensures that our data integrity checks cover all corner cases required for overall correctness.

2 PROBLEM STATEMENT

This section defines a verified database that captures the notion of data integrity. Our definitions are derived from prior work but tailored specifically to a cloud, enclave-based setting.

2.1 Architecture

Figure 1 gives an overview of the components of a verified database system. We focus in this paper on a key-value database system that exposes a simple *get()/put()* api to clients. For simplicity, we assume that keys are 32-byte strings. If the application uses keys from a different domain, we hash the keys using a standard cryptographic

hash function such as SHA-256 to generate 32-byte keys. This change can be implemented transparently to the clients by the verified database.

An *untrusted (host)* database system running on commodity hardware processes client *get/put* requests. For *get(key)* requests, it returns the current value of the key or *null* if the key is not in the database. For *put(key, value)* requests, it updates the database and logs changes to secondary storage for durability.

What makes the architecture of Figure 1 unique is the *verifier* within the enclave. Before returning the computed result of a client request, the host database system routes the result to the verifier. If the verifier is convinced of its validity (e.g., the result of a *get(k)* reflects the current value of key k), it digitally *signs*[2] the result indicating successful validation. The client only accepts results with signed verifier validation. The details of how the untrusted host database and the verifier interact to cryptographically validate a result forms the technical core of the paper as discussed in Sections 3-7.

To implement this protocol, we extend the *get*/*put* interface as follows:

- $get(k, t) \rightarrow \langle v, s_v(k, v, t)\rangle$.
- $put(k, v, t, s_c(k, v, t)) \rightarrow \langle s_v(t)\rangle$.

Here, $s_x(m)$ denotes a digital signature of message m using the private key of a participant in the protocol, such as a client or the verifier and t, a *nonce* used to prevent replay attacks as discussed below.

A *put* request takes two additional parameters. The parameter $s_c(k, v, t)$ denotes a client signature protecting the other parameters. The verifier is initialized with the public keys of authorized clients, and it rejects any operation not containing a signature from one of them. This check ensures that only authorized clients update the database; in particular, the host database system cannot generate a valid *put* request and therefore cannot unilaterally modify the database without the verifier detecting the tampering.

The nonce t prevents *replay attacks*. The verifier checks the nonce t of a request has not been previously seen. This check ensures that a malicious actor cannot replay a previous legitimate *put* request and cause unintended database changes. As a possible implementation, to generate nonces, a client uses a counter incremented for each request to assign nonces. The verifier tracks the last nonce seen from each client and checks if a new nonce is greater than the last one.

A *get* request also carries a nonce t. The verifier validation signature for this request covers the nonce. This design prevents the untrusted host from returning a stale (validated) output for the request.

2.2 Threat Model and Integrity Guarantees

The server hosting the database system is untrusted. Thus, we assume that the adversary has complete control over the server. The adversary can cause the server to exhibit byzantine behavior and deviate arbitrarily from its prescribed protocol. In particular, the byzantine assumption implies that our integrity guarantees do not depend on the implementation details of the host database system.

We rely on cryptographic primitives such as unforgeable signatures, collision-resistant hash functions, and pseudo-random functions. We assume that the adversary cannot break the cryptographic hardness assumptions required to implement these primitives.

We further assume that the enclave is shielded from the adversary, meaning the adversary cannot tamper with the verifier's computation and state. The attacker, however, can make arbitrary calls to the verifier using the verifier's API. Furthermore, the adversary can reboot the enclave, which resets the verifier to its initial state. We assume that the verifier maintains a small amount of persistent state to hold a single hash value; we use this state to prevent rollback attacks. In practice, such persistent state can be implemented using TPMs or a public blockchain using protocols such as Memoir [24].

We assume that clients are trusted and inaccessible to the adversary. However, any communication between the client and the verifier goes through the untrusted server and is therefore under the adversary's control. We assume that the clients and the verifier share cryptographic secrets such as symmetric keys to efficiently exchange and authenticate messages. Such shared secrets can be established using, e.g., a public key infrastructure.

The verifier code and the client code that checks verifier validation is the *trusted computing base (TCB)* of the verified database. In other words, our integrity guarantees rely on this code being correctly implemented.

We provide strong integrity guarantees based on sequential consistency [16]: An output seen by a client (and verified by the verifier) reflects some sequential ordering of historical updates. We do not provide guarantees on progress and availability and the adversary has the power to arbitrarily disrupt the database service or provide preferential treatment to one client over another. Again, these assumptions are common to other work on data integrity.

2.3 Performance Goals

As we discuss in subsequent sections, there are different solution approaches for implementing the verified database that differ in the design of the verifier and the details of how the untrusted host cryptographically proves the validity of an operation result to the verifier. We present here various performance desiderata that we use to characterize and compare different solution approaches. We show that our hybrid approach meets all the desiderata while prior approaches do not.

P1: Size of the Verifier State: Enclaves provide a limited amount of trusted memory, and the verifier can use this memory to improve performance. The performance of the verified database system, however, must degrade gracefully with the size of the memory allocated to the verifier. In particular, a solution should not rely on the verifier storing the whole database to achieve good performance.

P2: Verification Complexity: Verification complexity refers to the additional computation performed by the host and the verifier to validate an operation result. Verification complexity impacts the overall throughput of the verified database. The verification complexity should be $O(1)$ with a low-constant for operations over

[2] Alternately, the clients and the verifier establish a secure channel and use more efficient message authentication codes instead of digital signatures.

frequently accessed records. Given a lower-bound proof in prior work [11], we cannot avoid worst-case logarithmic verification complexity, but we seek solutions where this happens rarely, such as for operations over cold records.

P3: Verification Latency: One of the techniques for data integrity involves verifying operations in a batch for better performance. Such batching introduces latency in communicating a validated result of a client operation. A solution approach for verified databases should allow the client application to control latency, e.g., specify a latency bound of one second. In particular, the database size should not limit the size of the latency budget a client can set.

P4: Concurrency Bottlenecks: Verification should not introduce concurrency bottlenecks beyond those arising from the client workload. In other words, the concurrency bottlenecks of a verified database should be no different from that of a regular system without data integrity. An example of such a bottleneck is the Merkle root hash discussed in Section 1. As another example, if the verifier is single-threaded, the verifier thread becomes a contention point for the host threads seeking to validate results.

3 TRUSTED DATABASE APPROACH

A simple approach to implement a verified database system is to run the entire system within an enclave. The untrusted host merely relays requests and responses to and from the clients to the enclave database system. When the entire database fits within the enclave memory, this approach is sound and achieves good performance.

The main drawback arises from limited enclave memory. For example, the total amount of memory of an (Intel Coffee Lake) SGX enclave, for both code and data, is less than 200 MB, barely enough to fit a database of size 10M records with 8-byte keys and 8-byte values. This solution approach meets the performance goals *P2*-*P4* of Section 2.3, but not *P1*.

While the "trusted database" approach has an impractically high enclave memory requirement, it provides us with the concept of verifier *caching* that we leverage in subsequent sections to design better schemes. Consider a strawman instantiation of the architecture in Figure 1 similar to the trusted database where the verifier mirrors the entire key-value database within the enclave. To validate an output v for request $get(k)$, the verifier checks if its mirrored database has the record $\langle k, v\rangle$. To validate a $put(k, v')$ operation, the verifier updates the current value of k with v'. The mirrored database state within the verifier is correct for the sequence of operations that the verifier witnesses implying the validations by the verifier reflects this sequence.

To avoid storing the entire database within the enclave, we extend the strawman by adding mechanisms to "page-out" verifier records to untrusted storage outside the enclave and "page" them back in when needed for validation. The collection of records within the verifier is at any point a cached subset of the overall database. The problem with this approach is that the untrusted host could tamper with a paged-out record. If we, therefore, add data integrity checks using which a verifier can ensure that the content of a record (as identified by its key) when paged-in is identical to the content when it was last paged out, the execution of the extension becomes isomorphic to the strawman above.

The remainder of the paper develops increasingly sophisticated techniques for such data integrity checks for verifier caching that ultimately meet the performance goals of Section 2.3.

4 MERKLE TREES AND ENHANCEMENTS

4.1 State-of-the-Art: Sparse Merkle Trees

Merkle trees have been used for data integrity for decades [14, 18, 21]. A Merkle tree is a hierarchy of cryptographic, collision-resistant hashes constructed over the database records. The leaves of the tree contain hashes of the database records computed using a standard hash function such as SHA-256. An intermediate node contains the hash of its children. We can prove that the root hash is a collision-resistant hash of the entire database, i.e., the root hashes of two different databases collide with negligible probability.

To implement a verified database using Merkle trees, the verifier stores the root, and the host stores the rest of the Merkle tree. To validate the result v of a $get(k)$ operation, the host sends the verifier the ancestors of the referenced record $\langle k, v\rangle$ and their siblings; it does not send the root which the verifier already has. The verifier checks if each ancestor hash equals the hash computed from its children. If all these checks pass, it validates (signs) the value v. We can prove using the collision-resistant property of hashing that at least one of the checks fails if v is not the current value of key k. To validate a $put(k, v')$, the host sends the same ancestor hashes as for $get(k)$. The verifier uses the updated value v' to recursively compute the ancestor hashes and update the stored root.

One particular challenge is to verify a key does *not* exist for a *get* request that returns *null*. Prior work [9, 17, 23] has proposed a variant called a *Sparse Merkle Tree* to prove the non-existence of keys[3]. Conceptually, a sparse Merkle tree has a leaf key-value entry for every key in the key domain (which could be unbounded), using a special value *null* for non-existent keys (hereafter, *null-keys*). For a large key domain, most values would be *null*, and sparse Merkle trees use a representation that exploits the sparsity of non-null keys to use space linear in the number of non-null keys. A well-known implementation [9, 17, 23] builds a *Patricia trie* [22] over non-null keys. It augments this data structure by storing with each pointer a cryptographic hash of the pointed node.

Next, we present enhancements to sparse Merkle trees that change their representation to leverage verifier caching and change how updates are propagated to ancestor hashes. These enhancements improve the performance of the Merkle-tree-based approach while also providing a building block for the hybrid scheme of Section 6.

4.2 Encoding Sparse Merkle using Records

We encode Patricia sparse Merkle tree as a collection of records in $\langle key, value\rangle$ format, one for each node of the tree. With this change, there are two kinds of records: *data records* originate externally from the client and are referenced in *get*/*put* operations; *merkle records* are internal to the verified database and help with verification. We extend the prefixes *data-* and *merkle-* to apply to keys and values, so, for example, a *data key* refers to the key field of a data record and a *merkle value* refers the value field of a merkle record.

[3] Alternately, we can use a more complex verification mechanism where the host sends two paths to the verifier to prove the non-existence of a key [18].

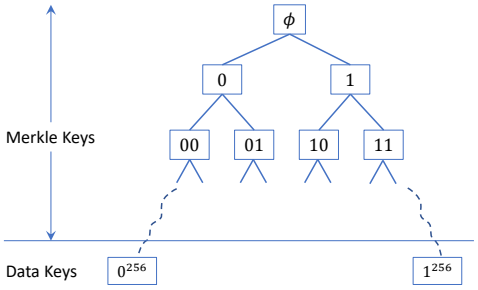


Figure 2: Tree relationship of Merkle and Data Keys.

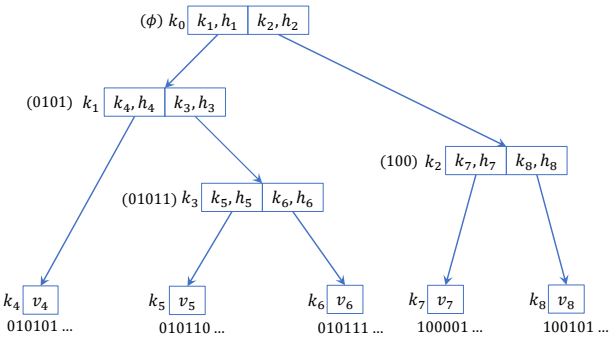


Figure 3: Example Sparse Merkle Tree

KeyName	Key	Value
k_0	ϕ	$\langle\langle k_1, h_1\rangle, \langle k_2, h_2\rangle\rangle$
k_1	0101	$\langle\langle k_4, h_4\rangle, \langle k_3, h_3\rangle\rangle$
k_2	100	$\langle\langle k_7, h_7\rangle, \langle k_8, h_8\rangle\rangle$
k_3	01011	$\langle\langle k_5, h_5\rangle, \langle k_6, h_6\rangle\rangle$
k_4	010101 ...	v_4
k_5	010110 ...	v_5
k_6	010111 ...	v_6
k_7	100001 ...	v_7
k_8	100101 ...	v_8

Figure 4: Record view of the sparse Merkle tree of Figure 3. The first four are Merkle records and the last five are data records (generated by the client). The symbols k_i, v_i, and h_i are names that represent binary strings.

As mentioned in Section 2.1, data keys are drawn from $\{0,1\}^{256}$. The domain of merkle keys is $\{0,1\}^d, d < 256$, i.e., a merkle key a bit string of any length up to (but not including) 256. We logically organize the set of all (merkle and data) keys as a binary tree with the empty string being the *root* (denoted ϕ) and the string k being the parent of strings $k \cdot 0$ and $k \cdot 1$, where $\cdot$ denotes string concatenation. We can view string k as the encoding of the path from the root, with the length of the key being its *depth* in the tree (see Figure 2). A key k' is an ancestor of a key k, if k' is a prefix of k. If k' is an ancestor of k and $k' \neq k$, we call it a *proper* ancestor. When k' is a proper ancestor of k, we use $dir(k, k') \in \{0,1\}$ to denote whether k is a left (0) or right (1) descendant of k'. For example, $dir(1011, 1) = 0$.

The domain of merkle values is a pair $\langle kh_0, kh_1\rangle$, where each kh_i is either null or a *key-hash* pair $\langle k_i, h_i\rangle$, where k_i is a descendant key and h_i, the cryptographic hash of the value associated with k_i. Informally, k_i is the least common ancestor of all non-null data keys along the left or right subtree of k_i. Figure 3 shows the record encoding of an example sparse Merkle tree in "tree" form; Figure 4 shows the same in tabular form. There are three non-null data keys in the left subtree of the root, and the least common ancestor of these keys is $k_1 = 0101$. The value associated with the root stores the key k_1 and the hash h_1 is the hash of the value $\langle\langle k_4, h_4\rangle, \langle k_3, h_3\rangle\rangle$ associated with k_1.

Example 4.1. We illustrate the implementation of a verified database system using the record encoding of sparse Merkle trees. The verifier stores the root record. Consider the database instance in Figure 3-4 and an operation $get(k_6)$. To get the verifier to validate the output v_6 for this operation, the host sends the records corresponding to the keys k_6, k_3, and k_1 along the path from k_6 to the root. The verifier uses the hash h_1 (stored internally as part of the root record) to verify the integrity of record k_1 (and therefore the hash h_3 stored in it). It uses the hash h_3 to verify the integrity of record k_3, and so on. For a $get(k)$ with a non-existent key $k = 010100\ldots$, the host sends the single record with key k_1, the least ancestor of k in the tree. Given that the left and right descendant of k_1 correspond to the keys 010101 ... and 01011, the verifier can validate that key k does not exist. □

4.3 Merkle trees with Verifier Caching

In Section 3, we presented a general approach where the verifier uses cached records to validate the operations referencing the records. To ensure correctness, the verifier needs to check that the content of a record added to the cache is identical to the content when it was last removed. We can naïvely use Merkle trees to implement this data integrity check: e.g., to add a (data) record to the cache, the host sends the merkle records along the root-to-leaf path as illustrated in Example 4.1.

We improve this approach by bringing merkle records into the ambit of verifier caching: any record (merkle or data) can now be added to and removed from the verifier cache. We use the availability of merkle records in verifier cache to simplify data integrity checks when adding and evicting records: when a (merkle or data) record $\langle k, v\rangle$ is added, the verifier checks that the *tree parent* of key k is in its cache, and checks that the hash stored in this record matches the hash of value v. The tree parent of a key is the key in the sparse merkle tree that "points" to the key; in Figure 3, for example, the tree parent of k_4 is k_1. To evict a record $\langle k, v'\rangle$ from its cache, the verifier again checks that the tree parent is in its cache and stores the hash of the updated value v' within parent record. The root record is always in the verifier cache and never evicted. Adding and evicting records to the verifier cache are now constant time operations rather than logarithmic in the database size.

Example 4.2. Since the root record is in the verifier cache, the record of key k_1 of Figure 3 can be added to the verifier cache. Next, the record of key k_3 can be added, and finally, the data record $\langle k_6, v_6\rangle$ can be added based on its tree ancestor k_3 already in the cache. Once $\langle k_6, v_6\rangle$ is in the cache, the verifier can validate *get* and *put* requests for k_6 knowing its value v_6 is correct. □

Adding the record $\langle k_6, v_6 \rangle$ requires the same number of hash computations (three) as the path-based check of Example 4.1. We have replaced a check involving a root-to-leaf path with smaller checks, one for each edge of the path. However, we can now exploit locality-of-reference of merkle records. We save hash computations by keeping a merkle record cached between two references. For example, given the verifier cache state after Example 4.2, we can directly add the record $\langle k_5, v_5 \rangle$ since its parent k_3 is already in the cache. A single hash computation is sufficient to check its integrity.

4.3.1 Lazy hash updates. In traditional Merkle trees, any update to a data record propagates to the root. An important feature of our caching is *lazy updates*. Any update of a (merkle or data) record is propagated to the immediate tree parent and not to ancestors beyond. The change propagates to the next ancestor when the parent record is evicted.

Example 4.3. Continuing Example 4.2, the verifier cache has five records: k_0 (root), k_1, k_3, k_5, and k_6. Assume the verifier validates two operations, $put(k_5, v_5')$ and $put(k_6, v_6')$, that update records k_5 and k_6 in the cache. If the record $\langle k_5, v_5' \rangle$ is now evicted from the cache, the verifier updates the record k_3 with the hash of the new value v_5'. It does not recursively update records k_1 and k_0. The hashes stored in records k_1 and k_0 are now stale, but we can prove this does not affect correctness since record k_3 is cached. Similarly, when record k_6 is evicted, the update is not propagated beyond k_3.

Lazy updates yield performance benefits when there is locality-of-reference of merkle records. With lazy updates, a single merkle record accumulates updates to all its descendants during the time it is cached and propagates them to its parent record with a single hash computation when it is evicted. To exploit this property, our hybrid scheme "manufactures" locality of reference by batching updates and applying them in sorted order.

4.3.2 Correctness. We argue that when a record $\langle k, v \rangle$ is evicted from the verifier cache, it cannot be subsequently added with a different value $v' \neq v$. This is because the hash of value v is stored at the tree parent of k and checked when the record is added. The parent may itself be evicted and added in the meantime. We use induction to prove that if the tree parent is evicted, it is added with the correct value (which includes the hash of v). This informal argument misses several details, and we defer them to a formal proof. For example, how does the verifier know what the tree parent of a key k is? In fact, it relies on the untrusted host to specify the parent key, but we can prove that the host cannot specify an incorrect parent without being detected.

4.4 Discussion

Traditional Merkle trees do not meet the performance goal *P2* since each operation incurs a logarithmic verification cost or the performance goal *P4* since every operation touches the root. They meet the performance goal *P3* since they do not add any verification latency beyond the time required for the hash checks; this contrasts favorably with deferred memory verification we discuss next, which introduces latency to accrue the benefits of batching.

Our enhancements use enclave memory to alleviate, but not eliminate, the limitations around *P2* and *P4*. For example, by caching higher levels of the Merkle tree, we eliminate root as the single point of contention. The verifier performs fewer hash computations, but the verification complexity is still logarithmic unless the cache size is large enough to fit most merkle records. In concrete terms, the changes improve the performance of Merkle trees from around 10k ops/sec to around 100k ops/sec.

More importantly, our improvements lay critical groundwork for the hybrid scheme which achieves two order-of-magnitude higher performance. Our changes break up the monolithic data integrity checking of traditional Merkle trees into finer-granularity checks: to check the integrity of a record, it suffices to check if the hash of its value field is equal to the hash stored at its parent record, once the parent is in the verifier cache. This change does not seem significant since the parent record is itself recursively checked in the same way. The real significance emerges with deferred memory verification we present next since it opens up the option of using a different mechanism for checking the integrity of the parent.

5 DEFERRED MEMORY VERIFICATION

An alternative approach to data integrity is *deferred memory verification*. The theoretical underpinnings of this approach were presented by Blum et al. [7], and recent work has introduced several extensions to make this approach practical [3, 28].

5.1 State-of-the-Art Deferred Verification

Deferred memory verification relies on a cryptographic primitive called *collision-resistant multiset hashing*, a hash function over multisets (bags) with the property that two different multisets hash to different hash values with high probability.

To implement a verified database using deferred memory verification, we do not require additional data structures such as Merkle trees. To validate an operation ($get(k, t)$ or $put(k, v', t, s_c)$ with nonce t) involving key k, the host sends the verifier the current record $\langle k, v \rangle$ in the database. The host is untrusted, so v could be a tampered value. The verifier does not have sufficient information to check the integrity of the record right away. Instead, it does some internal bookkeeping and signs a *provisional* validation $s_v(k, v, t, e)$ indicating it has recorded the operation as part of some batch e. At a later point, the host interacts with the verifier to perform integrity checking for all operations in the current batch e. If these checks pass, the verifier signs a *batch* validation $s_v(e)$. The provisional and batch validations together serve as the overall validation of the operation which the hosts forwards to the client.

The crucial detail in the description above is the verifier bookkeeping. For each operation ($get(k)$ or $put(k, v')$), the verifier adds the *pre-image*, the record $\langle k, v \rangle$ that the host provides, to a *read-set* that it tracks. The verifier also adds the *post-image* reflecting the effects of the operation to a *write-set* that it tracks. For a $get(k)$, the pre- and post-images are identical, and for a $put(k, v')$, the post-image reflects the new value v'. If the host is honest, the pre-image added to the read-set is identical to the post-image that was previously added to the write-set during the last operation referencing key k. Therefore, the read- and write-sets are equal, except the write-set contains one additional entry for each key corresponding to its most recent post-image. During batched verification, the host sends the entire database to the verifier. The verifier adds these records to the read-set and checks if the read- and write-sets are

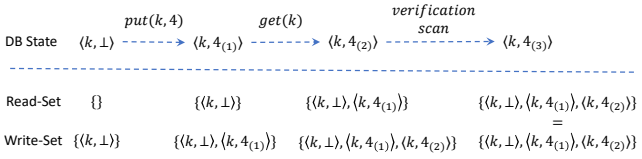


Figure 5: Offline memory checking for a toy database with one key, referenced in Example 5.1

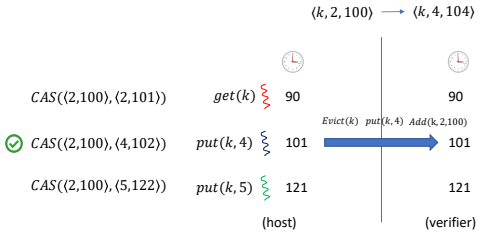


Figure 6: Example 5.2 illustrating the actions of three threads concurrently perform operations on a single record.

equal; any dishonest behavior by the host results in this check failing. The verifier does not materialize the read- and write-sets but maintains hashes of these sets using a multi-set hash function which is sufficient for the equality checking during verification.

Example 5.1. Figure 5 illustrates these ideas for a database of one record with key k and two operations. To validate $put(k, 4)$, the host sends the verifier initial record $\langle k, \perp \rangle$. The verifier adds this record to the read-set and the updated record $\langle k, 4 \rangle$, to the write-set. (We discuss the subscript (1) later.) For $get(k)$, the host sends the updated record $\langle k, 4 \rangle$, and the verifier adds the same record to both read- and write-sets. At all times, the write-set has one more entry than the read-set. The verification scan adds this entry to the read-set, making these two sets equal. If the host is malicious and sends an invalid record $\langle k, 5_{(1)} \rangle$ for $get(k)$, the invariant is broken. The read-set after this operation contains an element not present in the write-set, which leads to a subsequent verification failure. □

Our informal description elides subtle details essential for correctness. In particular, deferred memory verification associates a *timestamp* with each record. The verifier maintains an internal clock (counter) that is incremented each time a record is touched. The timestamps are shown as subscripts in Figure 5 and are included when adding a record to read- and write-sets. The host is required to store the timestamp and present it when the record is referenced next.

Concerto [3] introduces practical improvements to the theoretical construction of Blum et al [7], including the use of *epochs* to make verification recurring and an efficient AES-based multiset hash function. Informally each verification completes an epoch and starts a new one, and all the operations that happen within an epoch are verified in bulk by the verification operation at the end of that epoch. In this paper, we build on these and introduce two other modifications that significantly improve performance.

5.2 Deferred Verification with Caching

We instantiate the verifier-cache-based approach of Section 3 to use deferred memory verification to check that the content of a record added to the cache is identical to that when it was last removed. When a record is added to the cache, the verifier adds it to its read-set; when a record is evicted from the cache, the evicted version is added to the verifier's write-set. During verification, any cached record is ignored. The only change to that the validation of operations is provisional until the next verification scan.

5.3 Improved Concurrency

While the deferred memory verification technique of Concerto [3] has significantly better concurrent performance than Merkle trees, it still has two global contention points. First, there is a single verifier clock referenced and updated by all operations. Second, all interactions with the verifier across different host threads need to be fully serial. To keep the host threads and the verifier decoupled, the interaction between the host threads and the verifier happens using a log buffer; however, all operations are serialized into this single log. The performance benefits of Concerto arise since both these global contentions can be managed using lock-free operations. This means that the maximum rate of lock-free operations on a single data element is an upper bound on the performance of Concerto.

To break this ceiling, we introduce multiple *minimally interacting verifier (threads)*: each verifier thread has its own clock, own cache, own read-set and write-set hashes, and does not communicate with other verifier threads except at the end of verification epoch to aggregate local set-hashes. (As we will see in Section 7, each set hash is a small 16-byte value, and aggregating hashes is a simple xor operation.) The verifier logic is otherwise unchanged except for one critical detail: when the host adds a record to a verifier cache, if the record's timestamp is greater than the local clock of the verifier, verifier updates its clock to one larger than the timestamp. This is analogous to the update rule of Lamport's clocks [15]. We emphasize there are no constraints on which record can be added to the cache of which verifier thread. The same record can visit different verifier caches over its lifetime. We have formal proof that if at the end of verification, the aggregated read-set (hash) is equal to the aggregated write-set (hash), there *exists* a serialization of all operations that satisfy data integrity.

We use the extensions above to remove the concurrency bottlenecks of Concerto. We pair one verifier thread with one host thread. Just like in Concerto, the host thread interacts with its verifier thread asynchronously using a buffered log; however, the log is now local to a host thread eliminating contention. (Our implementation uses the same OS thread to assume the role of both the host- and verifier threads, further eliminating producer-consumer contention on this log.) Unless otherwise mentioned, a *log* refers to the asynchronous communication log between the host and verifier, not to the recovery log.

Each host thread operates in a simple loop processing *get/put* operations. For an operation on a key k, it adds (asynchronously, using the log) the record of key k to its local verifier cache, (provisionally) validates the operation, evicts the record from the cache, and updates the record in the database to reflect possibly new value and the new timestamp derived from its verifier's clock. Two or more

host threads touching different keys proceed entirely independent of one another.

An important detail is managing contention when two or more threads access the same key. A naïve approach is for a thread to lock the key, interact with its verifier, apply updates, and release the lock. Our implementation is lock-free for records with small (e.g., 8-byte) value fields and uses short-lived locks for larger fields. While the verifier clocks are *protected state* (meaning, if they can be tampered, the security guarantees can be broken), they are not confidential. A host thread can simulate and, therefore, mirror the changes of its verifier's clock. This means that the host thread can predict the record's updated value with the new timestamp, without requiring a roundtrip to the verifier. Based on this observation, a host thread speculatively attempts to update a record before serializing the operation to its log. If there is no contention, the update succeeds, and the host thread proceeds with the serialization a posteriori. If another thread wins out, the host thread retries.

Example 5.2. In Figure 6, three threads process concurrent *get/put* operations on a key k with value 2 and timestamp 100. Each thread tries to compare-and-swap install a new value and timestamp based on its local verifier clock. For example, the first (red) thread calculates that the result of adding the record to its verifier cache (which updates its clock to 100), validating the *get(k)* operation, and evicting the record results in the updated (value, timestamp) of $\langle 2, 101\rangle$; it, therefore, tries to install this value-timestamp. The second thread wins the CAS, so it proceeds with logging its verifier interactions; the other threads retry with the new state of the record. □

5.4 Discussion

Deferred memory verification meets three of the performance goals listed in Section 2.3. It is efficient, has minimal dependence on verifier caches, and the improvements presented in this section make the resulting verification scheme highly concurrent. However, it does not meet the goal P3 on bounded verification latency. Verification latency is linear in the database size since each record is routed to a verifier thread during the verification scan. We note that this is wasted work for rarely accessed records. In the next section, we combine the Merkle and deferred memory verification approaches to achieve the performance goals of Section 2.3. An enabling idea is that deferred memory verification does not assume anything about a record other than its key-value structure, so merkle records can be protected using deferred memory verification.

6 HYBRID APPROACH

Sections 4 and 5 presented two traditional approaches to data integrity with various enhancements. While these enhancements improve the performance of each approach in isolation, they do not change the fundamental tradeoff: Deferred memory verification has high throughput and concurrency but incurs high verification latency. Merkle trees have low latency but achieve lower throughputs and support limited concurrency.

A central contribution of this paper is a hybrid approach that lets us combine the advantages of both. Our encoding of Merkle nodes as key-value records and the use of verifier caches to validate operations is crucial to this hybrid approach. The integrity of a record (data or merkle) can be ensured in any of three ways: (1) it can be kept within a verifier cache, a protected state; (2) it can be protected by storing a cryptographic hash of its value in a different, specially identified record (its Merkle tree parent); or (3) it can be protected by recording its value a in write-set hash. For options (2) and (3), the record is stored in untrusted storage. However, its state recorded in the merkle parent record or the write-set hash ensures that the host cannot corrupt the record without being detected when it is next added to a verifier cache. Over its lifetime, a record can transition from one integrity protection mechanism to another independent of other records. We argue in Section 6.4 that no matter how we arrange such transitions, we correctly ensure data integrity of all records, assuming verification checks pass.

6.1 Verification Hierarchy

Our hybrid approach allows us to organize records into a *verification hierarchy* based on their access characteristics. Caching is at the top of the hierarchy. It is fastest since it does not involve any hash computations but also expensive since, in practice, enclave memory is a limited resource. So caching is ideal for *hot* records and for records that we know will be accessed in the near future. As we discuss in Section 6.3, we leverage the flexibility of the hybrid scheme to "manufacture" locality of reference to better exploit verifier caches.

Deferred memory verification is at the next level of the hierarchy. Protecting a record using deferred verification is less expensive than using Merkle hashing since the latter could induce a chain of $O(\log n)$ hash computations. On the other hand, to ensure bounded verification latency, we need to bound the number of records protected using deferred verification since the verification latency increases linearly with this number. There are no such constraints for protecting a record using Merkle hashing, and this approach serves as a fallback option for records that are not hot/warm for caches or deferred verification. Therefore protecting using Merkle hashing is analogous to secondary storage in a memory hierarchy.

We note that the notion of hotness or coldness of records applies naturally to merkle records as data records. For example, the descendants of a merkle record r could be in aggregate accessed frequently without any of the individual records being frequent. In such cases, it is beneficial to keep the record r protected using deferred verification or in a verifier cache; for any descendants of r, the chain of Merkle hashing stops at record r, and none of its ancestors need to be touched.

6.2 Parallelizing Merkle-Hashing

Merkle-tree-based verification (Section 4) is hard to parallelize across multiple verifier threads. This is because to add a record to a verifier cache, its Merkle tree parent needs to be in that cache, which in turn would have required its parent to be in the same cache when it was added, and so on. By an inductive argument, it follows that records can be added only to the verifier thread whose cache contains the root record; therefore, all Merkle hash computations required for verification happen within that verifier thread.

In the hybrid approach, we can use deferred verification to break this chain and parallelize Merkle hashing. Merkle records that are protected using deferred verification are "unshackled" from their parent records and can be added to the cache of any verifier thread.

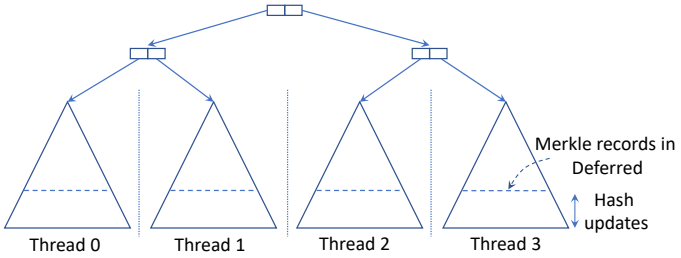


Figure 7: FastVer: organization of records

We use this property to partition Merkle records among the available threads in the system. With the sorted-merkle updates optimization below, all hash computations required for Merkle tree updates parallelize across all the threads in the system.

6.3 Sorted Merkle Updates

As we discuss in Section 8.5, there is an order of magnitude difference in performance when updates to the data records are applied in sorted order as opposed to random order, arising from better locality of reference for merkle records. As a subtle optimization the details of which are discussed in the full version of the paper, within each epoch, we temporarily record data updates using deferred verification, sort the keys touched by these updates and apply them back to Merkle in sorted order.

6.4 Correctness

We have formal proof that any collection data operations validated using our hybrid approach is correct, meaning that if the checks at all the verifier threads succeed, then the validated operations are *sequentially consistent* up to the last epoch verified. The proof relies on some subtle additional checks discussed in the full version of the paper. We have also formalized our core hybrid approach scheme in the F* proof assistant [30] and proven it correct. Our development involves around 20K lines of code and proof.

We emphasize that the host database system (prover) is untrusted for the above claims. The host can choose to arbitrarily violate the state machine and try adding a record that was evicted using the Merkle mechanism using the deferred verification mechanism. Such malicious behavior would result in some verifier checks failing. In other words, the verifier correctly detects the malicious behavior of the host as desired. Also, outside of the cache, each verifier thread uses $O(1)$ state (for the add-set and evict-set hashes). Finally, the verifier code is subtle but simple and our current implementation is around 500 lines of C++ code, not counting the code in the crypto libraries.

7 IMPLEMENTATION

Our implementation of FastVer is based on the C++ implementation of Faster available at [12]. FastVer is an application over unmodified Faster. Faster has an extensible api where we can specify custom logic for key-value operations, and our implementation instantiates this api with code required for verification. We add a 64-bit *aux* field to each value, used for internal bookkeeping and Faster manages the database of keys and values including the aux field; this includes persisting them for durability.

Thread model: Data processing with FastVer involves n identical worker threads running in a loop, processing client operations. For each operation, in addition to unmodified Faster processing, the worker performs the verification work that includes ensuring the record referenced by the operation is in the verifier cache, getting the operation validated by the verifier, and returning the validated result to the client. There are n logical verifier threads and the same OS thread performs both the host database processing and the verifier processing after entering the enclave. To amortize the overhead of entering and exiting the enclave, the worker serializes the verifier calls into a *verification log buffer*; when the buffer is full, the worker thread enters the enclave and processes the entries in the buffer. Each worker generates its own log stream and there is no log contention between different workers.

Worker inner loop: The aux field associated with a value encodes the current protection mechanism (cache or deferred verification or merkle-hashing based verification); for a record protected using deferred, it provides the associated timestamp; for a record in a verifier cache, it provides details of verifier thread and slot where it is cached. As an example of how the aux field is used and updated, consider a worker thread processing $put(k, v')$ and assume the aux field indicates the $\langle k, v\rangle$ is in deferred state with timestamp t. The worker generates a new timestamp t' that predicts the evict timestamp when the record is added to the verifier and subsequently evicted; since the verifier clock advances in a predictable manner, t' can be computed without actually running the verifier. The worker thread atomically updates $\langle v, t\rangle$ to $\langle v', t'\rangle$; this is accomplished using record-level mutexes in general and 128-bit CAS for 8-byte value fields. If the atomic update succeeds, the worker thread generates log entries that add the record to its verifier cache, validate the put operation, and evict the record. The actual steps would differ if the record were in a different state, but the overall template remains the same: a small, contended, atomic update step over value and aux fields, followed by uncontended generation of log and other bookkeeping for background tasks.

Background work: Every M operations for some configured value M, a worker performs background work essential for verification. This includes starting an epoch verification, migrating records from epoch e to $e+1$ required for deferred verification, sorting keys and applying Merkle updates, checking the fill status of the verification log buffer and running verification if it is full.

Cryptography: We use a C-implementation of Blake3 [6] for Merkle hashing. For multiset hashing, we use the construction suggested in Concerto with AES-CMAC as a PRF. We use an vectorized assembly implementation of AES using Intel AES-NI instructions based on a sample code provided by Intel [13].

Durability: The durability guarantees of FastVer are similar to those of prior systems [3], and a detailed description of durability is not a focus of this paper. Briefly, durability is implemented using standard logging techniques. We rely on CPR logging of Faster [25], which aligns well with epoch-based verification. By synchronizing the epochs in FastVer with that of Faster's CPR logging, we get the guarantee that when epoch e is completed, all

the database state is persisted. In addition to the database state, we checkpoint the verifier state, protected from tampering by a verifier signature. This includes the evict-set hashes of epoch $e + 1$ and a cryptographic hash of the verifier cache contents. We note that durability only guarantees recovery from media failures, not availability, since an adversary can destroy logs and make the system unrecoverable.

8 EXPERIMENTAL EVALUATION

This section presents the results of our experimental evaluation. The goals of the evaluation are: (1) quantify the cost of data integrity by comparing the performance of FastVer against Faster as a baseline; (2) illustrate the benefits of our hybrid design through evaluating the throughput and latency characteristics of FastVer; (3) compare with prior approaches; and (4) study scalability characteristics of FastVer.

Setup: We used an Ubuntu 18.04 machine with Intel Xeon 6140, 2.3GHz, two-socket, 36 core machine (18 physical cores x 2 with hyperthreading) for our experiments. The machine has 512 GB of main memory. This machine does not support Intel SGX enclaves, and we used it for our simulated enclave experiments, as discussed below. For true enclave experiments, we also run FastVer on an Microsoft Azure Confidential Compute VM with Intel SGX; this is a DC8_v2 VM with 8 vCPUs and 32 GB of RAM. We note that the current generation of SGX machines are fairly low-end and do not support a large number of CPU cores.

Benchmark: We use a variant of the YCSB [10] benchmark with 8-byte keys and 8-byte values. We vary the database size to illustrate different performance characteristics. For a database of size N, the domain of the keys is $0, \ldots, N-1$. As discussed in Section 2, we map the 8-bytes keys to 32 bytes by padding. YCSB benchmark specifies different workload characteristics with different ratios of *get* and *put* operations: YCSB-A is update-heavy with 50% *get*s and 50% *put*s; YCSB-B is read-heavy with 95% *get*s, YCSB-C is read-only, and YCSB-E is a scan-based, mostly-read-only benchmark. Unless otherwise mentioned, we use a zipfian distribution with $\theta = 0.9$ for key selection as proposed in the original benchmark [10].

Systems Evaluated: We use an unmodified Faster as a baseline to quantify the cost of data integrity. We use two variants of FastVer, one with true enclaves and one with *simulated* enclaves. In a simulated enclave, verifier calls are regular function calls with added delays to model enclave switching costs, following an approach used in prior work [5]. Simulated enclaves allow us to perform large-scale experiments without the constraints imposed by the current enclave technology. Our experiments of Section 8.2 indicate the performance of simulated- and real-enclaves are comparable, so our learnings with simulated enclaves should be relevant when more powerful enclave supporting processors become available in the future. We use different configuration options for FastVer depending on the experiment and we specify these with the experiment. Unless otherwise mentioned, we use 32 workers with simulated enclaves and 8 workers with true enclaves and a verifier cache of 512 entries for each verifier thread.

We also use micro-benchmarks (Section 8.5) loosely based on YCSB to profile different sub-components of FastVer. These experiments help illustrate the relative contributions of different techniques and explain end-to-end performance characteristics, including current bottlenecks. We also use these experiments to compare our techniques with those of prior work. Such comparisons at the technical level are more insightful than a system-level comparison; e.g., VeritasDB [29] uses RocksDB as the backend store, so a system level comparison is dominated by the relative performance of Faster and RocksDB, which is heavily biased in favor of FastVer.

8.1 Experiment: Throughput vs. Latency

YCSB-A: In this experiment, we evaluate the throughput vs. latency characteristics of FastVer for databases of 2M, 8M, 64M, and 128M records. FastVer offers two parameters to control the verification latency: the first is *batching*, where we batch a specified number of operations before initiating the verification scan. The second is the count of merkle records that are protected using deferred verification; the larger the number, the greater the verification delay since these records need to be migrated to the next epoch during every verification. Currently, the second parameter is an integer d and all merkle records at depth d are kept in deferred state.

We measured the throughput and latency for an exponentially varying values of batch size and different values of depth d as indicated above. Each experimental run involved executing 2^{32} (4 billion) key-value operations over 32 worker threads and measuring the average verification latency. Figures 12 shows the trend of throughput-latency values that we observed over these experimental runs for YCSB-A workload with 50% reads and 50% updates, with zipf parameter $\theta = 0.9$.

FastVer achieves high throughput exceeding 50M ops/sec for all the database sizes we evaluated. However, as the size of the database increases, high throughputs come with increasingly higher verification delays. This happens because the best throughput numbers are achieved by batching a large number of operations between verification scans which translates to higher verification costs. Even for very large databases, the parameters provided by FastVer allow us to tradeoff throughput for lower latency. For a database of size 128M records, we achieve a subsecond verification latency with a throughput of 10M ops/sec. In contrast, the best throughput reported in Concerto [3] is around 3M ops/sec, but incurs a verification latency exceeding 10s of seconds for a modest database of 10M records. FastVer achieves similar performance for other workload distributions; this happens because deferred verification turns a pure read into a read-modify-write operation (to update timestamps) and a similar observation has been made in [3]. In the extended version, we include experiments with a uniform distribution (zipf parameter $\theta = 0$) and our experiments validate the positive impact of skew. In general, the throughput of FastVer at skew $\theta = 0.9$ is about 30% higher than that at $\theta = 0$.

Figure 13a shows the throughput latency curves for YCSB-E scan-based workload, using scan length parameter 100 and a database of 64M records. FastVer is not a transactional system (nor is Faster), so the execution of scan queries is not guaranteed to be atomic. Figure 13a reports the per-key operation rate, so the per-scan operation rate is roughly 50 times slower since a scan contains

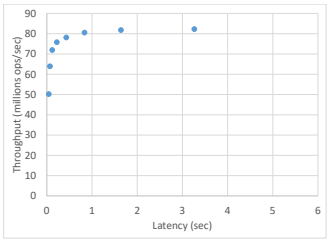


Figure 8: 2M records

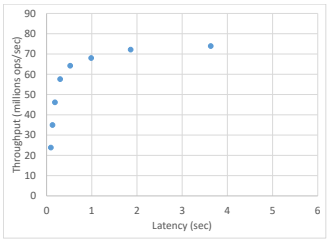


Figure 9: 8M records

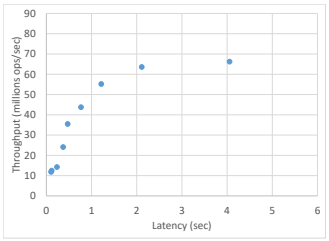


Figure 10: 32M records

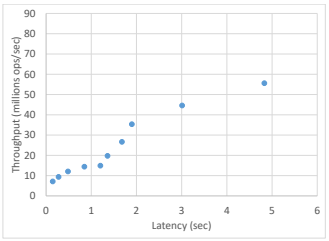
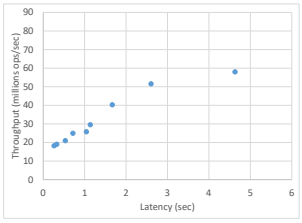
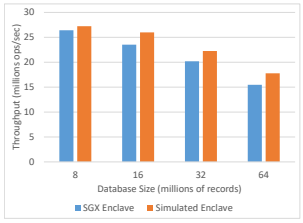


Figure 11: 128M records

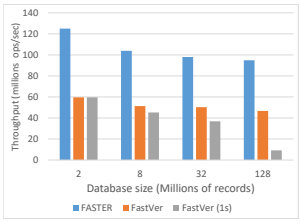
Figure 12: Throughput vs Latency curves for FastVer for workload of 50 % reads and 50 % updates with keys picked using a zipfian distribution with $\theta = 0.9$



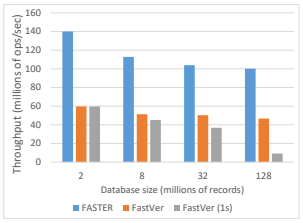
(a) Throughput vs Latency curve for FastVer for YCSB-E workload for a DB of 64M records, zipf $\theta = 0.9$, and scan length 100



(b) Throughput (at 1 sec latency) of FastVer with SGX enclaves and simulated enclaves for a workload of 50% reads and 8 worker threads.



(c) Comparing the throughput of Faster baseline with FastVer for a workload of 50% reads for different database sizes and 32 worker threads.



(d) Comparing the throughput of Faster baseline with FastVerfor a readonly workload for different database sizes and 32 worker threads.

Figure 13

50 keys on average. We did not notice a significant impact of scans on performance, i.e., we empirically observed the per-operation rate of YCSB-A to be very similar. One subtle impact is that the throughput-latency curve is *flatter* at lower latencies: this is the region where merkle records are protected using deferred verification and the impact of a cached merkle record is higher in a scan-based workload than for a non-scan one.

8.2 Experiment: SGX Enclaves

Figure 13b shows the performance comparison of FastVer with SGX enclaves and with simulated enclaves, 8 worker threads, for YCSB-A benchmark where the keys were picked using a uniform distribution. We report throughput numbers for a verification latency of 1 second. At all configurations, the performance of FastVer using SGX enclaves was about 90% of that of FastVer with simulated enclaves, and this trend remains true in other experimental settings. We believe the slight slowdown of performance with real enclaves arise from effects not modeled in our enclave simulation such as an increased overhead of memory accesses with true enclaves.

8.3 Comparison with Faster baseline

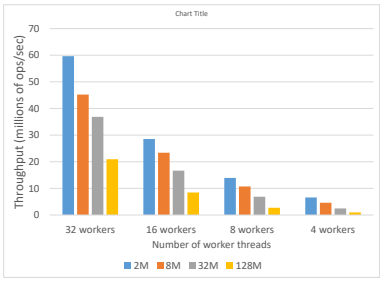
Figures 13c and 13d show the performance comparison between Faster and FastVer for a workload of 50% reads and 100% reads, respectively. The figure shows two throughput numbers for FastVer: the best throughput with no constraints on verification latency and the throughput achieved at sub-second verification latency. Overall, FastVer is competitive (less than x2 performance penalty) with Faster at all the database sizes that we can tolerate verification latencies of 10s of seconds. However, for sub-second verification latencies, for 128M database size, the performance of FastVer can be up to 10 times slower than Faster.

8.4 Experiment: Scalability

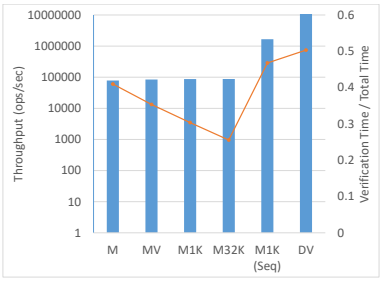
Figure 14a shows the performance of FastVer as we vary the number of worker threads for different database sizes and a workload of 50% reads. Overall, FastVer scales well with the number of worker threads. There is a minor super-linear effect (e.g., the performance for 32 workers is slightly more than double that of 16 workers) that happens due to partitioning of the Merkle tree as indicated in Section 6. The other parts of our verification mechanism, including deferred verification, is embarrassingly parallel and is able to leverage any available parallelism.

8.5 Experiment: Performance Drill-Down

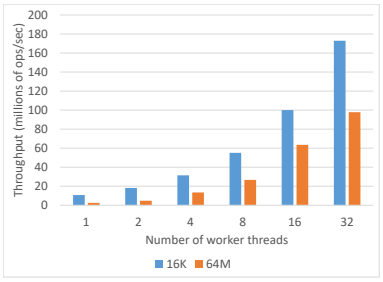
In this section, we present results of various "micro"-data integrity experiments designed to drill down and understand the performance characteristics of FastVer. Each of these experiments stores a large number of records in an array, performs key-value operations, and gets the verifier to check data integrity. By storing the records in an array, not Faster, we remove any effect of Faster code in the results. The operations are loosely modeled on YCSB-A



(a) Performance of FastVer with varying number of worker threads



(b) Single-threaded throughput for micro-experiments of Section 8.5



(c) Multi-threaded throughput for experiments of Section 8.5

Figure 14

(50% updates) and records are picked at random unless otherwise mentioned.

Single Threaded Performance: Figure 14b shows the single-threaded throughput performance of this setup for five different verification techniques for a dataset of size 64M records:

- *(M)* Plain Merkle tree without verifier caching.
- *(M1K)* Merkle tree with verifier caching (Section 4.3) with a cache size of 1K records.
- *(M32K)* Same as M1K, but with a cache of size 32K records.
- *(MV)* Merkle tree with caching of size 32K, but for an *put* operation we propagate Merkle hash updates all the way to root. This setup is designed to model the caching technique suggested in VeritasDB [29].
- *(M1K Seq)* Merkle tree with a cache of size 1K, but the workload is now sequential instead of random.
- *(DV)* Deferred verification.

Figure 14b also shows on the secondary axis, the fraction of the time an experiment run spent within the verifier. We used a simulated enclave for all the verifiers.

The performance of all the non-sequential Merkle variants is clustered around a 100K operations/sec. While caching does improve performance the improvements are small and not registered in Figure 14b which plots throughput on a log-scale. Caching improves verification costs as shown by the decreasing fraction of time spent within the verifier when we go from plain Merkle to Merkle with a cache of size 32K entries.

If the workload is sequential, verifier caching provides a significant performance improvement and the throughput increases by an order of magnitude to around 1M operations/sec. This performance improvement is the basis for the optimization of Section 6.3 where we sort the records prior to Merkle hash updates. Finally, deferred verification has a throughput of over 10M ops/sec. One of the reasons for this performance difference is that AES block encryptions (hardware optimized using AESNI instructions) used for multiset hashing are more efficient than Blake3 cryptographic hashing required for Merkle. Our profiling suggests we are able to perform multiset hashing at a rate of 3.2GB/sec and cryptographic Blake3 hashing at a rate of around 400MB/sec.

Multithreaded Performance: FastVer inherits many of the bottlenecks of Faster (or any key-value store). Record accesses could result in cache misses and incur memory access latencies. Since FastVer inlines additional information required for verification within a record the random memory access pattern of FastVer is nearly identical to that of Faster (FastVer needs additional random accesses to merkle tree records). FastVer does incur additional sequential memory accesses to read- and write verification logs and significant amounts of extra computation to perform hashing (quantified in the single-threaded experiments above).

Figure 14c seeks to shed light on how the computation costs of our hybrid approach interact with memory access costs by studying multithreaded performance for a small database of size 16K records (which fits in the L3 cache) and a large database of 64M records (which does not). Again, we implement the database using an array to remove effects of Faster code and use a YCSB-A style benchmark with the same fraction of reads and updates with records picked uniformly at random. While there is no data contention most record accesses should result in an L1/L2 cache miss. We use a batch size of 4M operations/worker at which setting almost all the records are protected using deferred verification (for that reason, the performance of one worker in Figure 14c is very similar to that of DV in Figure 14b). For a database of size 16K records the throughput performance scales linearly with an increase of about 75% everytime we double the number of workers; the 25% loss in performance reflects the effect of L1/L2 cache misses. The performance trends are similar for the larger database size and the performance gap from the smaller database size reflects the effect of L3 cache misses and memory access latencies.

9 CONCLUSIONS

In this paper, we present the design of FastVer, a high-performance key-value store with data integrity guarantees. The system leverages server-side enclaves and uses novel algorithms that combine the advantages of both Merkle trees and deferred memory verification. This approach achieves throughput which is in the ballpark of the highly optimized FASTER key-value store and which is orders of magnitude more than any state-of-the-art integrity solutions. We believe this work can serve as an important building block in making data integrity solutions a commodity.

REFERENCES

[1] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, et al. 2020. Azure SQL Database Always Encrypted. In *SIGMOD*. 1511–1525.

[2] Arvind Arasu, Ken Eguro, Manas Joglekar, et al. 2015. Transaction processing on confidential data using cipherbase. In *ICDE*. 435–446.

[3] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.

[4] Sumeet Bajaj and Radu Sion. 2014. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *IEEE Trans. Knowl. Data Eng.* 26, 3 (2014), 752–765.

[5] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3 (2015), 8:1–8:26.

[6] Blake3 cryptographic hash function 2018. https://github.com/BLAKE3-team/BLAKE3.

[7] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. 1991. Checking the correctness of memories. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*.

[8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: An Embedded Concurrent Key-Value Store for State Management. *Proc. VLDB Endow.* 11, 12 (2018), 1930–1933.

[9] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. 2019. SEEMless: Secure End-to-End Encrypted Messaging with Less Trust. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1639–1656. https://doi.org/10.1145/3319535.3363202

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, et al. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. 143–154.

[11] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. 2009. How Efficient Can Memory Checking Be?. In *Theory of Cryptography Conference (TCC)*.

[12] FASTER source code 2018. https://github.com/microsoft/FASTER.

[13] Intel AES NI Sample library 2013. https://software.intel.com/content/www/us/en/develop/articles/advanced-encryption-standard-aes-crypto-performance-analysis-project.html.

[14] Rohit Jain and Sunil Prabhakar. 2013. Trustworthy data from untrusted databases. In *ICDE*. 529–540.

[15] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[16] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (Sept. 1979).

[17] B. Laurie and E Kasper. 2013. Revocation transparency. www.links.org/files/RevocationTransparency.pdf.

[18] Feifei Li, Marios Hadjieleftheriou, George Kollios, et al. 2006. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*. 121–132.

[19] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[20] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. 2010. Depot: Cloud storage with minimal trust. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[21] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Proceedings of the International Cryptology Conference (CRYPTO)*.

[22] Donald R. Morrison. 1968. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (1968), 514–534.

[23] Alina Oprea and Kevin D. Bowers. 2009. Authentic Time-Stamps for Archival Storage. In *Computer Security – ESORICS 2009*, Michael Backes and Peng Ning (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–151.

[24] Bryan Parno, Jacob R. Lorch, John R. Douceur, et al. 2011. Memoir: Practical State Continuity for Protected Modules. In *IEEE S&P*. 379–394.

[25] Guna Prasaad, Badrish Chandramouli, and Donald Kossmann. 2019. Concurrent Prefix Recovery: Performing CPR on a Database. In *SIGMOD*. 687–704.

[26] Redis 2021. Redis in-memory data structure store. https://redis.io.

[27] Srinath Setty, Sebastian Angel, and Jonathan Lee. 2020. Verifiable State Machines: Proofs That Untrusted Services Operate Correctly. *ACM SIGOPS Operating Systems Review* 54, 1 (Aug. 2020), 40–46.

[28] Srinath T. V. Setty, Sebastian Angel, Trinabh Gupta, et al. 2018. Proving the correct execution of concurrent services in zero-knowledge. In *OSDI*. 339–356.

[29] Rohit Sinha and Mihai Christodorescu. 2018. VeritasDB: High Throughput Key-Value Store with Integrity. *IACR Cryptol. ePrint Arch.* 2018 (2018), 251.

[30] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

[31] Wenting Zheng, Ankur Dave, Jethro G. Beekman, et al. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*. 283–298.