Concerto: A High Concurrency Key-Value Store with Integrity

Arvind Arasu[1] Ken Eguro[1] Raghav Kaushik[1] Donald Kossmann[1]
Pingfan Meng[2] Vineet Pandey[2] Ravi Ramamurthy[1]

[1]Microsoft Research
Redmond, WA, USA
{arvinda, eguro, skaushi, donaldk, ravirama}@microsoft.com

[2]University of California
San Diego, CA, USA
{pmeng, vipandey}@ucsd.edu

ABSTRACT

Verifying the integrity of outsourced data is a classic, well-studied problem. However current techniques have fundamental performance and concurrency limitations for update-heavy workloads. In this paper, we investigate the potential advantages of *deferred* and batched verification rather than the per-operation verification used in prior work. We present *Concerto*, a comprehensive key-value store designed around this idea. Using Concerto, we argue that deferred verification preserves the utility of online verification and improves concurrency resulting in orders-of-magnitude performance improvement. On standard benchmarks, the performance of Concerto is within a factor of two when compared to state-of-the-art key-value stores without integrity.

1. INTRODUCTION

Storing data in the cloud involves crossing organizational trust boundaries. The data owner (clients) might not trust the cloud provider to handle its data correctly—a rogue administrator or a hacker could tamper with critical client data. Similarly, the cloud provider might be concerned about spurious claims of data corruption by the client. This lack of trust motivates the problem of designing *verification* mechanisms for *data integrity*. The goal is to design a protocol for data updates and retrieval that enables clients to *verify* (and the cloud provider to *prove*) that operations were implemented correctly over valid data.

There is a rich body of work on data integrity verification [8, 13, 22, 28, 30, 31, 36]. Most of this work relies on *cryptographic hashing* [16] and *Merkle* trees [25]. The main idea in this work is to store a hash of the outsourced data at a trusted location. Whenever the outsourced data is accessed, its integrity is verified by computing a new hash and comparing it with the hash stored at the trusted location; if the hash function is *collision resistant*, it is computationally difficult for an attacker to alter the data without changing its hash. Merkle trees enable locally verifiable and incrementally updatable hashing, meaning that only part of the data is required to verify integrity or update hashes. This property is achieved by a recursive application of a standard cryptographic hash function as we discuss in Section 3.

However, the Merkle-tree approach has fundamental limitations when faced with concurrent, update-heavy workloads. This is because every operation reads, and every update operation updates, the root of the Merkle tree. This introduces read-write and write-write conflicts at the root, which limits concurrency and reduces performance. Furthermore, the verification step is computationally intensive, requiring a logarithmic number of hash computations for each operation, making it a potential bottleneck. Since this step needs to access and possibly update the root hash value stored in the trusted location (another read-write conflict), it is not amenable to easy parallelization.

These limitations are magnified by modern hardware, which offers rich parallelism capabilities. State-of-the-art data stores [21, 38] leverage these capabilities to achieve millions of update-heavy operations per second on a single server node. On the other hand, using existing techniques for data integrity, our experiments and results from recent papers [13] indicate performance on the order of thousands of operations per second—a severe three orders-of-magnitude penalty to implement integrity.

In this paper we observe that the use of Merkle trees is closely tied to *online* verification, the verification model used in most prior work. In this model, verification occurs on an operation-by-operation basis. That is, when an operation is evaluated, the integrity of any data touched by that operation is immediately verified. There is evidence that some of the limitations of Merkle trees are fundamental to online verification. For example, Dwork et al. [9] show that any online verification technique requires logarithmic verification overhead, assuming bounded trusted state.

Motivated by this fact, in this paper we investigate an alternate verification model, moving away from operation-by-operation verification and exploring the benefits of batching verification across multiple operations. We present a comprehensive key-value store, *Concerto*, designed from the ground up around batched verification. Although batching changes the verification model somewhat since verification is now *deferred* to a later time to enable batching, we argue that the primary value of verification remains undiminished by deferring. At the same time, we are able to nearly close the three orders-of-magnitude performance gap. The performance of Concerto, including the cost of deferred verification, is within a factor of two of state-of-the-art key value stores without integrity. Although Concerto only supports simple key-value functionality currently, we believe it provides the building blocks for a more general integrity platform, which is an active research effort.

SIGMOD'17, May 14–19, 2017, Chicago, IL, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05. . . $15.00
DOI: http://dx.doi.org/10.1145/3035918.3064030

1.1 Deferred Verification

In deferred verification, integrity verification is decoupled from client data operations. If data is tampered with, this alteration is not detected immediately when it is accessed, but in a separate batched verification step that detects integrity violations of all data accessed since the last verification step. We emphasize that integrity violations never go undetected, but the discovery is merely delayed.

The delayed detection of integrity violation has two implications: First, deferred verification is not suitable for applications (e.g., ATMs dispensing cash) that involve operations that cannot be undone and cannot tolerate a verification delay. Second, deferred verification could lead to wasted work. Data operations that happened after an integrity violation but before it was detected need to be undone and possibly redone. However, as we argue in Section C, online verification does not necessarily preclude wasted work or make recovery from integrity violations any simpler.

Online verification is analogous to *pessimistic concurrency control* and detects integrity violations (conflicts) in data before performing operations over it. Deferred verification is similar to *optimistic concurrency control* and detects integrity violations (conflicts) at a later point in time. Optimistic concurrency control is not suitable for some applications and subject to wasted work due to aborts. Nevertheless, optimistic concurrency control mechanisms such as snapshot isolation have become the de-facto standard in the database industry. For data integrity, a deferred (optimistic) approach is even more attractive as integrity violations are rarer than conflicts (justifying optimism) and the cost of wasted work is much lower because the execution of operations can be made orders-of-magnitude cheaper than with online verification as we show in this paper.

Integrity verification plays the role of *deterrence* for malicious activity. The value of deterrence does not diminish if the detection is deferred. The data owner still obtains a formal proof of any incorrect operation by a cloud service, and the cloud provider still obtains a formal proof for correct operation. Besides, in terms of real-world utility we also empirically establish in Section 7 that the delay necessary to realize the performance benefits is relatively modest: on the order of seconds to less than a minute.

1.2 A Design Centered around Concurrency

Concerto's design exploits the additional flexibility provided by deferred verification to avoid concurrency bottlenecks like those in Merkle tree-based approaches. A central component of our design is reducing the problem of verifying key-value integrity to that of verifying memory integrity. For memory integrity, the deferred verification formulation enables us to use the *offline memory checking* algorithm proposed by Blum et al [5]. This algorithm is efficient and incurs a constant time overhead per operation. We extend the Blum approach by presenting techniques that enable parallelization of verification while avoiding hotspots such as the root node of a Merkle tree.

A second key component of our design is the separation of data integrity from indexing. In previous approaches, the same structure was used to find (or show the absence of) data and enable integrity. In Concerto, indexing is purely a performance accelerator and is unrelated to the verification of integrity. This design allows us to reuse state-of-the-art indexing techniques and inherit their concurrency optimizations for modern hardware.

Lastly, like much recent work [2, 39], Concerto relies on a server-based *trusted platform* for verification and to store trusted state (Section 2). A trusted platform is a shielded execution environment backed by *secure hardware* on an otherwise untrusted server. By design, a server administrator or attacker cannot tamper with the processing and state within the trusted platform. While we can design integrity solutions without it, a server-based trusted platform simplifies client-server protocols and improves integrity guarantees and performance [23]. This paper focuses mainly on the main memory setting where the overhead of integrity is the largest. A detailed study of how our techniques apply to an external memory setting is future work.

2. OVERVIEW AND ARCHITECTURE

2.1 Key-Value Store Functionality

Concerto provides the standard data model and functionality of an ordered key-value store. Keys are drawn from an ordered domain and values from an arbitrary domain. Concerto supports the standard $get(k)$, $put(k, v)$, $insert(k, v)$, and $delete(k)$ operations. Keys are unique within an instance and $insert(k, v)$ fails if key k exists in the database. With slight extensions, Concerto can support range scans, multi-key stores (where keys are not unique), and unordered keys (hash tables).

Concerto uses a client-server architecture as shown in Figure 1. Any number of client processes connect and issue concurrent operations to a Concerto server. The Concerto server serializes the operations and responds with the results. The result of an operation consists of a success/failure status bit. A successful $get(k)$ operation also returns the value associated with key k.

2.2 Trusted Platforms

Concerto relies on a server-based *trusted platform* for its integrity guarantees. The trusted platform enables isolated code execution and provides shielded state. An adversary with administrative privileges over the Concerto server cannot influence the execution or output of the *trusted code* running within the platform, for a given input, or tamper with its internal state. Trusted platforms can be implemented using specialized hardware (FPGAs [10], secure co-processors [2]) installed on the server machine or a special mode of the processor (e.g., Intel SGX enclaves [24]).

A trusted platform typically also provides an attestation mechanism by which a remote client can verify the authenticity of the code executing within the platform. Attestation prevents attacks where the adversary installs malicious code within the trusted platform. Upon successful execution, the attestation protocol also provides the remote client with a public key to bootstrap secure communication with the trusted code. The details of attestation and secure communication of a public key are specific to the platform, and we refer the reader to these references [24, 39].

2.3 Integrity Guarantee

Concerto provides the guarantee that any incorrect key-value store functionality is detected by a verification procedure. With concurrency, this guarantee means that after verification, the operations satisfy *sequential consistency* [20]: the result of each completed client operation is consistent with a sequential execution of all completed operations, and the operations from each client appear in that sequence in the order they were issued by the client. Concerto further ensures that only legitimate client operations are part of this sequence, without repetition. This means that an adversary cannot introduce spurious operations or replay legitimate ones. Our sequential consistency based integrity guarantee implies result *correctness, completeness, and freshness*, properties that have been used to formalize data integrity in prior work [22].

Our integrity guarantee relies only on the properties of the trusted platform listed earlier, the correctness of the code executing within it (trusted code), and the security of standard cryptographic primitives. In particular, this guarantee holds even if the Concerto server is com-

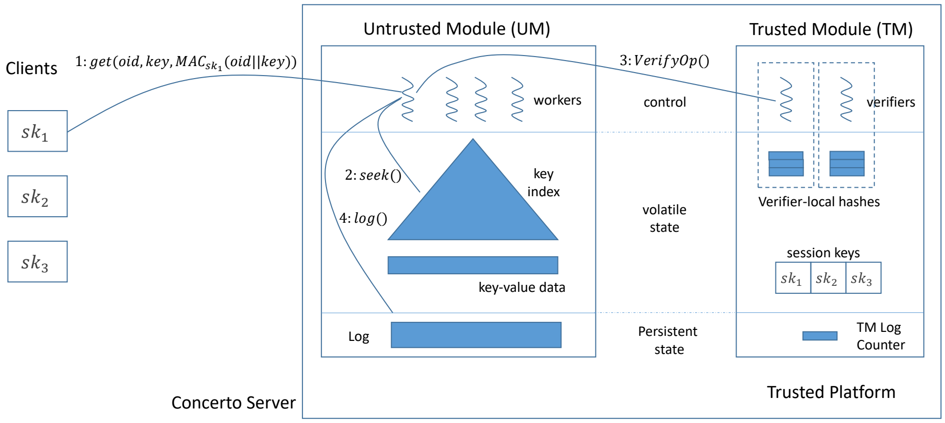


Figure 1: *Concerto Architecture*

promised by an adversary who seeks to disrupt the guarantee. The adversary has full control over the Concerto server: the adversary can change untrusted code and data outside the trusted platform, alter client and trusted code communication, and reboot the trusted platform.

Deferred Integrity Verification

When a client invokes a data operation, the Concerto server returns, along with the result of the operation, an attestation from trusted code running within the trusted platform. This attestation represents a *guarantee* that if the result of the operation is incorrect, it would be detected by a subsequent integrity verification step. Informally, the trusted code maintains internal hashes to support this guarantee, the details of which form the technical core of this paper.

To perform deferred verification, a client invokes the $DefVerify()$ method exposed for this purpose. This method returns a boolean value: a *true* indicates that the trusted code has verified the integrity of all previous operations and a *false* indicates that some previous operation had an integrity violation. (We discuss recovery from $DefVerify()$ failures in Section C.) $DefVerify()$ is a batched operation and performs verification of all operations (for all client processes) since the previous $DefVerify()$ call. As noted in Section 1, many of our performance advantages stem from being able to leverage batched verification. Deferred verification is non-quiescent meaning Concerto processes regular data operations while executing $DefVerify()$.

As the above interface suggests, actual integrity verification is performed by the trusted code within the server, and the client merely learns the result of the verification. Since the client-server communication is over an insecure channel, we protect responses using *message authentication codes (MAC)* originating from the trusted code as described in Section 2.4.1.

Non-guarantees

Concerto does not protect against denial-of-service attacks. For example, an adversary can drop all client operations. The adversary can also induce non-recoverable failures by destroying the database and the logs or the persistent state within the trusted platform. The adversary could also affect the progress and serialization order of operations by delaying the processing of selected operations. These non-guarantees stem from the power of the adversary, not from any design choices of Concerto; i.e. they exist in all systems that verify integrity.

2.4 Architecture

Figure 1 shows the overall architecture of Concerto. Concerto currently runs on a single server machine. The server consists of an *untrusted platform* consisting of conventional hardware (e.g. multiple CPU cores, large memory, and persistent SSDs or disk), an OS stack, and a *trusted platform* with the security properties discussed in Section 2.2. Our design is agnostic to the choice of trusted platform: we assume it supports parallel execution, limited volatile memory, and a few words of nonvolatile memory; all trusted platforms that we know of provide these features. With sharding and simple client side checks, single node Concerto can be used as a building block to derive a multi-node key-value store with the same integrity guarantees, but we do not discuss this setting in this paper.

Figure 1 shows the state maintained by Concerto across the trusted and untrusted platforms. Concerto stores, in the untrusted system, the key-value database, an index to efficiently access specific records in the database, and a persistent log for recovery after a hardware/software failure or an integrity violation. The trusted platform contains a volatile memory used to store a small number of hashes for integrity checking and a single persistent memory used to store a counter for recovery (Section C).

The Concerto server runs a collection of *worker* threads in the untrusted platform and a collection of *verifier* threads in the trusted platform (we use the term *thread* to mean a parallel execution unit, not necessarily an OS thread). Worker threads handle client operations. For each operation, a worker thread traverses the index, looks up or updates key value data, and logs changes. In addition, it invokes a remote *verification* procedure, $VerifyOp()$, within the trusted platform. A verifier thread handles the $VerifyOp()$ call, where it checks the authenticity of the operation. If the operation is valid, the verifier thread updates internal hashes (to be used by a subsequent deferred verification step) and generates an attestation for the result. As noted earlier, this attestation serves as a guarantee that the result will be verified by the next $DefVerify()$ call. Workers and verifiers similarly coordinate to implement $DefVerify()$ calls.

In the rest of this paper, we use the term *untrusted module (UM)* to refer to Concerto components in in the untrusted platform (e.g. workers, key-value data) and *trusted module (TM)* to refer to those in the trusted platform (e.g. verifiers, hashes).

2.4.1 *Client Server protocol*

Before submitting any operations, a client process establishes a *connection* with the Concerto server. As part of the connection setup process, the client verifies the authenticity of the verification code within the trusted platform (Section 2.2), gets access to its public key, and performs a key exchange to derive a symmetric *session* key. The trusted platform also verifies the authenticity of the client

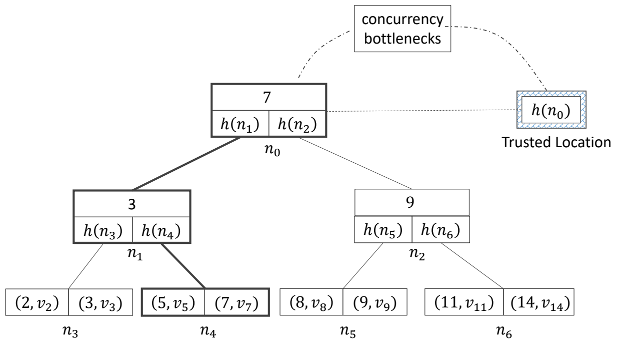


Figure 2: *Merkle (B-)Tree and concurrency bottlenecks. The hash of a node is stored at the parent and the hash of the root at a trusted location.*

process. We elide these details since they are orthogonal to data integrity; e.g., they could rely on PKI [16].

The session key is used to protect communication integrity. That is, when a client communicates an operation, all the parameters of the operation (keys, values) are protected by a MAC generated using the session key. The verifier thread handling this operation checks the MAC, so an adversary cannot introduce spurious operations into the system. As part of this, all operations carry an *operation id (oid)* parameter, which is required to be globally unique (e.g. monotonically increasing). The verifier thread checks the uniqueness of the oid to prevent replay attacks. The result attestation for an operation consists of a MAC, generated using the session key, that combines its oid and the result. The MAC ensures that the result as recorded by the verifier is not altered en-route to the client. We note that the same protocol is used both for regular data operations and for $DefVerify()$ calls, so the boolean result of $DefVerify()$ is protected by a MAC as well.

3. BASELINE USING MERKLE TREES

In this section, we present an overview of a Merkle-tree based solution for key-value data integrity. To simplify the comparison between the two approaches, we assume a similar server-based UM/TM architecture as for Concerto (Section 2). This Merkle-tree solution is also our experimental baseline implementation in Section 7. A server-based TM makes for a fairer comparison than the client-side verification used in most prior work. We specifically discuss a variant called *MB-tree* [22], but our arguments generalize to other variants as well.

An MB-tree is like a regular B-tree index over key-value data, except that in addition to every pointer in a non-leaf node we also store a cryptographic hash (e.g., SHA-256) of the node that is pointed to. For example, in Figure 2, node n_1 stores a pointer to node n_3 and the hash $h(n_3)$. The hash of the root node is stored within the TM. The MB-tree itself is stored in the untrusted system.

To process an operation involving a key k, the UM traverses the MB-tree using k just like in a regular B-tree. It then sends the root-to-leaf path to the TM. The TM checks, for every link in the path, if the hash stored with the link is equal to the hash of the node pointed to. If all hash checks pass, the collision resistance property of the hash function and the shielded value of the root hash in the TM imply the integrity of every node in the path. The TM then uses the leaf node in the path to determine the result of the operation (e.g., the return value of a $get()$ or the success of an $insert()$) and attests the result for the client. In this system, the root-to-leaf path essentially serves as a "proof" to the TM, validating the result.

In Figure 2, for the operation $get(5)$, the UM sends the TM the highlighted path with nodes n_0, n_1, and n_4. The TM checks if the hash of node n_0 matches with the root hash stored within the TM; if it does, this check establishes the integrity of node n_0 and therefore also the integrity of the hash $h(n_1)$ stored within n_0. The hash $h(n_1)$ stored in n_0 is then used to check the integrity of node n_1, and so on. Finally, the integrity of node n_4 establishes that the correct result for $get(5)$ is the value v_5.

For update operations, the TM recomputes the hashes along the path to maintain the MB-tree hash invariants and returns the updated nodes to the UM. For inserts and deletes, this might mean adding or deleting new nodes, so the TM code needs to handle the complexity of B-tree structure changes [14].

We note that authentication using a Merkle tree reveals information about data beyond what is being authenticated. For example, while verifying $get(5)$, the verifier learns about the value 7 and the existence of values lesser than 5 and larger than 7. While this is not an issue for our problem, there exist applications such as document publishing with access control restrictions where this is unacceptable. There exists an interesting line of work [18, 6, 17] that avoids such leakage, as discussed further in Section D.

3.1 Limitations

Hierarchical hashing and concurrency: As noted in Section 1, the hierarchical hash aggregation of Merkle trees is fundamentally at odds with concurrency. This arrangement introduces a read-write conflict at the root node between every update and any other operation. For example, in Figure 2, $get(5)$ needs to read the entire root node n_0 to compute its hash, while $put(14, v'_{14})$ needs to update the hash $h(n_2)$ stored in n_0. These operations touch unrelated keys and would rarely interfere with one-another in a regular key-value store.

Computation and communication overheads: Every operation involves shipping a root-to-leaf path and computing a hash for each node in the path in the TM, an overhead that is logarithmic in the database size. As shown in [9], this overhead is a fundamental lower limit for online verification. Besides, these costs do not amortize well even if we *batch* multiple operations together since the proof paths for different operations do not overlap much unless their keys happen to be "close-by." Thus, in general, they cannot benefit much from shared TM computation or communication. In Section D, we implement a batched version of Merkle tree and empirically validate this intuition.

Distributing TM state: Due to the opaque nature of the hash function, we do not know of any mechanism to distribute the TM state (i.e. the root hash) to parallelize verification. While we could shard the key-value data into multiple partitions, build a separate Merkle tree per-partition, and have a different verifier thread per partition, this approach does not perform well for skewed workloads where most of the operations touch a small number of keys.

4. BASIC DESIGN

This section presents a simplified variant of Concerto called *Basic Concerto*. Basic Concerto supports no concurrency: it processes client operations serially with a single worker and a single verifier. Our goal here is to introduce the central ideas of Concerto and argue their correctness in a simple setting. Section 5 enhances Basic Concerto with optimizations including support for concurrency.

4.1 Intuition

To present the intuition behind Concerto design, we revisit the high-level design of MB-tree: MB-tree stores the current key-value database instance in the untrusted storage. For any operation, the

UM sends a proof to the TM to verify its result. We can view the proof as consisting of two parts: (a) partial state from the current database that establishes the presence or absence of a key and the current value associated with a key; and (b) information to verify the integrity of the state in sent in (a). For example, for operation $get(6)$ in Figure 2, part (a) consists of the leaf node n_4. MB-tree stores the current records in the leaf nodes sorted by their keys, so node n_4 establishes that key 6 is not present; part (b) consists of the chain of hashes from n_4 to the root hash stored within the TM.

Our design uses a different strategy for constructing proof parts (a) and (b). We modify database storage to store along with each key-value pair, the *next key* in the current instance. With this change, a single record is sufficient for part (a), i.e., to prove the presence or absence of a key and establish the current value for a key. For the database of Figure 2, we store as one of the records, the triple $\langle 5, 7, v_5 \rangle$, with 7 being the next key after 5; this triple establishes the absence of key 6 for the $get(6)$ operation above. For part (b), the UM needs to prove the integrity of a (key, next-key, value) triple. To do this, we store the current triples in a specially identified memory and use memory integrity checking algorithms. Our proofs no longer rely on storing the records sorted, so the triples can be stored in arbitrary locations of this memory.

This proof construction has several advantages over the MB-tree one: (1) The proofs of operations touching different keys are distinct records (triples) and enable better concurrency since they do not overlap. (2) Our proofs require integrity for individual records, not sorted sequences. This change removes indexing from the purview of integrity; we use indexes purely for performance, and any off-the-shelf index works. (3) With our deferred verification formulation, we are able to adapt more efficient offline memory checking algorithm of Blum et al. [5].

4.2 Design Assuming Verified Memory

We first present our design assuming an abstraction of *verified memory*, that hides all memory integrity checking details.

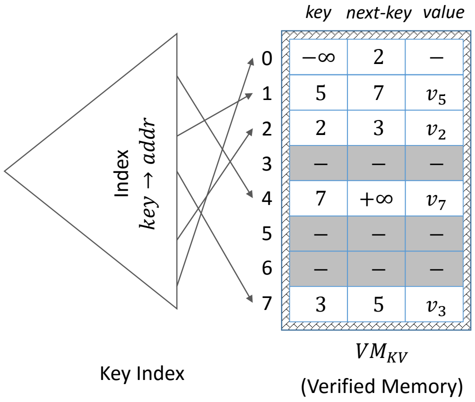
Verified memory is a collection of addressed memory locations called *vcells* stored in untrusted memory and designed to provide a verifiable read-write consistent view to the TM. Read-write consistent view means that the contents of a vcell that the TM sees on a read correspond to the most recent write by the TM to that location; otherwise, the TM detects an integrity violation. A strawman implementation of verified memory would be to keep within the TM for each vcell, a cryptographic hash of its contents; during a read, the TM checks the hash of the vcell against the stored hash and, during a write, updates the stored hash. We discuss a better implementation of verified memory in Section 4.3.

4.2.1 Concerto State

As outlined in Section 4.1, Concerto stores the current key-value database as (key, nextkey, value) triples in a verified memory instance called VM_{KV}. Figure 3 illustrates this storage for a sample instance. Formally, let D denote the current key-value database instance of size $n = |D|$. Let ∞ and $-\infty$ denote special values of the key domain greater than and lesser than any regular key, respectively. For any key $k < \infty$, let $next(k, D)$ denote the smallest key in D greater than k, or ∞ if no such key exists. For example, $next(-\infty, D) = 2$ and $next(4, D) = 5$ for the sample database of Figure 3. Note that $next$ is defined for all keys $< \infty$, not just keys in D.

Each vcell in VM_{KV} stores a triple of the form $\langle key, nextkey, value \rangle$ along with a *status* bit indicating if the contents of the vcell are *valid* or not. Concerto encodes D as follows:

1. For each $\langle k, v \rangle \in D$, there is a valid vcell $\langle k, next(k, D), v \rangle$.



	key	next-key	value
0	$-\infty$	2	–
1	5	7	v_5
2	2	3	v_2
3	–	–	–
4	7	$+\infty$	v_7
5	–	–	–
6	–	–	–
7	3	5	v_3

Figure 3: *Concerto state for the instance with four key value pairs:* $\langle 5, v_5 \rangle$, $\langle 2, v_2 \rangle$, $\langle 3, v_3 \rangle$, $\langle 7, v_7 \rangle$. *The verified memory* VM_{KV} *has 8 vcells, five valid and 3 invalid (shown in grey).*

2. There is a valid vcell $\langle -\infty, next(-\infty, D), - \rangle$.

3. All other vcells are invalid.

(We use $-$ to indicate a *null* value.) The n keys of D partition the key domain into $n + 1$ intervals and there exists one valid vcell for each interval. As noted earlier, this encoding has the property that for any key k', we can establish the presence or absence of k' in D using a single vcell. For example, in Figure 3, we can establish that key 4 is absent using $VM_{KV}[7]$. More generally, for any key k', there exists a unique vcell $\langle k, next(k, D), v \rangle$ such that $k \leq k' < next(k, D)$ that can be used to determine if k' is present is D. We call this the *proof vcell*, and it plays an important role in Concerto's data processing.

Initially, VM_{KV} contains a single valid vcell $\langle -\infty, \infty, - \rangle$ encoding an empty database. With data updates, the TM makes changes to VM_{KV} to maintain the invariant that VM_{KV} encodes the current database instance.

Concerto also maintains in untrusted memory an ordered index (called *key index*) mapping each key in D and the special key $-\infty$ to the vcell storing the key; e.g., in Figure 3, the key index maps key 3 to vcell 7. Standard search tree indexes can be tweaked to return, given a key k', the largest indexed key $\leq k'$; with our encoding, this returns precisely the proof vcell of k' described above. The key index is required only for performance, and the TM does not know of its existence. Finally, Concerto keeps track of the unused (invalid) vcells in a separate data structure in untrusted memory. We temporarily assume that the size of VM_{KV} is fixed and sufficiently large to store the database at all times; we present techniques for growing and shrinking verified memory in Section A.2.

Concerto state invariants might not hold with adversarial tampering since we store this state in untrusted memory. Such invariant violations ultimately manifest as read-write consistency violations of VM_{KV} during key-value operations and are detected by the TM. We present the processing of key-value operations assuming the state invariants hold and defer a discussion of how TM detects tampering to Section 4.2.3.

4.2.2 Processing Key-Value Operations

Recall that every operation has as input parameters a unique operation id (oid), a key k_o, and an optional value v_o (for $put()$ and $insert()$); a MAC generated using the session key protects these parameters.

Operation	*vcells*	*Precondition*	*TM Updates*
$get(k_o)$	$\langle k, k^+, v\rangle$	$k \leq k_o < k^+$	unchanged
$put(k_o, v_o)$	$\langle k, k^+, v\rangle$	$k \leq k_o < k^+$	$\langle k, k^+, v_o\rangle$
$insert(k_o, v_o)$	$\langle k, k^+, v\rangle$ unused	$k \leq k_o < k^+$	$\langle k, k_o, v\rangle$ $\langle k_o, k^+, v_o\rangle$
$delete(k_o)$	$\langle k, k^+, v\rangle$ $\langle k^-, k, v^-\rangle$	$k \leq k_o < k^+$	unused $\langle k^-, k^+, v^-\rangle$

Table 1: *The proof and aux vcells sent to the TM for various operations, the conditions that they satisfy, and the new contents of these vcells if the operation is successful.*

Worker thread: The worker thread traverses the key index and identifies the proof vcell for key k_o. Since the proof vcell determines the presence or absence of k_o in the current database, it is sufficient to determine success/failure of the operation. For an insert or delete operation which changes the number of records (and therefore the number of intervals of the key domain), the worker identifies an additional vcell called an *aux* vcell. For these operations, the aux and proof vcells will be modified by the TM to ensure the encoding described above. For an insert, the aux vcell is an unused vcell, and for a delete, it is the previous vcell to the proof vcell with the property $aux.nextkey = proof.key$. The worker then invokes the TM $VerifyOp()$ remote call passing along the operation parameters, the proof vcell, and the optional aux vcell. Table 1 summarizes the vcells used for each operation. Upon completion, the $VerifyOp()$ (discussed below) returns a MAC attesting the result of the operation and possibly new contents for the proof and aux vcells. The worker thread applies these changes, updates the key index if necessary, and returns the result of the operation and the TM provided MAC to the client.

Verifier thread: For each $VerifyOp()$, the proof and the optional aux vcells provided as input are considered as TM reads. The verifier thread, therefore, checks the memory integrity, i.e., if the contents of these vcells correspond to most recent TM write to these addresses. (The UM sends for each vcell both its address and contents.) For our verified memory implementation of Section 4.3, integrity checking is deferred and handled by a separate verification step. For presentation simplicity, we temporarily ignore this aspect.

The verifier thread then: (1) checks if the MAC of the operation parameters matches the MAC sent by the client; (2) checks if the oid was not previously seen; and (3) checks if the proof and aux vcells satisfy the pre-conditions of Table 1. Checks (1) and (2) ensure that the operation is legitimate and not a repeat of a previous operation. Check (3) is required because the worker thread cannot be trusted to provide the correct vcells. For Basic Concerto, oid is a session-local counter that increases with every operation, so for the check (2) above, the verifier thread checks if the current oid is greater than the last seen oid; to enable this check, the TM tracks the last seen oid using internal state. If all the checks pass, the verifier thread determines if the operation can succeed. If the operation can succeed, it updates the proof and aux vcells as shown in Table 1 to maintain the invariant that VM_{KV} encodes the updated database. These updates are TM writes, and the verifier updates TM-internal state to help verify the validity of a subsequent read to these locations (Section 4.3)[1] Finally, it generates a MAC attestation for the concatenation of oid, operation success, and the return value (for $get()$) using the session key. It returns the updated vcells and the MAC attestation.

EXAMPLE 1. *Consider the sample database of Figure 3. For a* $get(4)$ *operation, the worker thread sends the proof vcell* $VM_{KV}[7]$ $= \langle 3, 5, v_3\rangle$ *along with the operation parameters. This vcell establishes to the verifier that key* 4 *is absent from the database, so the verifier generates a MAC for* $(oid \mid\mid fail)$.

For an $insert(9, v_9)$ *operation, the worker sends* $VM_{KV}[4] = \langle 7, \infty, v_7\rangle$ *as the proof vcell and the unused vcell* $VM_{KV}[3]$ *as the aux vcell (*$VM_{KV}[5]$ *or* $VM_{KV}[6]$ *could have been used as well). The proof vcell establishes that key* 9 *is absent, so the insert operation succeeds. The verifier generates a MAC for* $(oid \mid\mid success)$ *and updates* $VM_{KV}[4]$ *to* $\langle 7, 9, v_7\rangle$ *and* $VM_{KV}[3]$ *to* $\langle 9, \infty, v_9\rangle$. *The resulting state of* VM_{KV} *encodes the database with five key-value pairs* $\langle 5, v_5\rangle$, $\langle 2, v_2\rangle$, $\langle 3, v_3\rangle$, $\langle 7, v_7\rangle$, *and* $\langle 9, v_9\rangle$.

4.2.3 Correctness Argument

We now discuss how our design provides the integrity guarantees specified in Section 2.3. For simplicity, we assume temporarily that memory verification is *online*, so a TM read inconsistent with the previous write is detected at the time of the read. We later extend these arguments for deferred memory verification.

To formalize Concerto's integrity guarantees, we focus on the TM view of the processing. The TM sees a sequence of $VerifyOp()$ calls each corresponding to an operation. For some of these calls, the verification checks fail, and the TM returns an error code. For others, the checks pass and the TM returns a MAC attesting the operation result and some vcell updates (TM writes). We call an operation whose $VerifyOp()$ call passes all checks as *TM verified* and the attested result a *TM verified result.* Under normal processing, without adversarial tampering or failures, all client operations would be TM verified. The following theorem formalizes the integrity guarantees of Concerto:

THEOREM 1. *The sequence of TM verified operations and their TM verified results represents correct key-value functionality. Further, the sequence contains only legitimate client operations without repetition.*

Correct key-value functionality means that if we run the same operation sequence on a correct untampered key-value store, we will get the same result for every operation as the TM verified one. Since the client accepts only TM verified results, the integrity guarantee of Theorem 1 is consistent with our definition of integrity from Section 2.3.

To see how Concerto provides this guarantee in the presence of an adversary who controls the UM state and processing, we focus on the TM view of the system. At any given point, the TM has verified some sequence of operations H. Let $D_{TM}(H)$ denote the database instance corresponding to these operations; $D_{TM}(H)$ is an abstract concept, and the actual database instance stored in the UM may not be $D_{TM}(H)$ if there is tampering. Similarly, when processing H, the TM performs various vcell writes. Let $VM_{KV}^{TM}(H)$ denote the verified memory state implied by these writes. Again, $VM_{KV}^{TM}(H)$ is an abstract concept, and the actual state of VM_{KV} in the untrusted system after processing H may not correspond to $VM_{KV}^{TM}(H)$. The important observation is that if the UM provides an input vcell for the next operation that is inconsistent with $VM_{KV}^{TM}(H)$, the TM detects this inconsistency during memory verification. Using this observation, we can show using induction that at any point $VM_{KV}^{TM}(H)$ correctly encodes $D^{TM}(H)$. This is true at the beginning before any operations. By induction hypothesis and memory verification, the TM sees "correct" vcell contents for an operation; if the TM sees correct vcells contents, we can show, by considering each operation type, that the TM identifies the correct result and makes the correct changes to the vcells. The full version [1] contains a formalization of this argument.

[1] Our use of the terms *read* and *write* is non-standard in the sense that the TM does not pick the vcells to read. The UM picks the vcells that TM reads and applies the updates sent back by the TM.

4.2.4 Discussion

We highlight a few aspects of our verified memory based design:

Similarity to regular key-value store: Concerto is not very different from a traditional key-value store. A traditional key-value store has an index over the keys, and the key-value records are the stored in the leaf nodes of the index. Concerto uses a similar index with a level of indirection, with the leaf nodes pointing to the actual records encoded within verified memory. Most of the computational work in a key-value store occurs in the index traversal. For Concerto too, the bulk of work is the UM index traversal. The TM-verification while subtle involve $O(1)$ simple checks.

TM State: The key-value portion of the verification requires no TM state and therefore parallelizes easily. Memory verification, however, requires TM state and does not lend itself to straightforward parallelization (Section 5.2).

Deferred verification: We made no reference to $\mathit{DefVerify}()$ in this section: our deferred integrity guarantees ultimately arise due to deferred nature of memory verification, and we discuss the implementation of $\mathit{DefVerify}()$ in Section 4.3.

4.3 Verified Memory

Recall that verified memory is a collection of addressed cells stored in the UM but designed to provide a verification-based read-write consistent view to the TM. To implement verified memory, the TM needs to check if each read is consistent with the most recent TM write to that location. While Merkle trees can be used to implement verified memory [39], this approach inherits the fundamental limitations highlighted in Section 1.

Our idea is to use *deferred* memory verification, where checking read-write consistency does not happen during reads but in a separate step. Deferred memory verification allows us to use offline memory checking algorithm of Blum et al. [5] (hereafter, Blum algorithm) which we show can be adapted to yield a lightweight and parallelizable verification. We begin by presenting intuition behind Blum algorithm before formalizing deferred memory verification and specifying implementation details.

Blum Algorithm: Intuition

In a read-write consistent memory, every read corresponds to the last write to that location. Read-write consistency implies that the *read-set (RS)* defined as the set of $(\mathit{address}, \mathit{content})$ pairs seen by TM during reads closely tracks the *write-set (WS)* defined as the set of $(\mathit{address}, \mathit{content})$ pairs that TM writes. In fact, we can show that $RS \subseteq WS$ and $(WS - RS)$ contains exactly one entry for each location corresponding to its last write. Figure 13 illustrates this property for a memory comprising two vcells. So, in an offline step, if we scan the entire memory and update the RS with the $(\mathit{address}, \mathit{content})$ pairs seen, $RS = WS$ for a read-write consistent memory. Conversely, any read-write inconsistency results in $RS \neq WS$. Checking for this equality forms the intuitive basis for the Blum algorithm.

Hash function for sets: Storing read- and write-sets is expensive, and Blum algorithm relies on a specially constructed *collision resistant* hash function h over read-write sets; i.e., h has the property that for any two sets $S_1 = S_2$ implies $h(S_1) = h(S_2)$, and $S_1 \neq S_2$ implies $h(S_1) \neq h(S_2)$ with high probability. Further, hash function h is incremental, so $h(S \cup \{e\})$ is computable from $h(S)$ and e. Blum algorithm maintains $h(RS)$ and $h(WS)$ (inside TM) during memory operations. During the offline verification step, it scans the entire memory, updates $h(RS)$, and checks $h(RS) = h(WS)$. The Blum algorithm as presented fails if there are pure reads (the TM reads but does not update) or if writes are not unique. To fix this, Blum algorithm adds a *timestamp* field to memory cells; this field is updated using a TM-internal counter every time a vcell enters TM; this change eliminates pure reads and makes every write unique. This feature is also a limitation of the Blum algorithm since it implies that we need to log timestamp changes even for key-value read operations. Our implementation of logging includes optimizations to minimize the overhead. Ultimately, we show empirically that this additional logging does not significantly affect performance.

Operation	*Steps*
read(vcell)	$h_{rs} \leftarrow h_{rs} \oplus PRF(\mathit{vcell.addr}, \mathit{vcell.content})$
write(vcell)	$TC \leftarrow max(TC, \mathit{vcell.timestamp}) + 1$ $\mathit{vcell.timestamp} \leftarrow TC$ $h_{ws} \leftarrow h_{ws} \oplus PRF(\mathit{vcell.addr}, \mathit{vcell.content})$

Table 2: *Details of TM read() and write() methods.*

Deferred Memory Verification

Deferred memory verification generalizes the idea of offline memory checking. Regular reads and writes do not detect memory integrity violations but merely update internal hashes. Memory integrity violations are detected in a separate verification step that involves scanning the entire memory.

This verification step is triggered when the client invokes the $\mathit{DefVerify}()$ operation. To support deferred verification, the TM exposes three methods: $\mathit{DefVerifyBegin}()$, $\mathit{DefVerifyNext}(\mathit{vcell})$ and $\mathit{DefVerifyEnd}()$. These methods are called by the worker thread handling $\mathit{DefVerify}()$ to initialize, iterate over all vcells, and obtain the result of verification, respectively. ($\mathit{DefVerifyEnd}()$ also generates a MAC using the client session key to protect the result of the verification.)

Unlike offline memory checking, a theoretical idea where memory is checked at the "end of time," deferred verification can be run any number of times, and each run verifies the integrity of all memory operations since the previous run. In Section 5, we present techniques to make deferred verification concurrent with regular data operations.

TM State and Read-Write operations

The TM maintains a read-set hash h_{rs}, a write-set hash h_{ws}, and a timestamp counter TC. Two TM internal methods $\mathit{read}(\mathit{vcell})$ and $\mathit{write}(\mathit{vcell})$ are used to record TM reads and writes, respectively. The $\mathit{VerifyOp}()$ method calls $\mathit{read}()$ with its input (proof and aux) vcells, then makes any changes to these vcells, and calls $\mathit{write}()$ with the updated vcells before returning.

All vcells have an additional $\mathit{timestamp}$ field required for memory verification, so vcells of VM_{KV} have three data fields (key, $\mathit{nextkey}$, and value), a status bit, and a $\mathit{timestamp}$.

Table 2 shows the details of $\mathit{read}()$ and $\mathit{write}()$ methods. The $\mathit{read}()$ method updates the read-set hash h_{rs}, and $\mathit{write}()$ does the same for write-set hash h_{ws}. The hash function that we use hashes a set (or a bag) by xor'ing the image of each element under a *pseudo-random function*. Our hash function is different from Blum's: Blum et al. [5] show that a hash function constructed as above but with a *random function* is secure, but eventually provide a complex construction designed to minimize the required number of random bits. We use an AES-based pseudo-random function [37]; assuming that it is indistinguishable (as widely believed) from a true random function, we get the same security properties.

The $\mathit{write}(\mathit{vcell})$ method updates the TC counter and sets $\mathit{vcell.timestamp}$ to the updated TC value. TC is updated to $max(TC, \mathit{vcell.timestamp}) + 1$. This is a subtle change from the

Operation	*Steps*
DefVerifyBegin()	$h_{wsnew} \leftarrow 0, TC_{new} \leftarrow 0$
DefVerifyScan(vcell)	$h_{rs} \leftarrow h_{rs} \oplus PRF(vcell.addr, vcell.content)$ $TC^{new} \leftarrow TC^{new} + 1$ $vcell.timestamp \leftarrow TC^{new}$ $h_{ws}^{new} \leftarrow h_{ws}^{new} \oplus$ $PRF(vcell.addr, vcell.content)$
DefVerifyEnd()	$succ \leftarrow (h_{ws} = h_{rs})$ $h_{rw} \leftarrow 0, h_{ws} \leftarrow h_{wsnew}, TC \leftarrow TC_{new}$

Figure 4: *Details of TM deferred verification methods.*

original Blum algorithm, which checks $TC >= vcell.timestamp$ and increments TC. The normal case behavior of both variants is identical but when they detect memory integrity violations is slightly different. This change helps in parallelizing memory verification presented in Section 5.2. We prove the correctness of this variant in the full version [1].

Memory Initialization: The TM initializes $h_{rs} \leftarrow h_{ws} \leftarrow TC \leftarrow 0$ and calls $write()$ on each vcell with its initial contents. Recall from Section 4.2.1, for VM_{KV}, one vcell is initialized to $\langle -\infty, \infty, - \rangle$ and all others are marked as unused. This initialization establishes the RS, WS invariant described earlier (under hashing).

Deferred verification: Implementation

Deferred memory verification is performed by calling $DefVerifyBegin()$, iterating over each vcell using $DefVerifyNext(vcell)$, and calling $DefVerifyEnd()$ to obtain the boolean result of verification. Consistent with our informal description, $DefVerifyNext(vcell)$ internally calls $read(vcell)$ to update h_{rs}, and $DefVerifyEnd()$ returns the truth value of the equality ($h_{rs} = h_{ws}$) as the result of verification. To enforce each vcell is visited exactly once during the scan, the TM expects the vcells to be scanned in address order and uses a temporary internal variable to track the next expected address and enforce this ordering.

In addition to verifying prior operations, these methods also perform initializations required for the next verification. During the verification scan, memory is effectively "re-initialized" with current (verified) content, but with new timestamps for each vcell; TM internal state, h_{ws}, h_{rs}, and TC, is reset at the end of verification to reflect this re-initialization. Figure 4 presents these details.

5. ENHANCEMENTS

In this section, we present various optimizations that remove the restrictions of Basic Concerto including allowing for an arbitrary number of workers and verifiers.

5.1 UM Optimizations

Recall that the correctness of Basic Concerto relies only on the sequence of $VerifyOp()$ calls to the TM. Therefore any optimization that preserves the "TM view" of Basic Concerto (same sequence of $VerifyOp()$ calls) maintains the integrity guarantees.

A related observation is that verified memory state changes are *UM-computable*: While in Basic Concerto, we relied on the TM returning updated contents of input (proof and aux) vcells, as Table 1 suggests, the UM has sufficient information to compute these changes without relying on the TM. This observation is valid also for the *timestamp* field within vcells which depends on a TM internal counter: the TC counter deterministically increases by one after every operation so the UM can predict the current value of this counter. (The vcells still need to be sent to the TM for the side-effect of updating TM internal hashes.)

We use these two observations to make Concerto multi-threaded in the UM. Each worker thread now handles a client operation and makes all state changes in the UM simulating what Basic Concerto would have done with a TM $VerifyOp()$. Except for minor differences in how data records are encoded and the additional timestamp field, this part of Concerto processing is independent of integrity and similar to what happens in any key-value store. An important component of this UM processing is the key index used to efficiently identify the raw key-value data relevant to each operation: Our current prototype uses Bw-Tree [21], a multi-threaded lock-free index, but any index could be used.

Once a worker completes all the UM state changes for an operation, it fires off an *asynchronous* $VerifyOp()$ with the pre-images of relevant vcells—so the TM reads the state before any changes—and then moves on to the next client operation. We still have a single verifier thread that handles the $VerifyOp()$. This operation now returns only the result MAC, not the updated vcell values but is otherwise unchanged. (The updated vcell values are still internally computed to update the write-set hash). When the verifier is processing a $VerifyOp()$ the UM could generate other $VerifyOp()$s and these are enqueued at the UM. This queue decouples UM and TM processing in Concerto, and at any time, the key-value state in the UM could be ahead of the state in the TM implicitly captured in the read- and write-set hashes.

Finally, Concerto *batches* the parameters and results of multiple $VerifyOp()$s into a single call. This batching reduces the number of UM-TM roundtrips and is critical for trusted platforms such as a PCIe-based FPGA where each roundtrip incurs significant latency.

5.2 Parallelizing Verification

We now present extensions to Concerto that parallelize TM processing using multiple verifier threads so that different threads concurrently handle different $VerifyOp()$ calls. There is no affinity between TM verifiers and UM workers in our design: A $VerifyOp()$ from any worker can be handled by any verifier.

High-level Insight

Only the memory verification component of TM processing which involves recording vcell reads and writes into hashes is stateful. The key-value component is stateless except for the session key which is read-only after session-setup and therefore does not interfere with concurrency.

One mechanism for parallelizing memory verification is to shard the verified memory and have each verifier thread own a shard. We do not rely on such sharding, which would perform poorly for skewed workloads.

Our design allows $VerifyOp()$ to be routed to any verifier, meaning any vcell can be read and written by any verifier. Each verifier tracks its own read- and write-set hashes corresponding to the vcell reads and writes in its $VerifyOp()$s. During deferred verification, the verifier-local read- and write-sets hashes are combined to create hashes of the global read- and write-sets. The key insight is that the xor operation used in the construction of hash is distributive, so the global read- and write-set hashes can be computed by xor'ing the local ones. Finally, we check the equality of the global read- and write-set hashes to determine if memory operations were read-write consistent. Figure 14 illustrates these ideas for reads and writes of a single vcell across two verifiers.

Details

Each verifier thread has a unique id and maintains local read- and write-set hashes and a TC counter; for thread i, we refer to this state using $h_{rs}(i)$, $h_{ws}(i)$, and $TC(i)$. Each vcell, in addition to

Operation	*Steps*
read(vcell)	$h_{rs}(i) \leftarrow h_{rs}(i) \oplus PRF(vcell.addr, vcell.content)$
write(vcell)	$TC(i) \leftarrow max(TC(i), vcell.timestamp) + 1$ $vcell.timestamp \leftarrow TC(i)$ $vcell.vid \leftarrow i$ $h_{ws}(i) \leftarrow h_{ws}(i) \oplus PRF(vcell.addr, vcell.content)$

Figure 5: *Details of TM read() and write() methods at verifier i.*

the timestamp field now contains a vid field that records the id of the verifier that saw the most recent write to the vcell. The $read()$ processing is unchanged and the $write()$ processing is unchanged except that $vcell.vid$ field is updated with the id of the verifier thread. Deferred verification methods $DefVerifyBegin()$, $DefVerifyNext()$, and $DefVerifyEnd()$ are sent a special leader verifier (say, with id 0). The implementation of these steps is similar to the non-parallel case, except $DefVerifyEnd()$ aggregates individual read- and write-hashes into global read- and write-hashes using an xor before checking their equality. Figure 5 presents these details.

Meaning of timestamps: In the non-parallel version, TC is an internal clock that tracks the number of writes since the previous deferred verification, and the timestamp field for a vcell is the time of its most recent write. In the parallel version, per-verifier $TC(i)$ can be viewed as a collection of distributed *Lamport's clocks* [19] if we treat the vcells as "messages": If verifier i writes a vcell with timestamp t_i a different verifier j that next reads this vcell updates its timestamp to $1 + max(t_i, TC(j))$. This timestamping scheme ensures that a read time is always less than the time of the next write to a vcell, a property that is critical to argue correctness. We prove the correctness of this scheme in the full version [1].

5.3 Non-Quiescent Deferred Verification

In Basic Concerto, deferred memory verification is monolithic: once deferred verification starts ($DefVerifyBegin$), there are no data operations until all vcells are scanned using $DefVerifyNext()$ and $DefVerifyEnd()$ returns the verification result. We now extend Basic Concerto to allow $DefVerifyNext()$ calls to be interspersed with regular data operations (and therefore memory reads and writes) and then describe how these extensions work with the other optimizations described in Sections 5.1-5.2.

In Basic Concerto, we imagine the entire system as existing in an integer *epoch*: Initially, the system is in epoch 0 and the current epoch increases by one after every deferred verification scan. We begin our modifications by using different TM state for each epoch: the state for epoch e consists of the read- and write-set hashes $h_{rs}[e]$ and $h_{ws}[e]$, and counter $TC[e]$. In epoch e, reads and writes update $h_{rs}[e]$ and $h_{ws}[e]$, respectively, and use $TC[e]$, instead of the unindexed h_{rs}, h_{ws} and TC. Clearly, separating the TM state in this way does not impact correctness.

The basic idea to break up the monolithic deferred verification is to use the epoch indexed state to let memory exist in two epochs simultaneously: some vcells exist in the *current* epoch e and others in the *next* epoch $e + 1$. We use the same TM methods as before and we interpret $DefVerifyNext()$ as transitioning its input vcell from epoch e to epoch $e + 1$. Regular memory read-write operations can now occur between two $DefVerifyNext()$ calls. In the implementation of these operations, a vcell that has been transitioned to epoch $e+1$ uses $h_{rs}[e+1]$, $h_{ws}[e+1]$, and $TC[e+1]$, and vcell yet to be transitioned uses $h_{rs}[e]$, $h_{ws}[e]$, and $TC[e]$; all other details of TM read-write methods are unchanged. $DefVerifyNext(vcell)$ itself is special: it reads the input $vcell$ in epoch e and writes the output to epoch $e + 1$. Finally, $DefVerifyEnd()$ checks $h_{rs}[e] = h_{ws}[e]$ as before. Figure 6 illustrates this overall scheme.

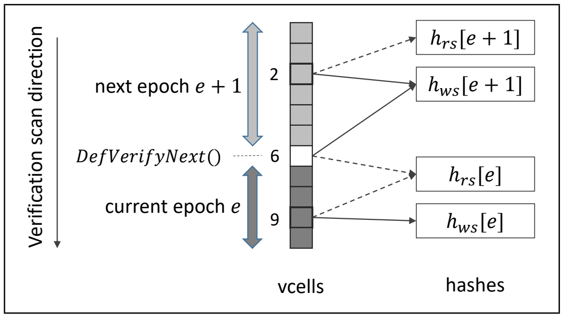


Figure 6: *The idea of epochs for non-quiescent deferred verification: $DefVerifyNext$ has been called on vcells 0-5. A read-write operation on vcell 2 now uses $h_{rs}[e + 1]$, $h_{ws}[e + 1]$, and $TC[e + 1]$. A read-write operation on vcell 9 uses $h_{rs}[e]$, $h_{ws}[e]$, and $TC[e]$.*

The epoch mechanism is a logical concept that emerges from the sequence of $DefVerifyNext()$ and $DefVerifyEnd()$ calls. In particular, vcells are not physically copied to a different location when they transition epochs; we do not maintain state within each vcell recording which epoch the vcell belongs to. For correctness, the sequence of $DefVerifyNext()$ calls is constrained to happen in increasing order of vcell addresses as illustrated in Figure 12. To enforce this order, the TM maintains an internal counter $next\_transition\_addr$ that tracks the address of the next vcell to be transitioned to the next epoch. This counter also helps the TM determine the epoch to which a vcell belongs: vcells with an address less than $next\_transition\_addr$ belong to the next epoch, and the rest, to the current epoch. In Figure 6, the value of this variable is 6. Once all vcells have been transitioned to the next epoch, the TM expects a $DefVerifyEnd()$ call to verify the integrity of memory operations of the current epoch. We note that there are at most two active epochs at any given point and the $DefVerifyEnd()$ calls happen in the increasing order of epochs.

To argue correctness, we can show that any sequence of $read()$, $write()$, and deferred verification calls can be reordered to an equivalent sequence that has the same final hash values ($h_{rs}[e]$, $h_{ws}[e]$ for all epochs e) but where the deferred verification calls are contiguous like in Basic Concerto.

Since only two epochs are active at any point in time, it suffices to store h_{rs}, h_{ws}, and TC for the current and next epoch, for a total of four hash values and two timestamp counters. As noted above, the TM also maintains a counter $next\_transition\_addr$ to help constrain epoch transitions.

TM parallelism: When there are multiple verifier threads, each thread maintains its local (four) hashes and (two) timestamp counters. As in Section 5.2, all deferred verification is handled by a special verifier thread with id 0. This thread also maintains the $next\_transition\_addr$ variable to check the completeness of deferred verification scan. The main issue is for other verifier threads (which do not have this variable) to determine if a vcell that they see is in the current or next epoch. To enable this, we add an epoch field to the vcells; in fact, instead of the full epoch number, we store the parity of the vcell's epoch. Since each verifier tracks the current epoch, it can use the parity bit to determine if a vcell is in current or next epoch. We can show the regular memory verification mechanisms prevent tampering of the epoch parity field.

Continuous background verification: Based on the techniques described here, Concerto runs deferred verification as a continuous background operation that scans the verified memory at a configured rate transitioning vcells to the next epoch. When a client invokes $DefVerify()$, this call attaches itself to the current scan and gets the result at the end of the scan from the corresponding $DefVerifyEnd()$ call. We note that different client processes can concurrently invoke $DefVerify()$ and all these calls share the same scan and the result. In our current prototype, the scan rate is configured by a parameter V with the interpretation that we transition one vcell after every V data operations.

6. IMPLEMENTATION

Our current prototype implements the techniques and optimizations presented in Sections 4-5. For the key-index, we use Bw-Tree [21], a state-of-the-art latch-free main-memory index structure. We implemented our own logging module that includes some low-level optimizations to log timestamp increments for data read operations compactly. The prototype starts with verified memory of size one containing the vcell $\langle -\infty, \infty, - \rangle$ and uses memory elasticity techniques (Section A.2) to adjust the size of memory. Concerto currently supports two trusted platforms that we describe next.

6.1 FPGA-based Trusted Platform

Prior work [39] has shown that FPGA can be used as a trusted platform for integrity verification; this paper also contains details such as embedding keys and non-volatile memory required for integrity.

As part of this work, we have designed an highly optimized FPGA circuit that implements the TM verification functionality described earlier. Briefly, our design involves multiple FPGA cores and each core implements the functionality of one verifier thread: at each step, it consumes a batch a $VerifyOp()$s, updates internal read and write-set hashes, performs checks specified in Section 4.2, and returns a result MAC for each operation result. Our design incorporates several hardware-level optimizations such as pipelining AES block computations—our PRF [37] and MAC are both AES-based, so AES block computations form the bulk of verification computations. A full description of these optimizations is outside the scope of this paper but Section A.1 offers a brief summary.

As a representative number illustrating our design optimizations, with 8-byte keys and 8-byte values, a single core of the FPGA circuit is theoretically able to process 19M inserts/sec and 27M lookups/sec, with a 175 MHz clock. As we will see, a single FPGA core is able to handle verification input from 20 worker threads running on 20 cores on a server-grade machine.

6.2 SGX-like Trusted Platforms

We also have a C++ implementation of the TM functionality for emerging platforms such as Intel SGX [24] that support regular binaries to be loaded and run within the platform. While server-grade Intel CPUs supporting SGX are not yet available, prior work has suggested a methodology for simulating SGX performance on an Intel CPU without SGX [4, 35]. This methodology essentially adds a penalty (e.g., TLB flush) every time a thread enters the *enclave (trusted) mode*. While we do report numbers based on this simulation, we expect these numbers to be ballpark and useful mainly to illustrate relative performance; in particular, we use this to compare the performance of MB-tree with Concerto. Our C++ implementation also quantifies the size of our trusted code (TCB)—around 100 lines of C++ code not counting the crypto libraries for AES block computations.

7. EXPERIMENTS

This section presents the results of an experimental evaluation of Concerto. The goals of our experiments is to: (1) evaluate the overhead of integrity in Concerto by comparing it against a key-value without integrity; (2) evaluate the performance advantages of our design compared to a Merkle-tree based approach; (3) study the tradeoff between deferred verification rate and throughput in Concerto; (4) understand concurrency characteristics of Concerto by varying available parallelism in the untrusted and trusted platforms.

7.1 Benchmark and Methodology

All experiments reported in this paper were performed using the YCSB benchmark [7] designed for key-value stores. Each experiment involved loading an initial state of the database and then running one of four workloads described below against it. We used 8-byte integers for both keys and values. The initial database consisted of N key value pairs with all keys in the range $1 \ldots N$ and values picked randomly. Unless otherwise stated, we used $N = 10$ million pairs as the initial database size.

We used the following four workloads specified by YCSB:

1. *A. Update-heavy:* This workload consists of 50% $get()$ operations and 50% $put()$ operations.

2. *B. Read-heavy:* This workload consists of 95% $get()$ operations and 5% $put()$ operations.

3. *C. Read-only:* This workload consists of 100% $get()$ operations.

4. *D. Inserts:* This workload consists of 95% $get()$ operations and 5% $insert()$ operations.

For workloads A, B, and C, the keys of all $get()$ and $put()$ operations were in the range $[1 \ldots N]$, where N is the database size (which does not change for these workloads). Since these keys are all present in the database, all operations were successful. As specified in the benchmark these keys are generated using a Zipf distribution. Sampling from the Zipf distribution requires assigning a frequency rank to the keys: As per the benchmark specification, we used a random permutation of the key-domain for the ranking, so the frequently used keys were randomly sprinkled across the key domain. For workload D, the newly inserted keys are random keys not in the original database; a Zipf distribution is used to pick keys for $get()$ operations, but the rank is derived using recency, i.e., the most recently inserted key is queried with the highest probability.

Unless otherwise specified, we ran the experiments as follows: Each experiment was repeated three times and in each of these three runs 100 million operations specified by the corresponding workload were evaluated. We measured both throughput and latency. The throughput was measured as an average throughput over time (while a run was executed) and across the three runs. For Concerto, we included the time for deferred verification when calculating throughput numbers. The throughput variance was small so we do not show error bars. For the latency experiments, we show both the median and the 90% quantile over all three runs.

We implemented the YCSB benchmark driver in C++ and all the systems being compared were linked to the driver code as a library. This means that all the key-value operations were simple function calls and the same thread assumed the role of both workload generator and the (UM) worker. Our results, therefore, do not include network effects which could dampen the overhead of integrity. The workload generation itself is lightweight and forms an insignificant part of the overall CPU costs.

7.2 Systems Compared

We compared the following systems in our evaluation:

- *No Integrity Baseline:* We used Bw-Tree [21] as the baseline system without integrity.
- *Concerto(FPGA):* This system is the Concerto prototype of Section 6 with the FPGA-based TM presented in 6.1.
- *Concerto(SW):* This system is Concerto where the TM functionality is implemented in regular C++ code as discussed in Section 6.2.
- *Merkle (SW):* We implemented the MB-trees by extending Google B-trees, an open source implementation of B-trees. Here again, the trusted code is regular C++ code running in a separate thread as discussed in Section 6.2.
- *Merkle with Batching (SW):* We extended the MB-tree above to support batching. The combined proof of a batch of operations is the union of the proof-paths of each operation, i.e., a subtree of the MB-tree. The TM verifies the integrity of the entire subtree and returns new hash values for all nodes in the subtree, except the root which is maintained within the TM. We expect batching to improve performance since hash computations are shared across operations; batching also reduces contentions between operations within a batch.

All experiments were carried out on a dual-socket 3.4 GHz Windows Server 2012 machine with 6 physical/12 logical cores per socket and 128 GB of main memory. Logs were written to a Samsung MZHPU512HCGL SSD (PCIe 2.0 x4) with a maximum read and write bandwidth of 1170MB/s and 930MB/s respectively. The Concerto(FPGA) system used an Altera Stratix V GSD5 FPGA connected via PCIe 3.0 x8 (a theoretical bi-directional bandwidth of 7.7GB/s).

For Concerto systems, a deferred verification operation was always on in the background. Unless otherwise mentioned, we configured these systems to verify (transition to next epoch) one vcell for every 16 data operations. Therefore, hypothetically, if Concerto processes $1M$ operations per second on a database instance of size $1M$ records, then each complete deferred verification scan takes 16 seconds, meaning each operation is verified within 16 seconds of its completion (our actual numbers are in the same ballpark).

7.3 Exp 1: Throughput , Vary UM Cores

Figure 7 shows the throughput of the four compared systems for workloads A, B, C, and D. In these experiments, we used one verifier thread (for systems supporting integrity) and varied the number of UM workers, each pinned to a separate CPU core.

Our first observation is that for all workloads, the throughput of Concerto(FPGA) is competitive with the non-integrity baseline and is at most a factor 2 away. We reiterate that these throughput numbers include the cost of the deferred verification. While the actual throughput numbers vary, the relative performance of Concerto(FPGA) vs. non-integrity baseline does not significantly change as we vary the workload characteristics from read-only to update-heavy.

Second, we note that the performance of all systems except Merkle(SW) increases almost linearly as we increase the parallelism in the untrusted system. In particular for Concerto(FPGA), since we keep the number of verifier threads the same (one), this indicates that the untrusted functionality (key index traversal, serializing $\mathit{VerifyOp}()$ parameters, and logging) is the bottleneck. Although both have the same bottleneck, the performance of Concerto(FPGA) is less than that of the non-integrity baseline, since Concerto(FPGA) does additional UM work (e.g., serializing parameters) and also incurs additional contention for the TM-bound $\mathit{VerifyOp}()$ queue. The UM being the bottleneck also explains why the performance of Concerto(FPGA) is comparable to Concerto(SW), although these two use vastly different TM computational platforms (CPU vs. FPGA) and vastly different UM-TM communication (PCIe bus vs. shared memory).

Third, the large asymmetry between the number of worker threads (up to 12) and the number of verifier threads (fixed at 1) in these experiments demonstrates the lightweight nature of Concerto's TM functionality.

Fourth, our throughput numbers expose a subtle trend when we move from a update-heavy workload (A) to a read-only workload (C). As expected, the performance of the non-integrity baseline increases significantly—around a factor 1.6 for the 12 worker case. The performance improvement of Concerto is modest by comparison—around a factor 1.2. This happens because Concerto pays a logging overhead even for a read-only workload (due to Blum algorithm timestamps) while the no-integrity baseline does not.

Last but not the least, the performance of Concerto(SW) is about three orders of magnitude greater than that of Merkle(SW). Adding batching to Merkle tree does not significantly improve performance. Figure 8 compares the performance of batched Merkle tree against the non-batched version (leftmost data point with batch size 1). The performance improvement of batching flattens out after a batch size of about 20. The various counters (e.g., the number of nodes hashed) we collected from the experiments indicate that batching does not significantly reduce hash computations, which is the overall bottleneck of the system. As suggested in Section 3, this happens since different operations do not share their proof paths. These numbers support the central thesis of this paper that a batched and deferred verification model can be leveraged for tremendous performance gains.

7.4 Exp 2: Throughput, Vary Epoch Transition Rate

Recall that in our current prototype, a verification scan that transitions each vcell from one epoch to the next is always "on" in the background. This transition is programmed to happen at a fixed rate of 1 vcell epoch transition ($\mathit{DefVerifyNext}$ call) for every R normal key-value operations, where R is a configurable parameter that controls the rate of epoch transition. A smaller value of R implies higher verification rate and shorter epochs; since we do more verification work per key-value operation, we expect lesser throughput. In all the previous experiments, R was set to 16. In this experiment, we vary R and study the effect on throughput. Figure 9 shows the actual trend that we observe is consistent with this expectation.

We can use the results of Figure 9 to determine the maximum delay in detecting integrity violations. Integrity violations are detected once all vcells are transitioned from current to next epoch. At $R = 2$, Concerto achieves a throughput of about 2M ops/sec, so it transitions vcells from current epoch to the next at the rate of 1M vcells/sec. Since the overall size of verified memory $\approx 10M$, the size of the database, the epoch transition rate for the entire memory is about 10 seconds.

The main takeaway here is that absolute time for each verification can be made relatively small without seriously affecting performance. For example, we can achieve most (90%) of the throughput while verifying the entire database every 20 seconds. This means that any database tampering by an adversary would go undetected for a mere 20 seconds before Concerto detects it. We believe for many applications, the 20-second delay in detecting tampering might be acceptable given the three orders-of-magnitude performance improvements that this delay enables.

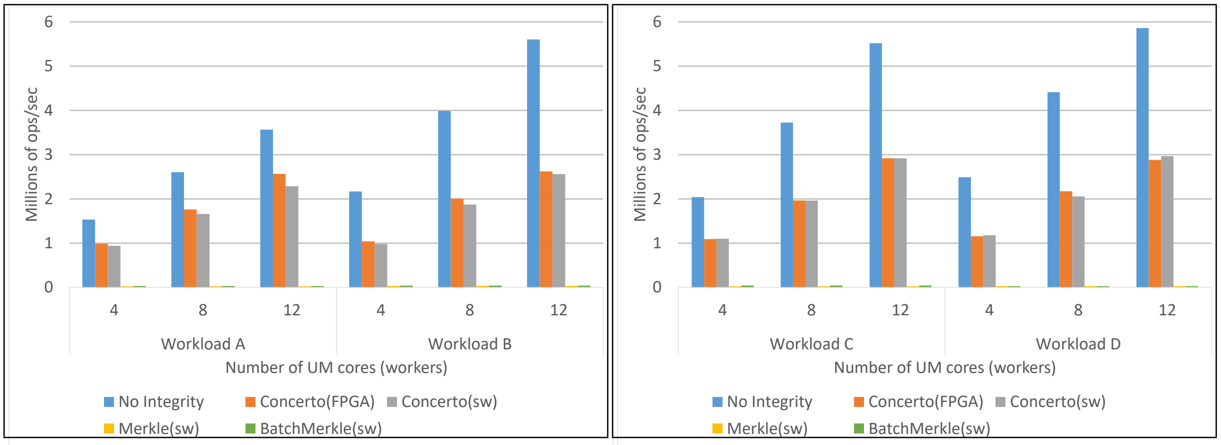


Figure 7: *Experiment 1: Throughput for workloads A, B, C, D*

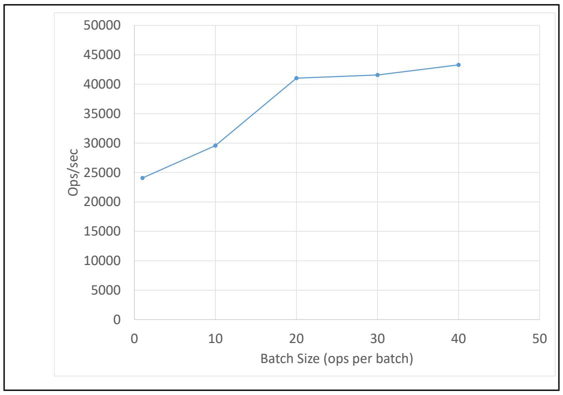


Figure 8: *MB-tree with batching (leftmost point is non-batching case with batch size 1). Workload C, 12 UM, 1 TM.*

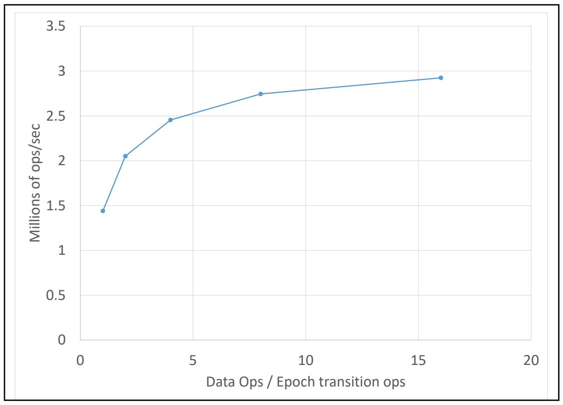


Figure 9: *Exp. 2: Throughput of Concerto(sw), Vary Epoch Transition Rate Workload C, 12 UM, 1 TM*

7.5 Exp 3: Throughput, Vary TM Cores

In this experiment, we enable additional parallelism in the trusted platform and study its impact on overall throughput. As noted in Section 7.3, the overall bottleneck in the system for the configuration studied there was the untrusted processing. Therefore, enabling additional parallelism in the TM does not help, and our experiments (not shown) validate this. This observation remains valid with Concerto(FPGA) even when we increase the number of workers (and CPU cores) to 20[2]; our FPGA TM was simply able to handle the increased verification work without becoming a bottleneck. This changes when we switch to Concerto(SW): Figure 11 which shows the overall throughput for workload C when we vary the number of UM cores from 4-18 and the number of TM cores from 1-2. With 1 TM core, the throughput tapers off at around 12 UM cores suggesting that the TM becomes the bottleneck. With 2 TM cores, the throughput keeps increasing until 18 UM cores. This experiment suggests that with sufficient UM parallelization, a non-parallel TM can become the bottleneck and our parallelization techniques of Section 5.2 are important to make the best use of available UM and TM hardware parallelism.

7.6 Exp 4: Performance of Recovery

As noted in Section C, in the worst case, the adversary can introduce errors and arbitrarily slow down Concerto. That said, a key value store can have a broad spectrum of error conditions, not necessarily all malicious. In this experiment, we seek to understand the performance of Concerto in the presence of such errors. We devise a synthetic experiment where we inject memory corruption into the system at a controllable rate and study the effective throughput. The log itself is untampered, so Concerto rolls back to the most recent verified epoch (which is the previous epoch of the one where the error is detected). In our current prototype, recovery is quiescent (no data operations). For the setup of this experiment, it involves applying the log from the previous epoch and running a verification scan over the memory, which succeeds since log records are untampered. In addition to the fault injection rate, we vary the epoch transition rate using the parameter R described in Experiment 7.4.

Figure 10 plots the overall throughput of the system, factoring in the lost work due to the fault and reduced performance due to recovery, for different settings of the epoch transition rate parameter. As Figure 10 indicates, the effective performance drops precipitously as the fault interval approaches the epoch duration (around 1 minute), but quickly approaches normal performance as the error rate interval increases beyond the epoch duration. Also, as expected, at higher fault rates, configurations that use shorter epochs (smaller R value) outperform configurations with larger epochs, since they have lesser wasted work and a shorter log to process during recovery.

[2] For this experiment alone, we switched to a different machine with 10 physical and 20 logical cores.

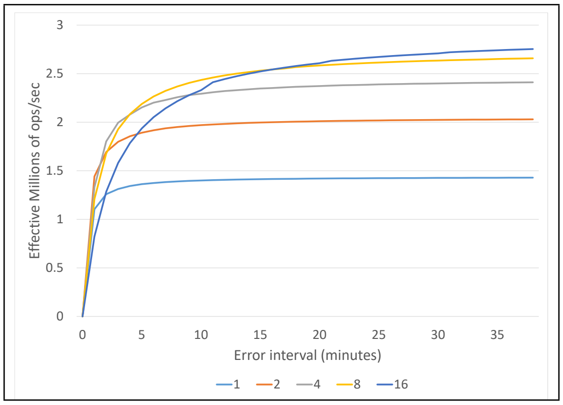


Figure 10: *Exp. 4: Effective throughput in presence of errors. Concerto (SW), 12 UM, 1 TM, Workload B*

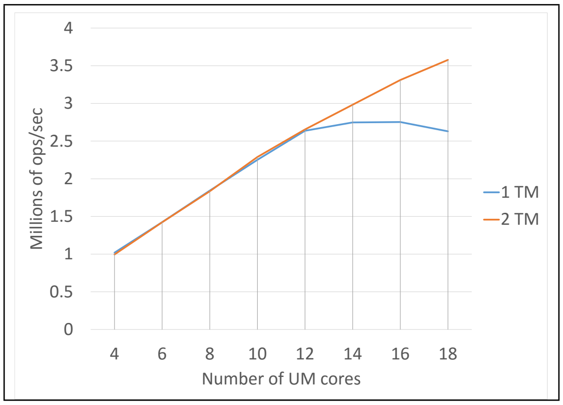


Figure 11: *Exp. 3: Throughput, Vary UM and TM cores in Concerto(SW)*

APPENDIX

A. ADDITIONAL DETAILS

A.1 FPGA as the trusted machine

An important building block of Concerto is server-side trusted computing. One contribution of this work is that we have shown that a single-core FPGA-based implementation of the TM can accommodate the full workload generated by a server-class machine. Here, we would like to provide some background on both the nature of the FPGA as a trusted computing element and a brief description of our hardware implementation of the Concerto TM. In [10], the authors describe a methodology to use existing commodity FPGAs to build a cloud-based trusted computing platform. This work describes the bootstrapping of the FPGA with a cryptographic identity and the subsequent loading/authentication of application code, in this case, our implementation of the Concerto TM.

Each FPGA-based Concerto TM is heavily pipelined for maximum performance. As shown in Figure 15, for each batch of data the TM must: (1) update read-set hashes for all vcells, (2) check the MACs of operations to verify their integrity, (3) update the contents of input vcells based on the operations, (4) update the write-hashes using the updated vcells, and (5) generate the output MACs for the results. One important computation to both the MAC and hash operations is an AES block operation. In our implementation, we use two fully unrolled AES modules.

Each input batch is serialized as a sequence of vcells and operations: given that vcells and operations are streamed onto the FPGA and that our AES block implementation can compute hash updates and MACs in completely pipelined fashion, we use the same AES module (A in Figure 15) for both computations. VCells are stored in internal memory (module B) and updated by the input operations (module C). As discussed in Section 5.1 , a vcell could be updated by multiple operations and the final updated version of the vcell is used to update the write-set hash. As an optimization, we speculatively compute the write-set hash delta whenever a vcell is updated; if the same vcell is updated again, we undo the previous update (by xor'ing the previous hash delta) and apply the new delta. We use separate AES modules (D and E) for computing the write-set hashes and the result MACs; the latter is streamed out back to the UM.

This fully pipelined implementation allows a single TM core to process batched operations at 1.4 GBps. Extrapolating, if the UM were to be able to saturate the PCIe connection with work, five TM cores would be needed to prevent the TM from becoming a bottleneck. However, as shown Section 7, even the baseline system with no integrity creates only 0.5 GBps of verification work for the TM. That said, each TM core occupies 21% of FPGA resources on the Altera Stratix GSD5 chip used in our testing, so adding TM instances is possible.

A.2 Verified Memory Elasticity

We describe memory elasticity for Basic Concerto of Section 4; extending these details to Concerto with optimizations of Section 5 is straightforward and uses the idea of relying on a special verifier thread for memory elasticity operations.

To support memory elasticity, the TM maintains an internal variable, $size$, which tracks the current size of verified memory. To increase the size of memory, the TM exposes a $NewVCell()$ method. The implementation of this method within the TM initializes a new vcell with address $size$ and a predefined initial state (for VM_{KV}, the initial state, e.g., would set the status bit to invalid). It performs a $write$ on this vcell to update h_{ws}, increments $size$, and returns the new vcell.

Memory shrinking is more involved. For shrinking to be usable in practice, we need the ability to compact the valid vcells to locations with smaller addresses. To enable such compacting, the TM exposes a $Move(destvcell, srcvcell)$ with expected semantics. To shrink memory, the amount of memory size reduction is specified at the beginning of a deferred verification scan. This information is used during the scan to ensure that the h_{ws}^{new} (Figure 4) the write-set hash for the next epoch correctly reflects the reduced number of vcells. Finally, $DefVerifyEnd()$ updates $size$ to the new (reduced) value.

B. ADDITIONAL EXPERIMENTS

B.1 Exp 5: Latency, Vary UM Cores

Figure 16 shows the median and 90% quantile latency of Concerto(FPGA) for our four workloads. The main takeaway here is that the latency incurred by our current configuration of Concerto(FPGA) is significant, in the order of milliseconds. This is primarily because of our aggressive batching both for TM roundtrips and for logging. In other words, our current Concerto configuration is optimized for throughput at the expense of latency. In the cloud context though, high in-server latencies might be masked by the network latencies, and therefore acceptable. That said, optimizing throughput over latency is not central to our design, and we could expose different threshold/latency characteristics by tweaking our batching thresholds. A second related observation is that the latency is highest for the read-only workload C. This happens because this is the workload with the highest throughput and therefore able to batch operations the most.

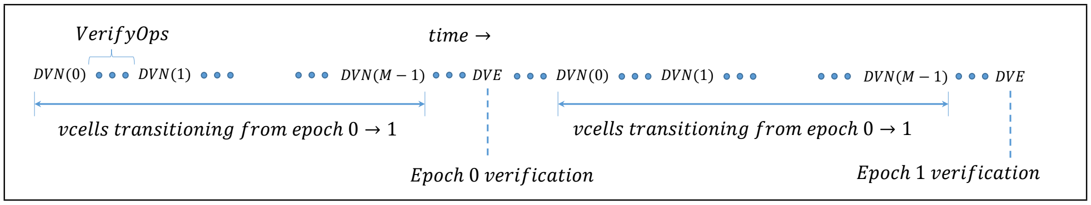


Figure 12: *Sequence of $DefVerifyNext()$, $DefVerifyEnd()$ calls. $DVN(0)$ is shorthand for $DefVerifyNext()$ for vcell 0, and DVE, a shorthand for $DefVerifyEnd()$. M denotes the size of verified memory.*

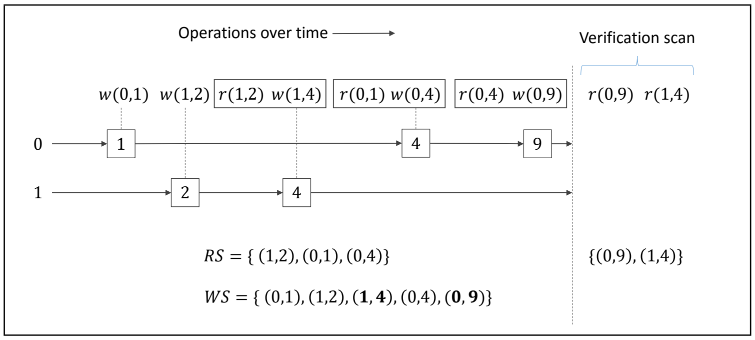

Operations over time
Verification scan
w(0,1) w(1,2) r(1,2) w(1,4) r(0,1) w(0,4) r(0,4) w(0,9)
r(0,9) r(1,4)
0
1
1
4
9
2
4
RS = { (1,2), (0,1), (0,4)}
{(0,9), (1,4)}
WS = { (0,1), (1,2), (1, 4), (0,4), (0, 9)}

Figure 13: *Read- and write-sets for 2 cells: $w(a, v)$ (resp. $r(a, v)$) indicates value v was written to (resp. read from) location a. In a read-consistent memory RS trails WS by exactly two elements. A verification scan adds these elements to RS and makes them equal.*

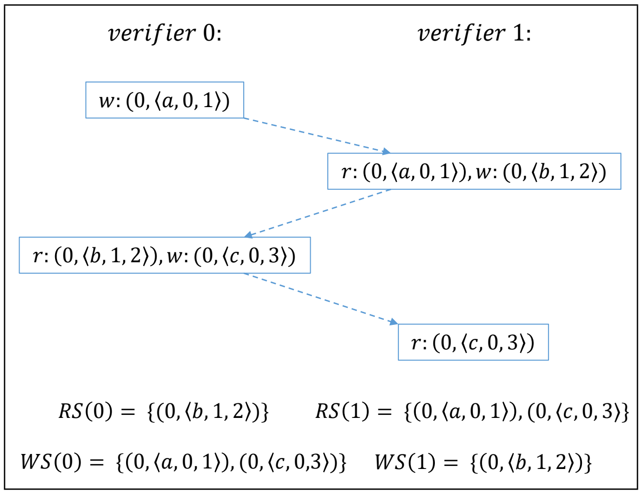

verifier 0:
verifier 1:
w: (0, ⟨a, 0, 1⟩)
r: (0, ⟨a, 0, 1⟩), w: (0, ⟨b, 1, 2⟩)
r: (0, ⟨b, 1, 2⟩), w: (0, ⟨c, 0, 3⟩)
r: (0, ⟨c, 0, 3⟩)
RS(0) = {(0, ⟨b, 1, 2⟩)}
RS(1) = {(0, ⟨a, 0, 1⟩), (0, ⟨c, 0, 3⟩}
WS(0) = {(0, ⟨a, 0, 1⟩), (0, ⟨c, 0,3⟩)}
WS(1) = {(0, ⟨b, 1, 2⟩)}

Figure 14: *Reads and writes of a single vcell with address 0 at two verifier threads. We use the notation $(a, \langle p, v, t\rangle)$ to denote a vcell with address a, payload p, vid v, and timestamp t. The read-sets and write-sets (under hashing) are tracked separately. With read-write consistent memory, we have $RS(0) \cup RS(1) = WS(0) \cup WS(1)$. Under hashing, the union operation becomes an $\oplus$.*

C. DURABILITY AND RECOVERY

To elaborate the durability guarantees provided by Concerto, we distinguish *anticipatable failures* from adversary induced failures. Anticipatable failures refer to failures such as power outages, disk and memory corruption. Traditional transactional systems model the effect of such failures to recover automatically using redundancy and logging. Concerto uses similar techniques, and a successful response from Concerto for a key-value operation means that the effect of the operation is durable and recoverable from anticipatable failures. (Our current prototype handles memory errors, but we can use standard redundancy techniques to handle disk errors.)

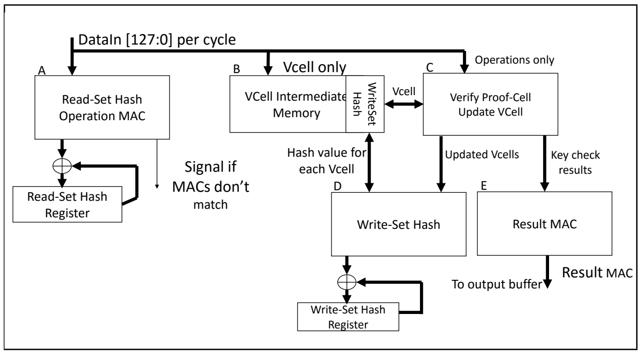


Figure 15: *Design of the FPGA TM*

Since the adversary has full control over the Concerto server machine, adversarial errors can be arbitrary. Durability guarantees of any transactional system do not hold in the presence of such errors. For example, a rogue administrator could simply delete the database and the log, rendering it unrecoverable to a state consistent with a previous committed transaction. Recovery from adversarial errors cannot therefore be automatic and requires out-of-band mechanisms.

Recovery Guarantees: For all failures, Concerto exposes a single method $recover()$ to recover the system to a functional state. During normal operations, Concerto logs various state changes, and recovery uses the information in the log to recover the system to the most recent consistent state possible. For transient failures in an untampered system, such a recovery is *complete*, i.e., no updates are lost, and normal processing resumes at the end of recovery. If there is tampering or non-recoverable hardware failure, recovery is partial meaning the system is in a consistent but historical state, and the system returns information (protected by a TM MAC) that can be used to determine lost updates. We emphasize that an adversary cannot induce the system to rollback changes without being detected.

Logging: During normal processing, Concerto stores changes to the UM and TM state in a log maintained in untrusted persistent storage.

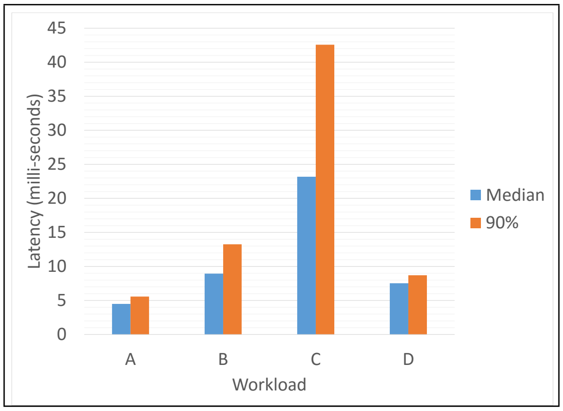


Figure 16: *Exp. 4: Latency (millisecs), 12 UM, 1 TM*

The log contains sufficient information so that during recovery we can restore the UM and TM state to the current or any historical version. We use standard techniques to log UM state changes; we log just the VM_{KV} changes (and therefore raw key-value data changes) and the key index is rebuilt after recovery.

Logging TM state changes involves additional details to prevent log tampering attacks. We assume the TM has access to a secret symmetric key SK_{TM}. Such a key can be built into the trusted platform [39] or derived from other primitives such as PKI identity provided by the platform. After processing a batch of $\mathit{VerifyOp}()$ calls, a verifier thread generates a log record that contains its latest hashes and TC counter value. It encrypts and MACs the record using SK_{TM} before returning it to UM. The UM stores the log record as part of the log. We note that there is one TM log record per batch, not per operation.

To ensure consistency between log records of verifier threads, the TM internally maintains a *log sequence number (LSN) counter*. When a verifier generates a new log record, it increments the LSN counter, and the current value of the counter is stored in the log record. While this counter is shared by all verifier threads, it is updated once per batch of operations so synchronizing access to the counter does not add much overhead.

The LSN counter is stored in nonvolatile storage within the TM. We note that a small amount of nonvolatile storage is available in many trusted platforms [24, 39]; TPM chips if available can also be used for this purpose using techniques presented in [34].

Recovery: Recovery is a series of attempts to get the system to a consistent state using the information in the log. In each attempt, we logically replay a prefix of the log to restore the UM and the TM to a historical version, and we check if the UM state is consistent with the TM hashes by running a deferred memory verification scan. If the verification succeeds, the TM determines whether or not the recovery was complete by comparing the internal LSN counter with the LSN of the TM log record visited during recovery.

For correctness, we note that the adversary cannot tamper with the TM log records due to the MAC protecting them. The TM checks during recovery that these records are visited in LSN order. This implies that the resulting TM state is a legitimate historical one. The restored UM state may or may not be legitimate, but this aspect is checked by the deferred verification step.

We have omitted in our description details relating to recovery performance such as the use of checkpointing to avoid replaying the entire log. These details are fairly standard and not specific to Concerto.

D. RELATED WORK

There is a rich body of prior work [3, 4, 8, 13, 22, 27, 30, 31, 36, 40] on integrity verification for outsourced databases. It is useful to distinguish between the general setting where a collection of identical clients use and update the database from the *publication setting* where a single *data owner* publishes data, and the other clients query it.

Merkle Trees: Most prior systems rely on Merkle trees [25]. One important design choice in these systems is the location where the hash of the Merkle tree root is stored: It could be stored in an in-server trusted module [3, 4] or at a client [8, 13, 22, 36]. The latter approach is often used in the data publication setting with the data owner storing the root hash. To securely propagate the root hash to other clients, the data owner typically *signs* the root hash and stores the hash and its signature at the untrusted database server. The other clients fetch the root hash from the server, which is unforgeable due to the signature. This approach, however, makes the problem of ensuring *freshness* of query results in presence of database updates challenging. To invalidate an out-of-date hash and its signature, these systems rely on techniques similar to *certificate revocation* [26]. The freshness guarantees resulting from these techniques is approximate (e.g., no tuple is stale by more than δ hours or days) which is acceptable for data publication, but weaker than the serialization-based guarantee of Concerto.

One notable example of a client-based system that does provide strong serialization guarantees is [13]. Here, both data owner and other clients can introduce updates. The untrusted server computes the new root hash after an update; the sequence of root hashes and updates that produce them is logged at the data owner. The server could introduce an integrity violation and log an incorrect root hash which is not immediately detected by the data owner. The verification is *on-demand*, and the fidelity of any suspicious operation can be verified subsequently from the root hashes immediately before and after the operation from the data owner log. The verification of one operation does not verify other (previous) operations so integrity violations could go undetected unless all operations are verified (which is expensive). In Concerto, in contrast, verification is not tied to a particular operation and checks the integrity of *all* prior operations. (Other than the innovative logging protocol, the actual data processing uses regular Merkle trees similar to our baseline implementation in Section 7.)

Signature Aggregation: Beyond Merkle trees, a second approach used in prior systems relies on digital signatures [30, 29, 32]. Informally, database tuples are signed with the private key of the data owner. The unforgeability of the signatures helps verify the authenticity of the tuples. (An interesting optimization available for these systems is *signature aggregation*, where signatures for individual tuples can be aggregated to a single signature reducing verification costs.) For completeness, these systems rely on a clever signature generation scheme that covers adjacent tuples in a sort order. Unlike Merkle tree, this approach does not have a single updatable digest, so ensuring freshness is challenging. One interesting approach to addressing this issue is presented in [32], where the data owner periodically publishes signed bitmaps of updated tuples using which clients can check for stale records.

General Queries: While Merkle trees and signature-based approaches can be used to verify SPJ queries, the size of proofs they generate could be large compared to result size, which translates to high verification cost. Papamanthou et al. [33] propose techniques for optimal verification of operations over dynamic sets; these techniques are optimal in the sense that their proofs and verification costs are linear in the query and result size. These ideas are generalized in IntegriDB [40], a system that supports efficient verification of SPJ queries with some aggregations.

General Systems: Beyond database systems, our work is also related to file systems [23, 11, 39, 12] and web applications [15] providing integrity verification. These systems ultimately rely on Merkle trees to provide integrity for the underlying state.

Redactable Signatures: Kundu and Bertino [18] introduce an interesting variant of integrity verification in data publication setting: The outsourced data is tree-structured such as an XML document and each client is authorized to view a part of the data such as a subtree. The goal is to design a verification scheme such that the client can verify the subtree she retrieves is authentic; at the same time the client should not learn anything about data not in the subtree or even that the subtree is *redacted* from a larger tree. Brzuska et al. [6] formalize the notion of redactable signatures and Kundu et al. [17] extend these techniques to graph-structured data.

E. REFERENCES

[1] A. Arasu, K. Eguro, R. Kaushik, et al. Concerto: A high concurrency key-value store with integrity (full version). Technical report, Microsoft Research, 2017.

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

[3] 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.

[4] A. Baumann, M. Peinado, and G. C. Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI*, pages 267–283, 2014.

[5] M. Blum, W. S. Evans, et al. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[6] C. Brzuska, H. Busch, Ö. Dagdelen, et al. Redactable signatures for tree-structured data: Definitions and constructions. In *ACNS*, pages 87–104, 2010.

[7] B. F. Cooper, A. Silberstein, et al. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[8] P. T. Devanbu, M. Gertz, C. U. Martel, and S. G. Stubblebine. Authentic data publication over the internet. *Journal of Computer Security*, 11(3):291–314, 2003.

[9] C. Dwork, M. Naor, G. N. Rothblum, et al. How efficient can memory checking be? In *TCC*, pages 503–520, 2009.

[10] K. Eguro and R. Venkatesan. FPGAs for trusted cloud computing. In *FPL*, pages 63–70, 2012.

[11] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *NDSS*, 2003.

[12] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *ISC*, pages 80–96, 2008.

[13] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *ICDE*, pages 529–540, 2013.

[14] J. Jannink. Implementing deletion in B+-trees. *SIGMOD Record*, 24(1):33–38, 1995.

[15] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for web applications. In *IEEE Security and Privacy (SP)*, pages 895–913, 2016.

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

[17] A. Kundu, M. J. Atallah, and E. Bertino. Leakage-free redactable signatures. In *CODASPY*, pages 307–316, 2012.

[18] A. Kundu and E. Bertino. Structural signatures for tree data structures. *PVLDB*, 1(1):138–150, 2008.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[20] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[21] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*, pages 302–313, 2013.

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

[23] J. Li, M. N. Krohn, D. Mazières, et al. Secure untrusted data repository (SUNDR). In *OSDI*, pages 121–136, 2004.

[24] F. McKeen, I. Alexandrovich, A. Berenzon, et al. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[25] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

[26] S. Micali. Efficient certificate revocation. Technical report, MIT Laboratory for Computer Science, 1996.

[27] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2004.

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

[29] M. Narasimha and G. Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *DASFAA*, pages 420–436, 2006.

[30] H. Pang, A. Jain, K. Ramamritham, and K. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, pages 407–418, 2005.

[31] H. Pang and K. Tan. Authenticating query results in edge computing. In *ICDE*, pages 560–571, 2004.

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

[33] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, pages 91–110, 2011.

[34] B. Parno, J. R. Lorch, J. R. Douceur, et al. Memoir: Practical state continuity for protected modules. In *IEEE Security and Privacy (SP)*, pages 379–394, 2011.

[35] F. Schuster, M. Costa, C. Fournet, et al. VC3: trustworthy data analytics in the cloud using SGX. In *IEEE Security and Privacy (SP)*, pages 38–54, 2015.

[36] S. Singh and S. Prabhakar. Ensuring correctness over untrusted private database. In *EDBT*, pages 476–486, 2008.

[37] J. Song, R. Poovendran, J. Lee, et al. The advanced encryption standard-cipher-based message authentication code-pseudo-random-function-128 (aes-cmac-prf-128) for the internet key exchange protocol (ike), 2006. RFC 4615.

[38] S. Tu, W. Zheng, E. Kohler, et al. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[39] H. Yang, V. Costan, N. Zeldovich, et al. Authenticated storage using small trusted hardware. In *CCSW*, pages 35–46, 2013.

[40] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *CCS*, pages 1480–1491, 2015.