VeritasDB: High Throughput Key-Value Store with Integrity using SGX

Rohit Sinha
Visa Research
rohit.sinha@visa.com

Mihai Christodorescu
Visa Research
mihai.christodorescu@visa.com

***Abstract*—Applications depend on the safe operation of underlying databases. Alarmingly, cloud-backed database services face threats from exploits in the privileged computing layers (e.g. OS, Hypervisor) and attacks from rogue datacenter administrators, which tamper with the database's storage and cause it to produce incorrect results. Although integrity verification of outsourced storage and file systems is a well-studied problem, prior techniques impose prohibitive overheads (up to 10x in throughput) and place additional responsibility on clients.**

We present VeritasDB, a key-value store that guarantees integrity to the client in the presence of exploits or implementation bugs in the database server. VeritasDB is implemented as a network proxy that mediates communication between the unmodified client(s) and the database server, which can be any off-the-shelf database engine (e.g., Redis, RocksDB). Since the proxy is trusted, we use security primitives offered by modern processors, such as Intel SGX enclaves, to protect the proxy's code and state, thus completely eliminating trust on the cloud provider. To perform integrity checks in the proxy, we design an authenticated Merkle B-tree that leverages features of SGX (protected memory, direct access to unprotected memory from enclave code, and CPU parallelism) to implement several novel optimizations based on caching, concurrency, and compression. On standard YCSB and Visa transaction workloads, we observe an average overhead of 2.8x in throughput and 2.5x in latency, compared to the (insecure) system with no integrity checks — using CPU parallelism, we bring the throughput overhead further down to 1.05x. Thus, VeritasDB provides an order of magnitude improvement over existing techniques for integrity verification.

I. Introduction

While cost-efficiency and availability requirements drive businesses to deploy their databases in the cloud, it also requires them to blindly trust the cloud provider with their data. Remote attackers constantly try to exploit bugs in the database server or the infrastructure software (e.g. OS) to gain privileged access to the victim's databases; on-premise deployments face similar threats from remote attackers and rogue administrators alike. Accessing and tampering of sensitive data poses significant threat to an application, one that cannot be fully mitigated by encrypting and/or signing data at rest.

A client application requires that queries to a database return values that are consistent with the history of interaction between that client and the database. To that end, there is a large volume of work on database integrity verification [11], [12], [32], [20], [21], [31], mostly relying on *authenticated data structures* such as a Merkle hash tree (MHT). The main idea is to store a cryptographic hash digest of the database on the client — the hashes of individual database records are arranged in a tree structure, where the hash of the root node summarizes the entire database. Integrity is verified on each query by a computing a hash of the result (along with auxiliary information from the MHT) and comparing with the trusted digest. Security is reduced to the collision resistance property of hash functions. We refer the reader to Mykleton et al. [27] for an introduction to MHT, though § IV-B of this paper shows one standard scheme for using the MHT (B-tree variant) to verify integrity of key-value stores.

We find that enterprise applications generally have two key requirements for any additional security mechanism: 1) minimal changes, and 2) negligible performance overheads. However, all prior work on using MHT for ensuring integrity force a redesign of the client and server, and also incur significant performance overheads. First, the requirement that the client application store a trusted hash digest imposes changes in the application's source code, how that application is deployed, how its state is persisted, and how it is replicated and migrated across machines. Second, the MHT structure requires a logarithmic (in the size of database) number of hash computations for each read, with writes incurring an additional logarithmic number of hash computations and updates to the MHT; computing these hashes imposes significant performance penalty for large databases. Furthermore, since each write modifies the trusted digest, it incurs data conflicts at the root node of the MHT, and at several other nodes depending on the workload's access patterns; this limits concurrency. Our initial experiments and work by Arasu et al [11] measured overheads of at least 10x in throughput, which is steadily worsening as mainstream databases get faster.

We present VeritasDB, a key-value store with formal integrity guarantees, high performance, and a tiny trusted computing base (TCB). VeritasDB leverages modern trusted hardware platforms to achieve an order of magnitude improvement in throughput while also removing large parts of the system from the trusted computing base. Recognizing the problem of infrastructure attacks, processor vendors are now shipping CPUs with hardware primitives, such as Intel SGX enclaves [25], for isolating sensitive code and data within protected memory regions (current hardware limits to 96 MB) which are inaccessible to all other software running on the machine. In this new paradigm, enclaves are the only trusted components of a system. Furthermore, being an instruction set extension to x86 rather than a co-processor, SGX provides enclaves with native processor execution speeds, direct access to non-enclave memory, and multicore parallelism — we posit that these features are an ideal fit for integrity verification.

Keeping the MHT (we use a B-tree variant, hereon called *MB-tree*) in unprotected memory, VeritasDB uses an enclave to host the integrity checking code and necessary trusted state, which at the least includes the MB-tree's root node hash (digest). Since the enclave can directly access DRAM memory that hosts the MB-tree, we observe an improvement over prior work, which requires the integrity proof to be sent via network to the client. While this improvement is attributed entirely to hardware advances, VeritasDB includes additional optimizations based on caching, concurrency, and compression. First, we leverage the remaining protected memory (96 MB) to cache commonly accessed parts of the MB-tree, by developing novel caching algorithms based on finding heavy hitters (count-min-sketch [16]) and cuckoo hashing [19]. Second, we compress parts of the MB-tree, allowing VeritasDB to scale to larger databases. These optimizations allow VeritasDB to shave the overhead down to an average 2.8x in throughput and 2.5x in latency across standard YCSB benchmarks and Visa transaction workloads. Finally, to avail hardware parallelism, we shard the MB-tree and associated trusted state, which reduces contention and increases throughput. Not surprisingly, since integrity verification is compute-bound, SGX-enabled CPU parallelism helps VeritasDB close the throughput gap to within 5% of the baseline (insecure) system with no integrity checks. Our system achieves an order of magnitude improvement over prior work, which have overheads close to 10x in throughput. In developing these optimizations, VeritasDB addresses several novel challenges, such as ensuring safety against an attacker that is allowed to tamper with non-enclave memory (holding the MB-tree) at any time during an operation.

VeritasDB is implemented as a network proxy (implementing a standard key-value store protocol) that mediates all communication between the client(s) and the server, which is any off-the-shelf NoSQL database (Redis [8], Cassandra [4], etc.)—therefore, VeritasDB incurs no modification to either the client or the server. Since the server is untrusted, VeritasDB also protects against implementation bugs in the server, allowing developers additional freedom in selecting the NoSQL database — it is important to note that while integrity verification detects incorrect results, it cannot correct them (this would fall under disaster recovery). The proxy can be deployed either in the cloud or on a client machine, as long as SGX is available, making deployment flexible.

In summary, this paper makes the following contributions:

- We present a design and implementation (based on Merkle B-trees) for checking integrity of key-value stores, without incurring any modification to the client or the server. VeritasDB is released open source at [TBD].
- We improve throughput of integrity verification by using features of SGX processors to implement optimizations based on caching, concurrency, and compression. Furthermore, we prove that these optimizations are sound, even in an adversarial setting where the attacker can modify non-enclave memory and storage at any time during operation.
- We develop a secure recovery mechanism that prevents attackers from using crashes to exploit the application.
- We evaluate VeritasDB on standard YCSB benchmarks and Visa transaction workloads, and present the impact of these optimizations.

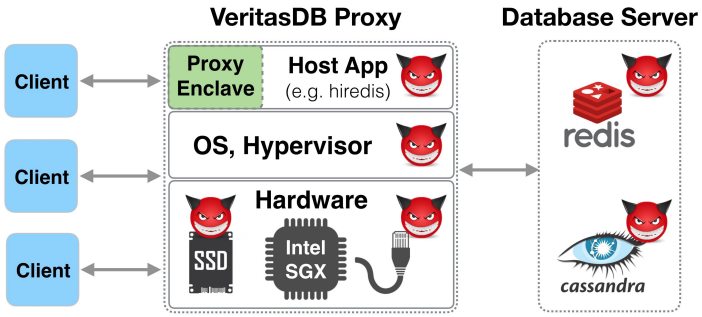

Fig. 1: Architecture and Threat Model.

II. System Model

A. System Architecture

VeritasDB's architecture (Figure 1) consists of a proxy and an unmodified NoSQL server (e.g. Redis); i.e., it extends the client–server architecture with a trusted proxy (protected using Intel SGX) that intercepts all of the client's requests and the server's responses in order to provide integrity. By mediating all interaction, the proxy is able to perform necessary bookkeeping to track versions of objects written for each key and to enforce freshness on all results sent back to the client. The database server can be hosted on any untrusted machine, and the proxy must be hosted any machine with Intel SGX CPU — since cloud providers are offering SGX machines, we co-locate the database and the proxy on the same machine for improved latency. The proxy consists of the following components: enclave code for performing the integrity checks, unprotected memory (containing a MB-tree), and enclave-resident state (for authenticating the MB-tree and other bookkeeping). The unprotected part of the proxy performs untrusted tasks, such as socket I/O with the clients and server.

VeritasDB's API provides the standard operations of a key-value store: $get(k)$, $put(k, v)$, $insert(k, v)$, and $delete(k)$. While the proxy exposes these operations, it does not implement them entirely but rather piggy-backs on the server (any mainstream off-the-shelf key-value store). This allows us to leverage modern advances in NoSQL databases with negligible effort, as nearly all of them (e.g. Redis, RocksDB, DynamoDB, etc.) support a superset of VeritasDB's API. Instead, the contribution of VeritasDB is the trusted proxy that efficiently checks integrity of all responses from the server, using the definition of integrity in § III — henceforth, we refer to VeritasDB and the proxy interchangeably.

The client establishes a secure TLS channel with the proxy to protect data in transit — the client uses SGX's remote attestation primitive [14] to ensure that the TLS session endpoint resides in a genuine VeritasDB proxy. The client is unmodified, and is only responsible for generating API requests, which require the protection of TLS to avoid tampering of API arguments. When using VeritasDB with multiple clients, the proxy orders all the requests, thus enforcing sequential consistency in addition to integrity (see § III).

B. Background on Intel SGX Enclaves

Hardware support for isolated execution enables development of applications that protect their code and data even while running on a compromised host. Recently, processor

vendors have started shipping CPUs with hardware primitives, such as Intel SGX[1] [25] and RISC-V Sanctum [17], to create memory regions that are isolated from all other software in the system. These protected memory regions are called enclaves, and they contain code and data that can only be accessed by code running within the enclave, i.e., the CPU monitors all memory accesses to ensure that no other software (including privileged OS and Hypervisor layers) can access the enclave's memory. A SGX enclave is launched by its host application, and it occupies a contiguous range of the host application's virtual address space and runs with user-level privileges (ring 3 in x86). The host application can invoke code inside an enclave via a statically defined entry-point, and the enclave code can transfer control back via an exit instruction in SGX. Additionally, the CPU implements instructions for performing hash-based measurement and attestation of the enclave's initial memory, thereby enabling the user to detect malicious modifications to the enclave's code or initial state. VeritasDB uses SGX enclaves to protect the trusted proxy's code and state.

Since the OS cannot be trusted to access enclave's memory safely, the CPU prevents enclave code from issuing system calls. However, the enclave can directly access the host application's memory (but not the other way around), which enables efficient I/O between the enclave and the external world — system calls must effectively be proxied by the hosting application. In VeritasDB, we demonstrate that direct low-latency access to untrusted memory (which stores the Merkle B-tree) from enclave code enables efficient integrity checking. That being said, current generation of SGX processors only provide 96 MB of protected memory (in comparison to several GBs required for the Merkle B-tree), and this paper demonstrates an efficient design and optimizations to overcome this limitation.

C. Threat Model

VeritasDB places no trust in the backend server. Since the clients are only generating requests, no trust is placed on them either. The only trusted component in VeritasDB is the proxy, which we harden using Intel SGX. Figure 1 illustrates all system components that are under the attacker's control.

Except the SGX CPU running the enclave program, the adversary may control all hardware in the host computer, including I/O peripherals such as the disk and network. We assume that the attacker is unable to physically open and manipulate the SGX CPU, and assume that SGX provides integrity guarantees to the enclave memory. The adversary may also control the privileged system software, including the OS, hypervisor, and BIOS layers, that the database server depends on. This capability can be realized by a rogue datacenter administrator with root access to the servers, or a remote attacker that has exploited a vulnerability in any part of the software stack. Such a powerful attacker can snapshot or tamper the state of the database in memory and/or disk, from underneath the server's software; the attacker may also tamper the server's code. The attacker also controls all I/O: it can create, modify, drop, and reorder any message between between the client, server, and the proxy. We do not defend against denial of service attacks; the server may choose to ignore all requests, and destroy the entire database.

[1]SGX is currently available on all Intel client-grade CPUs of 6th generation and above.

III. Problem Formulation

VeritasDB provides the standard operations of a key-value store: $get(k)$, $put(k, v)$, $insert(k, v)$, and $delete(k)$. A $get(k)$ operation outputs the value associated with key k; $put(k, v)$ updates the internal state to associate key k with value v; $insert(k, v)$ is used to create a new association for key k, and it fails if k already exists in the database; $delete(k)$ removes any associations for the key k. VeritasDB outputs one of the following results for each operation: (1) success, along with a value in the case of get operations; (2) integrity violation; (3) application error (indicating incorrect use of API, such as calling get on non-existent key); (4) failure (due to benign reasons, such as insufficient memory).

Though we don't list it as a contribution, we define integrity using formal logic to avoid ambiguities — we are unaware of any formalization of integrity for key-value stores. The following definitions are required to state the integrity property.

Definition 1: An interaction is a tuple $\langle op, res \rangle$, where op $\in$ { $get(k)$, $put(k, v)$, $insert(k, v)$, $delete(k)$ } is the requested operation and res is the result sent back to the client.

Definition 2: A session π between a client and VeritasDB is a sequence of interactions $[\langle op_0, res_0 \rangle, \ldots]$, in the order in which the operations were issued by the client. We let $\pi[i] = \langle op_i, res_i \rangle$ denote the i-th interaction within the session, where $\pi[i].op$ denotes op_i and $\pi[i].res$ denotes res_i. Furthermore, $|\pi|$ denotes the number of interactions in π.

We say that a session π satisfies integrity if the properties listed in Table I hold for each interaction $\pi[i]$, where $i \in \{0, \ldots, |\pi| - 1\}$. A key-value store must provide integrity in all sessions during its lifetime. One can observe from the definition in Table I that integrity has two main sub-properties: *authenticity* and *freshness*. Authenticity implies that a get operation returns a value that was previously written by the client, and not fabricated by the attacker. Authenticity is stated implicitly because all properties in Table I refer to prior interactions in the session, and we require that the results are a deterministic function of those previous interactions (and nothing else). Freshness implies that a get operation returns the latest value written by the client (thus preventing rollback attacks), and that the set of keys in the database at any time is a deterministic function of the most up-to-date history of client's operations (i.e., the attacker does not insert or delete keys). Freshness is stated explicitly in the property for get in Table I, which stipulates that the returned value must equal the most-recent put, and that the results of $insert$ and $delete$ operations are consistent with the latest account of interactions in that session — for instance, an $insert(k, v)$ only succeeds if that the key k does not exist. Further observe that error results also have authenticity and freshness requirements (see Table I).

Since integrity is a safety property, it ensures that when a result is produced, it is a correct one. Therefore, only successful results and application errors are assigned a property in Table I. On the other hand, no guarantee is given when the server fails to perform the operation, which would violate

availability[2] but not integrity — VeritasDB's proxy cannot verify error messages such as insufficient memory. In such cases, the client could retry the operation in future.

VeritasDB enforces these properties for each interaction, even in the presence of a buggy or compromised database server. However, it is important to note that VeritasDB cannot fix an incorrect response, and thus cannot ensure availability during an attack — in other words, a misbehaving server causes denial of service to the clients. When it detects an integrity violation, the proxy stops accepting new requests and terminates the session. Our logical formalization of integrity is aligned with the definitions in prior work [23]. While our definition is tailored for the single client setting, it can be easily extended to multiple clients since the proxy's operation is agnostic to the number of clients that it serves — in that case, the ordering is defined by the order in which operations arrive in the proxy's message queue, and the clients would enjoy a sequential consistency property [22], [11].

We do not consider confidentiality in this work, and let the application encrypt stored values. In practice, we observe that critical enterprise applications are commonly deployed with default encryption mechanisms (e.g. key management backed by hardware security modules), which we intend to preserve.

IV. Integrity Verification

This section describes VeritasDB's base mechanism; while it lays the groundwork for describing our contributions (§ V onwards), this section describes the use of several prior techniques and justifies design decisions in VeritasDB, but is not entirely novel by itself. As stated in § III, integrity is composed of authenticity and freshness guarantees, and VeritasDB uses the proxy to enforce both. As we discuss in § IV-A, authenticity is enforced via a cryptographic MAC (computed using a secret key), and freshness is enforced using protected state that keeps track of the latest version for each key. Furthermore, since this protected state may grow linearly with the size of the database size (specifically, the number of keys), and current SGX CPUs only provide fixed (relatively tiny) amount of protected memory, § IV-B describes a Merkle B-tree data structure, which allows storing state in untrusted non-enclave memory while still cryptographically protecting accesses to that state. Finally, we discuss in § IV-C how to ensure integrity in the event of crashes in the proxy or the database server, in which case we must defend against rollback attacks.

A. Basic Design

To issue database operations, the client first establishes a TLS channel with the proxy. The channel's remote endpoint terminates within the proxy enclave, hence the interactions are secure against network or OS-level attacks. The channel establishment uses an authenticated Diffie-Hellman key exchange, where the proxy's messages are signed using the SGX CPU's attestation primitive — the CPU computes a hash-based measurement of the proxy enclave's initial state (as it is being loaded into memory), and this hash is included in the signature, thus allowing the client to verify the genuineness of the proxy enclave. The client issues all operations over this secure channel, by serializing the opcode and operands, and then encrypting them using the symmetric channel key.

To perform the integrity checks, the proxy maintains the following state: 1) 128-bit HMAC key (called `hmac_key`) for authenticating server-bound values; 2) a map (called `present`) from keys to 64-bit counters for tracking latest version for each key present in the database; 3) a map (called `deleted`) from keys to 64-bit counters for tracking deleted keys (to prevent a subtle attack explained below; see *insert* operations). For the purpose of § IV-A, the reader can assume that all of the proxy's state is stored within enclave memory. We now describe the basic mechanism with the proxy for handling each operation.

Put Operations: The proxy performs some bookkeeping and transforms the arguments to a $put(k, v)$ request before contacting the server. First, the proxy throws a key_missing_error if the proxy does not maintain a binding for key k. Otherwise, it computes an HMAC to authenticate the payload, where the HMAC binds the payload (containing value v and incremented version) to the key k. The binding of payload to k ensures that an attacker cannot swap the payloads associated with any two different keys. The server may choose to deny the put request, in which case the failure message is propagated to the client; else, the proxy acknowledges the successful operation by incrementing `present`[k].

```
if (k in present) {
  tag := HMAC(hmac_key, k || present[k] + 1 || v);
  res := server.put(k, present[k] + 1 || v || tag);
  if (res == ok) { present[k] := present[k] + 1; }
  return res;
} else { return key_missing_error; }
```

Get Operations: The logic for *get* operations is effectively the dual of *put*. The proxy forwards the $get(k)$ request to the backend server if it finds a binding for key k; else, it throws a key_missing_error to the client. The payload returned by the server is checked[3] for authenticity using an HMAC, which is computed over the requested key k, version counter `ctr`, and the value `v`. Once the payload is deemed authentic (proving that it was previously emitted by the proxy), the proxy proceeds to check that the counter `ctr` matches the value in the local state `present`, thus guaranteeing freshness.

```
if (k in present) {
  res := server.get(k);
  if (res == fail) { return fail; }
  ctr || v || tag <- res; /* deconstruct payload */
  assert tag == HMAC(hmac_key, k || ctr || v);
  assert ctr == present[k];
  return (ok, v);
} else { return key_missing_error; }
```

Insert Operations: An $insert(k, v)$ operation is handled similarly to $put(k, v)$, except the proxy adds key k to the present map. While one might expect k's counter to initialize at 0, this would not defend against the following attack. Consider a session where the client deletes a key and later inserts it again, i.e., a session of the form [..., $insert(k, v)$, ..., $delete(k)$, ..., $insert(k, v')$, ...]. If the version in present were to reset to 0 on $insert(k, v')$, then the attacker can start supplying older values on *get* requests (i.e. v' for v) but still satisfy the integrity checks. To prevent

[2] We note that payment incentivizes cloud to provide availability, which we consider outside the scope of this study.

[3] A failed assert halts the proxy and denies future requests.

API $\pi[z].op$	Result $\pi[z].res$	Property
$insert(k, v)$	ok	$\neg(\exists x.\ x < z\ \wedge\ \pi[x] = \langle insert(k, v), \mathsf{ok}\rangle \wedge \forall y.\ x < y < z \Rightarrow \pi[y] \neq \langle delete(k), \mathsf{ok}\rangle)$
	key_present_error	$\exists x.\ x < z\ \wedge\ \pi[x] = \langle insert(k, v), \mathsf{ok}\rangle \wedge \forall y.\ x < y < z \Rightarrow \pi[y] \neq \langle delete(k), \mathsf{ok}\rangle$
$get(k)$	(ok,v)	$\exists x.\ x < z \wedge (\pi[x] = \langle put(k, v), \mathsf{ok}\rangle \vee \pi[x] = \langle insert(k, v), \mathsf{ok}\ \rangle) \wedge$ $(\forall y, v'.\ x < y < z \Rightarrow\ \pi[y] \neq \langle delete(k), \mathsf{ok}\ \rangle \wedge \pi[y] \neq \langle put(k, v'), \mathsf{ok}\rangle)$
	key_missing_error	$\neg(\exists x.\ x < z\ \wedge\ \pi[x] = \langle insert(k, v), \mathsf{ok}\rangle \wedge \forall y.\ x < y < z \Rightarrow \pi[y] \neq \langle delete(k), \mathsf{ok}\rangle)$
$put(k, v)$	ok	$\exists x.\ x < z\ \wedge\ \pi[x] = \langle insert(k, v), \mathsf{ok}\rangle \wedge \forall y.\ x < y < z \Rightarrow \pi[y] \neq \langle delete(k), \mathsf{ok}\rangle$
	key_missing_error	$\neg(\exists x.\ x < z\ \wedge\ \pi[x] = \langle insert(k, v), \mathsf{ok}\rangle \wedge \forall y.\ x < y < z \Rightarrow \pi[y] \neq \langle delete(k), \mathsf{ok}\rangle)$
$delete(k)$	ok	$\exists x.\ x < z\ \wedge\ \pi[x] = \langle insert(k, v), \mathsf{ok}\rangle \wedge \forall y.\ x < y < z \Rightarrow \pi[y] \neq \langle delete(k), \mathsf{ok}\rangle$
	key_missing_error	$\neg(\exists x.\ x < z\ \wedge\ \pi[x] = \langle insert(k, v), \mathsf{ok}\rangle \wedge \forall y.\ x < y < z \Rightarrow \pi[y] \neq \langle delete(k), \mathsf{ok}\rangle)$

TABLE I: Integrity property of VeritasDB, based on result types. The property guaranteed by an interaction is defined with respect to its position z in the session π. Unspecified result types have no guarantees.

this attack, we check in the `deleted` map whether the key k was previously inserted, and resume the counter if found.

```
if (k in present) { return key_present_error; }
else {
  ctr := k in deleted ? deleted[k] + 1 : 0;
  tag := HMAC(hmac_key, k || ctr || v);
  res := server.insert(k, ctr || v || tag);
  if (res==ok){
    present[k] := ctr;
    deleted.remove(k);
  }
  return res;
}
```

While our identification and solution for this attack is novel, we must point out that this attack is an aftermath of our design choice to use version counters to authenticate values. We now justify this design choice. Since the semantics of our operations treat each key independently, the proxy must store some metadata for each key to ensure freshness; i.e., linear storage (in the number of keys in the database) is unavoidable. VeritasDB stores a 64-bit version counter for each key. Alternatives include storing a collision-resistant digest, such as a cryptographic hash, of the stored value (which would require at least 256 bits per key for sufficient security), or storing the entire value itself (which is clearly wasteful duplication for large values). While storing a hash of the value would avoid the above attack, it significantly increases the storage requirement — instead, based on the observation that $delete$ operations are infrequent in large classes of workloads, we choose the counter-based design and manage a separate `deleted` map. In § VII, we present a scheme for periodically purging elements from the `deleted` map.

Delete Operations: The proxy simply forwards the delete command to the server. If the server deletes successfully, we remove the binding by deleting k from `present` and adding k to `deleted`, to prevent the attack explained above.

```
if (k in present) {
  res := server.delete(k);
  if (res == ok) {
    deleted[k] := present[k];
    present.remove(k);
  }
  return res;
} else { return key_missing_error; }
```

B. Merkle B-Tree and Operations

The security of VeritasDB rests on being able to perform the integrity checks securely, which necessitates storing the proxy's code and data in enclave memory. As described in § IV-A, the proxy's state includes the `present` and `deleted` maps, which requires 64 bits for each key. Hence, the proxy's state grows with the database size (number of keys), whereas SGX only offers fixed size enclave memory — 96 MB for current generation CPUs. For any reasonable size database, we quickly exhaust the enclave's memory.

A strawman solution is to piggy-back on SGX's secure paging which allows the enclave to address virtual memory that is larger than 96 MB, relying on the OS and CPU to co-operate together on page-level management of enclave memory. In SGX, on accessing an unmapped page, the OS applies a page-replacement policy and instructs the CPU to encrypt (using AES-GCM authenticated encryption) the 4KB page from enclave memory, swap it out to non-enclave memory, and decrypt the requested page before loading into the enclave's memory — note that this process is in addition to the virtual memory paging between DRAM and persistent storage. This mechanism preserves both integrity and authenticity of the enclave's state because the adversarial OS only chooses the evicted page but is unable to observe or tamper with the plaintext contents of that page. However, this mechanism has two notable drawbacks. First, paging incurs significant overhead as accessing even 1 bit incurs encryption and copying out of 4 KB, followed by decryption and copying another 4 KB — this problem is exacerbated on uniform workloads which don't exhibit temporal locality of access patterns. Second, while we can instantiate larger enclaves, this approach requires us to commit to the maximum size of the enclave statically, which would cause the proxy to crash once the database reaches a certain size.

Instead, a more practical solution is to employ an *authenticated data structure*, such as a Merkle-tree [26], [23], which stores a tiny digest in trusted memory and uses an untrusted medium (non-enclave memory) to hold the proxy's state — we refer the reader to [27] for an introduction to using Merkle Hash Trees. MB-tree offers logarithmic time update and retrieval operations. VeritasDB employs two Merkle B-trees (MB-trees), one each for `present` and `deleted`,

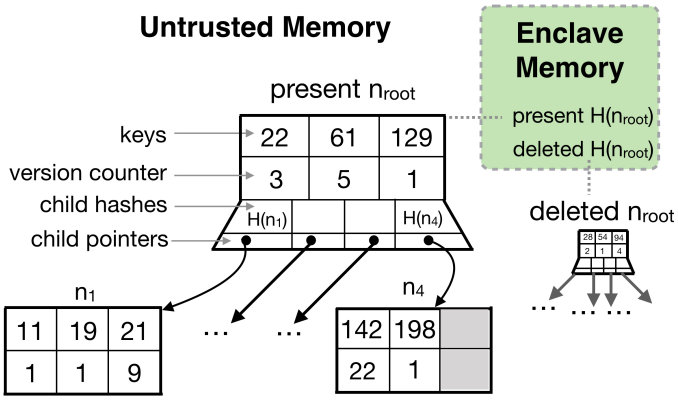

Fig. 2: Merkle B-Tree for authenticating `present` and `deleted` maps. Hash of root nodes are stored in enclave.

which use several novel performance optimizations based on Intel SGX primitives. Before diving into those optimizations (focus of § V, § VI, § VII), we use this section to discuss the specific application of MB-trees in VeritasDB.

Figure 2 illustrates the MB-trees structure for a sample database. Each intermediate node contains several keys (depending on the configured branching factor), a counter per key, pointers to child nodes, and a hash of each child node. As per the properties of a B-tree, the left (right) child relative to any key k holds those keys that are strictly less (greater) than k. A leaf node does not contain any child pointers or hashes. The entire tree is stored in unprotected memory, while the enclave stores the hash of the root node. A node's hash is computed using a SHA-256 function applied to the concatenation of the following values: keys, `present`$[k]$ bits, and hashes of child nodes. Note that we do not include the child pointers while computing the hash; the reason is that we intend the hash to be a concise digest of the contents of the subtree, independent of its physical location — we may even relocate parts of the tree for better memory allocation or while reconstructing the tree following a restart, but this should not affect the hash.

Integrity of any node's contents can be established by either storing that node's hash in trusted storage (as we do for the root node), or authentically deriving that node's expected hash by recursively computing child hashes along the path from the root node — see algorithm for *get* operations below. Each update to the tree (*put*, *insert*, and *delete* operations) forces modifications of child hashes along one or more paths in the tree, including an update to the root node hash stored in enclave memory. With the introduction of the MB-tree, we modify the integrity verification logic for each operation, as follows.

Get: The algorithms presented in § IV-A assume trusted access to `present` and `deleted` maps. Since this state is stored in the MB-tree in unprotected memory, we prepend the algorithm for $get(k)$ in § IV-A with the following steps to ensure authentic reads of that state.

```
trusted_hash := H(n_root); /* stored in enclave */
path := search(n_root, k); /* root to node with k */
for (i = 0; i < path.size() - 1; i++) {
  cp(n_local, path[i]); /* memcpy node to enclave */
  assert H(n_local) == trusted_hash; /*authenticate*/
  j := find(path[i+1], n_local); /* index of child */
  trusted_hash := n_local.child_hash[j];
}
j := find(k, n_local); /* index of k in keys */
/* use n_local.present[j] in place of  present[k] */
...
```

We iterate from the root node down to the node containing k, while authenticating each node to determine the trusted hash of the next child node — the trusted root node hash H(n_{root}) bootstraps this process. While this is mostly standard procedure for Merkle Trees, note that for each node along the path from the root node to the node containing k, this process copies the node's contents to enclave memory (to avoid a TOCTOU attack) and computes its SHA-256 hash; this path is logarithmic in the database size. This path verification adds significant overhead, which we measured to be roughly 10x in throughput for a modest size database holding 50 million keys (see Figure 6). A key contribution of this paper is a set of optimizations (presented in § IV-D) for bridging this gap to roughly 2.8x on average, and 1.05x using the parallel implementation.

Put, Insert, and Delete: Updates to the tree cause the updated hashes to propagate all the way to the root, modifying each node along the path. A $put(k, v)$ operation increments `present`$[k]$, $delete(k)$ adds k to `deleted`, while $insert(k, v)$ adds k to `present`. For these operations, we first search for the node containing k, and use the same logic as *get* (shown above) to attain authentic contents of that node — this is a prerequisite to recomputing the hash, as several fields within that node retain their values. Next, we update the node, which necessitates updating the hash of that node (in the parent), and recursively, the hashes of all nodes in the path up to the root. Hence, update operations require at least twice as much computation as *get* operations. We omit pseudocode due to space constraints.

An $insert(k, v)$ or $delete(k)$) operation creates a new entry in the `present` or `deleted` MB-tree, respectively. The new key is placed in an empty slot in an existing node or a newly allocated node, which incurs a change in the structure of the tree. In either case, we must authenticate a path in the tree to derive authentic contents of one or more nodes, and update the hashes (similar to *put* operations).

C. Persistence and Fault Tolerance

The VeritasDB proxy may crash at any point in time, either due to benign causes (such as insufficient resources) or attacker's operations (the proxy may run on untrusted infrastructure). If we do not persist state across restarts, then it allows the attacker to reload an older state of the system — specifically, the attacker loads an older copy of the backend database, and a corresponding snapshot of the proxy's memory. This would be an integrity violation, and VeritasDB defends against such attacks by using hardware-backed monotonic counters to ensure state continuity.

VeritasDB adopts Ariadne's scheme [33] for state continuity of enclave programs. We denote the hardware monotonic counter as `sgx.ctr`, and bind this counter to some data d by producing a package containing $d \parallel$ `HMAC(hmac_key,` `sgx.ctr` $\parallel d$), where `hmac_key` is a 128-bit HMAC key (from § IV-A) known only to the enclave. To persist the database state, the proxy takes the current hash of the root node, i.e., `trusted_hash`, and writes the package

HMAC(hmac_key, sgx.ctr + 1 ‖ trusted_hash) to disk, after which it increments sgx.ctr. The proxy the sends a "save" command to the backend database — most key-value storage systems, including Redis and RocksDB, provide an explicit command to force them to persist in-memory contents to disk. Should the proxy crash at any time in future, both the database and the proxy must be restored with the latest saved state, as any other state would cause an integrity violation in the proxy — on restart, the proxy checks that the trusted_hash is retrieved from a package with the correct counter value sgx.ctr. To restore the proxy's state, the proxy recovers the package from disk, initializes the MB-tree, and then creates additional two packages (with successive counter values, using the above steps) to avoid the dictionary and liveness attacks described in [33].

Prior works [24], [33] measure that incrementing monotonic counters on TPM devices requires upwards of 100 milliseconds, and the TPM devices often wear out after a few million writes. Therefore, VeritasDB cannot afford to persist state after each update operation, as doing so would drop the performance to fewer than 10 operations per second. Instead, similar to mainstream key-value storage systems, VeritasDB provides a *save*() API to trigger the above process. We leave it to the client application to determine when and how often to persist, where higher frequency leads to increased protection against rollback attacks (the attacker must at least supply the last saved state). Note that VeritasDB is agnostic to the implementation of monotonic counters; prior works have also proposed using distributed systems [24] to provide the abstraction of monotonic counters with statistical guarantees.

D. Performance Optimizations

In the remainder of this paper, we present optimizations based on caching, concurrency, and compression, which further reduce the overhead, in addition to the gains from SGX. Note that while these optimizations provide an order-of-magnitude speedup in practical workloads, they do not improve asymptotic behavior. Dwork et al. [18] proved that integrity verification (in an online setting) with bounded trusted state requires logarithmic amount of computation. VeritasDB has the same asymptotic complexity, because SGX protects fixed size memory (96 MB on current CPUs).

V. Caching in VeritasDB

Since SGX offers significantly more protected memory than what we need for storing code pages of VeritasDB, we reserve a portion of enclave memory to cache a subset of the MB-tree's state, in addition to the mandatory storage of the root node's hash H(n_{root}). By caching these bits, we avoid computing hashes to authenticate reads of that state.

VeritasDB employs two caches: 1) hash-cache for caching child hashes of MB-tree nodes, and 2) value-cache for caching the version counters within the present map. Figure 3 illustrates these two cache structures. The rationale for having two caches is that they are suited to different types of workloads, and together enable VeritasDB to perform well on a wider variety of workloads — we observe empirically (§ X-D) that skewed workloads benefit more from the value-cache, while the hash-cache benefits workloads with higher

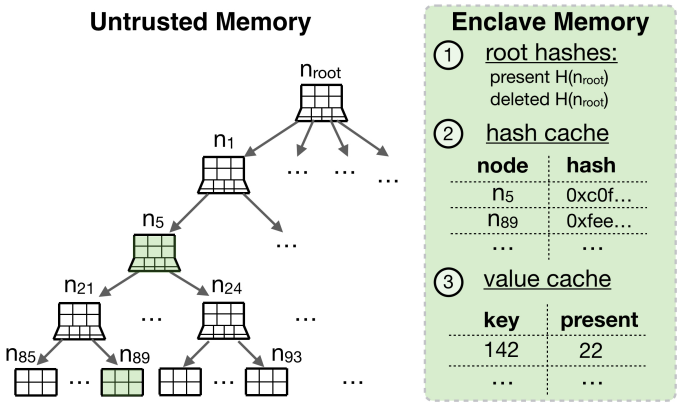

Fig. 3: Caches optimize reads of MB-tree's state.

entropy distributions. We leave it to future work to tune the size of the two caches based on runtime monitoring of the access patterns.

A. Hash Cache

The hash-cache caches hashes of a subset of nodes within the MB-tree, and it is indexed by the memory address of a MB-tree node. Since the cache is located within the enclave, a cached node's hash is authentic, which obviates computing hashes of its predecessors in order to authenticate that node's contents — experiments confirm that the hash-cache speeds up read operations significantly (§ X-D). However, update operations force recomputations of hashes recursively along the path from the deepest modified node up to the root node, so they benefit less from caching; should any node along this path exist in the hash-cache, its hash must also be updated to ensure correctness. VeritasDB's implementation of hash-cache adopts cuckoo hashing for desirable attributes such as efficient, constant-time lookups and updates (we fix maximum 20 cuckoo relocations), and constant space requirements (we disable resizing). This section develops a novel caching algorithm to manage the hash-cache, though we contend that it can be improved in future work.

1) Counting Accesses: Assuming that past behavior is an indication of future (which no cache can predict), an ideal cache of size n stores the top n most-frequently accessed items, aggregated over a sliding time window to ensure adaptability to recent access patterns. Not surprisingly, an impossibility result stipulates that no algorithm can solve this problem in an online setting using sub-linear space [10], thereby urging us to introduce approximation — this is acceptable in VeritasDB's setting because caching can withstand inclusion of a few light hitters. Through empirical evaluation using enterprise workloads (described in § X), we find that a cache with LRU (least recently used) policy suffers from thrashing, and can be improved by tracking historical access patterns. To that end, VeritasDB implements an algorithm based on count-min-sketch [16] (CMS), where we modify the vanilla algorithm to find approximate heavy hitters within a sliding time window, while achieving constant space, predictable timing, and high performance. It works as follows.

To count accesses to nodes of the MB-tree, VeritasDB allocates a CMS table, which is a 2-D array of cells. CMS provides a space-accuracy tradeoff, and we find that, for reasonable

expected error, we need a CMS table with 16,000 columns and 5 rows (corresponding to pairwise independent uniform hash functions), which requires approximately 3 MB — while this is another parameter in VeritasDB, we empirically observe these values to provide nearly optimal performance across all workloads, so we do not report performance measurements for varying values of these parameters (due to space constraints). Due to hash collisions in mapping a large universe of MB-tree nodes (several millions) to a smaller CMS table, the CMS scheme necessarily over-approximates; hence, we inherit its trait of admitting a few light hitters within the cache. VeritasDB introduces a simple modification: each cell in the CMS table stores not a single counter but a vector of N counters, used as follows. The desired sliding window (say 10,000 operations) is divided up into N (say 10) equi-width epochs, where an epoch denotes a contiguous subsequence of interactions in a session. Each counter in the vector maintains the access count for the duration of an epoch, as in vanilla CMS. Epoch transitions move over to the next counter in the vector, whose value is initialized to 0 — since a sliding window has N epochs, an epoch transition at position N-1 wraps around to the counter at position 0.

To determine whether a node at address a should be included in the cache, we compute a point query on the sketch:

$$\mathsf{count}(a) \doteq \min_{0 \le j < 5} \sum_{n=0}^{N-1} \mathsf{CMS}[j][h_j(a)][n]$$

Should count(a) be larger than a threshold, the hash of node a is deemed worthy of inclusion in the hash-cache — we will shortly discuss how to set this threshold. Updates to the CMS table occur on each $get(k)$ operation, where for each node on the path to the node containing k, we 1) compute the 5 hashes of k, each mapping to one of the 16,000 columns, and 2) increment the counter corresponding to the current epoch in each of those 5 cells. The update to the CMS table uses the following logic, which is prepended to the algorithm for get operations detailed in § IV-B.

```
trusted_hash := H(n_root) /* stored in enclave */
path := search(n_root, k)
for (i = path.size() - 1; i >= 0; i--) {
  if (hash_cache.has(path[i])) { /* optimization */
    trusted_hash := hash_cache.get(path[i])
  }
  for (j = 0; j < 5; j++) { /* i = dist from root */
    CMS[j][h_j(path[i])][clock % N] += i
  }
  if (count(path[i]) >= threshold) {
    hash_cache.insert(path[i])
  }
}
...
```

Here, clock refers to the global clock incremented by 1 on each interaction in the session; h_j denotes one of 5 uniform hash functions; each CMS cell is a vector indexed using the ring counter (clock modulo N). Contrary to vanilla CMS, which increments each count by 1 for each hit, we increment the counter by weight i, which is the distance of the node path[i] from the root node. This has a simple explanation: on accessing that path in a future operation, caching the hash of a node at distance i from the root saves i hash computations, and therefore must be prioritized over nodes with depth less than i. In other words, there is no benefit from caching a parent node when all of its children are cached, and hence we need a mechanism to prioritize the children nodes over the parent node. Note that this does not imply that we only end up caching leaf nodes in the MB-tree, since a parent node accrues hits from all paths originating from that node — as expected, in a workload with uniform distribution of keys, we find that this scheme converges to (i.e., has maximum likelihood of) a state where the hash-cache contains all nodes at a median height of the MB-tree.

While a proposed extension to CMS, namely the ECM-sketch [30], also provides a sliding-window sketch with provable error guarantees, it has higher expected error and incurs a wider distribution of running times for queries and updates. Not only is our algorithm simpler to implement, but it also achieves a tighter error bound by allowing for variations in the length of the sliding window, which we argue to be acceptable for our setting — we solve a different problem as our point query computes an aggregate over a sliding window whose length is somewhere between N-1 and N epochs, depending on the duration of the current epoch. This is acceptable because we are not necessarily interested in estimating the count over a fixed sliding window. Rather, we wish to rank nodes in order of frequency, or more specifically, determine whether a node's frequency is above a threshold — despite a varying-length time window, we provide fairness to all MB-tree nodes when we compute point queries. Note that our scheme achieves the same error bound as vanilla CMS applied over an input stream of events containing the previous N - 1 epochs followed by the requests in the current epoch. Meanwhile ECM's probabilistic error guarantee is for a fixed window of size N, where in addition to the over-approximation from hash collisions, ECM incurs error from the sliding window counter estimation, where the weight of the oldest bucket (or epoch) is halved to minimize the expected error.

2) Setting Threshold: Computing the sketch synopsis is only half the battle; a caching algorithm must also determine an appropriate threshold for deciding inclusion, i.e., for evaluating the check `count(path[i]) ≥ threshold`. We implement a heuristic scheme that dynamically tunes the threshold based on the load factor (i.e., cache occupancy in percentage), best described with the help of a sample execution illustrated in Figure 4. Every few epochs (say 10), we re-tune the threshold as follows:

- if load factor is below a certain fraction (say 90%), we reduce the threshold using a multiplicative factor (say 5/6);
- if load factor is above 97%, we increase the threshold by a multiplicative factor (say 4/3), and also flush half of the occupants with below-average count.

If the cache is too vacant, it indicates that our threshold is too high, so we correct course by decreasing it by a multiplicative factor. On the other hand, we increase the threshold when the cache is highly occupied; while we prefer our cache to be occupied completely at all times, the intention here is to ensure that heavy hitters are the chosen occupants, which is why we also flush the lighter half of the hitters to make room. We choose 97%, as opposed to 100%, because cuckoo hashing (with 4 hash functions and 1 cell per hash bucket) reaches 97% achievable load before the probability of irresolvable collisions (which cause unsuccessful insertions)

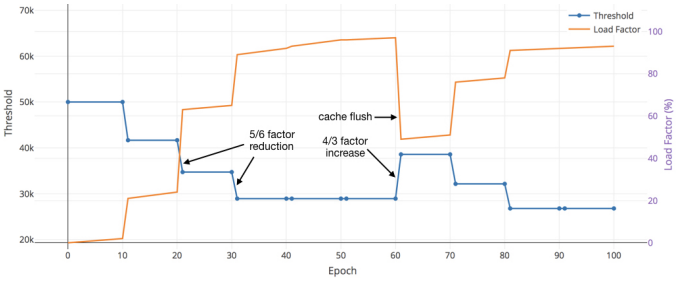

Fig. 4: Threshold tuning during a run of YCSB-C

rises dramatically — using graph theory, one derives that for a cuckoo hash table of size 64K cells, at 97% load factor, we have successful insertions (with fixed amount of work) with 99% probability [19]. We can also use finer grained adjustments at smaller occupancy increments, but we defer this exploration to future work.

B. Value Cache

We also reserve some protected enclave memory to cache entries in the `present` map, which obviates computing any hashes to authenticate their values — we call this the value-cache, and it is indexed by the key. Since the hash-cache converges to near-optimal caching strategy on uniform workloads, we design the value-cache to exploit more skewed workloads, where a larger fraction of the operations target fewer set of keys. To that end, we employ the LRU cache replacement policy, where the cache is built using a cuckoo hash table with the following configuration: 4 hash functions, bins of size 1, maximum of 20 cuckoo relocations, and at least 64K cells (fixed throughout runtime). Cuckoo hashing allows for constant lookup time in the worst case, and using 4 (pairwise-independent, uniform, and efficient) hash functions enable higher achievable loads (fewer irresolvable conflicts) without computing too many extra hashes — note that each hash computation is followed a comparison of the stored key with the search key, which is the more significant cost. We modify the vanilla cuckoo hashing to use an approximate LRU replacement policy in lieu of periodically resizing the table, which is not viable for fixed size enclaves.

Each cache entry stores the key k, the `present`$[k]$ counter, and the timestamp of last access. On each $get(k)$ operation, we first check if the value-cache already contains k by searching within the 4 potential slots, as is standard for cuckoo hashing. If found, we update the timestamp (of last access) on that cache entry; otherwise, we insert k (and `present`$[k]$) by relocating existing elements to make room (up to a maximum of 20 relocations). If an empty slot is not found within 20 cuckoo relocations, then the LRU entry is vacated to make room for k. Since the LRU entry may be any of the 20 relocated entries, we first simulate the 20 relocations before actually performing them — simulation is cheap as computing the uniform hash function is inexpensive; we use SpookyHash [3] which uses 2 cycles / byte on x86-64. Once the LRU item (entry with lowest timestamp) is identified, the relocations are performed until we reach the LRU item to be evicted — we call our scheme approximate LRU since the LRU item is chosen from a subset of the entire cache. To track time, we use the global clock mentioned previously. VeritasDB flushes the cache when this counter overflows, which is expected to occur every 12 hours approximately (at 100K operations per second). While storing a 32 bit timestamp for each cache entry may seem wasteful, the dominating consumer of space in the cache is the indexing key k.

Depending on the benchmark, we find that the value-cache can provide up to 5x increase in throughput.

VI. Concurrency in VeritasDB

The integrity verification logic in VeritasDB is CPU bound, as it computes a sequence of hashes in addition to an HMAC. Fortunately, SGX enables hardware parallelism by allowing multiple CPU threads within an enclave, and this section describes their use in VeritasDB.

A. Sharding

As a potential solution, we could instantiate a message queue to accept API requests from the clients, and launch multiple threads that concurrently process requests from the queue. Since there is a read-write and write-write conflict at the root of the MB-tree — update operations modify the root and all operations read the root — we can introduce locking on MB-tree nodes to avoid data races. However, we forego this approach due to the performance impact of locking, and the complexity of dealing with structural changes in the MB-tree during $insert$ and $delete$ operations.

Instead, we adopt the simpler approach of sharding the `present` map into separate maps, and build a separate MB-tree for each shard. VeritasDB uses a dedicated integrity verification thread for each shard. Assignment of keys to shards is done using a fast hash function (e.g. crc32). This design effectively produces in a Merkle forest, i.e., a set of disjoint trees. More importantly, there is no shared state across threads, removing the need for any form of locking. VeritasDB implements asynchronous I/O mechanism within the enclave to allow the threads to operate without context switching from enclave mode to untrusted code. We illustrate this design in Figure 5.

VeritasDB uses a message queue in untrusted memory for accepting encrypted client requests — this is typically managed by a distributed messaging library such as zeromq [2]. Within the proxy enclave, we launch a dedicated I/O thread (marked io in Figure 5) to handle all I/O to/from the enclave, and a set of worker threads (marked $t_1, \ldots, t_n$ in Figure 5, one for each shard) to perform integrity checks and issue commands to the server. Recall that the client establishes a TLS channel with the proxy enclave, and uses the session key to encrypt all interactions. The I/O thread fetches encrypted requests from untrusted memory, decrypts them using the TLS session key, and copies them into the in-enclave input buffer of one of the worker threads (based on the output of `crc32` function applied to the key being operated on). Each worker thread t_i fetches the next request from its dedicated buffer, processes it, and optionally interacts with the untrusted server as follows. The worker thread serializes the operands to non-enclave memory and busy-waits for the server's response to be populated at a designated location in non-enclave memory; a dedicated thread in the untrusted host application interacts with the server (using a standard client library such as hiredis [5] for Redis [8]). We do not implement asynchronous buffered I/O

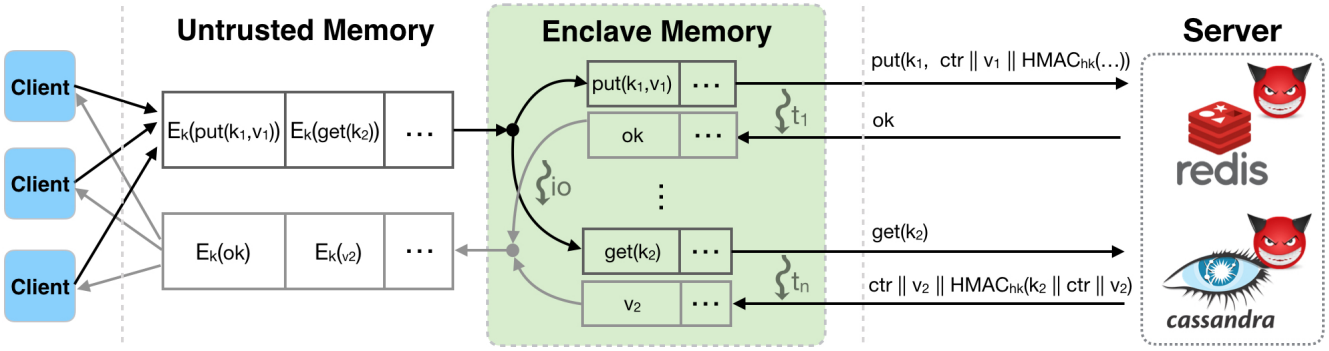

Fig. 5: Proxy architecture for concurrent query processing.

for worker-server interaction because the worker's computation depends on the response from the server, and the MB-tree state produced at the completion of this request affects future requests. In addition to exploiting hardware parallelism, our use of asynchronous I/O between client and worker threads avoids context switches across the enclave boundary — [35] shows that these context switches can take upto 7000 cycles (about 35x slower than a system call).

It is important to note that sharding does not suppress the caching optimization discussed earlier, as each shard receives an equal share of the hash-cache and the value-cache. While prior work, namely Concerto [11], briefly mentions sharding as a potential method for concurrency, they do not pursue it because skewed workloads (where a few keys account for most of the operations) can create a load imbalance. While this is a valid concern in Concerto's design, we find that our caches overcome this potential problem from skewed workloads.

We observe in practice that sharding (with other optimizations also enabled) achieves significant improvements in all workloads, especially since integrity verification in VeritasDB is compute bound. By availing enough CPU threads (typically 2-4), we can reduce the overhead to within 5% of the insecure system with no integrity (see § X-E).

B. Exitless MB-Tree Operations

While sharding-based concurrency allows VeritasDB to extract higher throughput from available hardware parallelism (compared to single-threaded implementation), we empirically observe that naïve use of SGX causes prohibitively impractical performance. Specifically, since updates to the MB-tree require allocation from the host application's memory, a standard approach (as suggested by the SGX programming guide [1]) is to transfer control from the enclave to the host application, which will then perform the necessary memory allocations and updates to the MB-tree structure. As mentioned above, these context switches require upwards of 7000 CPU cycles [35], so VeritasDB implements exitless operations, as follows.

On enclave creation, the host application allocates a large region of memory (in the order of GBs) from its heap and passes the base address of this region to the proxy enclave. Since the host is untrusted, the enclave checks that the region is entirely contained in non-enclave memory — without this check, an attacker may cause the enclave to tamper with its internal data structures, or worse, perform remote code execution. The enclave hosts the malloc and free code, which traverses and updates the free list by directly accessing untrusted memory. A standard implementation of malloc stores critical metadata, such as the size of the allocated memory buffer, adjacent to that buffer. By controlling untrusted memory, the attacker may overwrite this metadata and cause the memory management to fail, and compromise integrity of the MB-tree state. However, such operations provide the attacker with no more capability than modifying the MB-tree state directly — in other words, any tampering of this metadata can be modeled as a tampering of the MB-tree structure, which is detected by the integrity verification logic in VeritasDB.

In general, the use of exitless operations to speed up enclave computation was first proposed by [29], though our concrete use case and techniques are novel.

VII. Compression Optimizations in VeritasDB

A. Compressing MB-Tree

Keys within a MB-tree node are stored in order and thus share bits starting from the most significant bit. For instance, the keys in the root node in Figure 2 share all but the lowest 8 bits. This creates the opportunity for an empirical optimization. Instead of duplicating the other 248 bits (where maximum key length is set to 64 bytes) three times, VeritasDB stores these common bits once. This form of compression reduces memory footprint by an average of 30% across our benchmarks.

B. Purging Deleted Keys

Recall that to prevent rollback attacks, VeritasDB records all the deleted keys in a separate MB-tree called `deleted`, which would monotonically grow throughout the application's history as keys are deleted from the database. VeritasDB implements the following scheme to manage this space wasteage. In addition to a counter for each `present` key, VeritasDB maintains a global counter (common for all keys), and appends this global counter to the argument to `HMAC` computation (see § IV-A) — this binds the stored value to the global counter, in addition to the key-specific version counter. Periodically, when the size of the deleted set grows beyond a configurable threshold, VeritasDB empties the set and increments the global counter. At this point, the proxy must rewrite all stored keys in the database server to use the new global counter.

VIII. Security Analysis

We first argue that our base design, as presented in § IV, is safe against our threat model where the attacker controls

the entire software stack on both the proxy and the database server, except for the SGX enclave. Specifically, we show that VeritasDB thwarts the following attacks.

- **Tampering attack against the MB-Tree.** Assuming the standard collision-resistance property of hash functions (we use 256-bit SHA-2), any modification of contents within a MB-tree node or any modification to the structure (child pointers) of the MB-tree results in a different root node hash. Therefore, modifications to the MB-tree are detected by the proxy enclave. That being said, without sufficient safeguards, the proxy may also be vulnerable to the following time-of-check-to-time-of-use (TOCTOU) attack. Since the MB-tree is stored in untrusted memory, and the enclave directly accesses that memory, an attacker may modify a MB-tree node after its hash is verified, convincing the enclave to trust future accesses to that node's contents. Instead, we design our enclave logic to either copy the node's contents to enclave memory (if multiple accesses are performed) or ensure that the same values are never accessed twice.
- **Tampering attack against the database server.** On retrieving any key from the database server, the proxy enclave verifies the HMAC tag, where successful verification implies that the stored value was generated by the proxy enclave, has the correct version, and is bound to the correct key. From the standard HMAC assumption of unforgeability under chosen message attacks, we derive that arbitrary tampering by the database server would result in an incorrect HMAC tag. Furthermore, since the HMAC tag computation uses the version counter in the MB-tree, rollback attacks (i.e., supplying older values bound to the same key) would also be detected — the unforgeability assumption implies that an attacker cannot forge a valid HMAC tag which uses an older value and the latest version counter.
- **Restart attack against the proxy enclave.** By periodically checkpointing state bound to monotonic counters, VeritasDB prevents an attacker from using termination as a mechanism to perform rollback attacks. However, as mentioned before, due to limitations in current NVRAM technology, monotonic counters have slow updates, and we leave it to the client application to determine the frequency of checkpointing process. Should these operations no longer become a bottleneck in future, VeritasDB can perform them on each update operation by default.

Next, we informally prove that our optimizations are sound.

Lemma 1: Hash cache is safe.

Proof: Since the hash-cache is indexed by the memory address of the MB-tree node, we must show that tampering of either the MB-tree structure or the node contents does not compromise the integrity verification logic. First, the logic for $get(k)$ operations traverses the MB-tree to search for a path to the node containing k. Should this search fail, we ensure the correctness of key-missing-error response by verifying the entire path to the node (or two nodes) containing a key greater than and less than k — this verification does not use the hash-cache, and its security can be reduced to the security of the base design. Alternatively, if k is found and a node along the MB-tree path is cached, then its hash is a unique digest summarizing the contents of all nodes in the subtree (following from the collision-resistance property) — while the attacker may swap blocks of memory in untrusted memory, it is forced to preserve the ordering of keys and the values of the version counters. ■

Lemma 2: Value cache is safe.

Proof: Since the value-cache is indexed by the key and is updated whenever there is a relevant modification to the MB-tree, an attacker cannot tamper the binding between a key and its version counter. ■

Lemma 3: Sharding of MB-tree is safe.

Proof: Sharding decomposes a MB-tree into a set of disjoint MB-trees, and uses the enclave to disjointly maintain the trusted state for each MB-tree. Since the integrity verification logic is replicated for each MB-tree, our security proof carries over. ■

Lemma 4: Management of deleted keys is safe.

Proof: We must show that a version counter is never reused to write two different values for the same key, as that would allow the attacker to swap them while still passing the integrity verification checks. We enforce this constraint by 1) retaining version counters of deleted keys and using higher version counts upon re-insert, and 2) incrementing global version counter upon purging the set of deleted keys. ■

IX. Implementation

VeritasDB can be deployed on any machine with an SGX CPU, and can be co-located with the client or the server. The proxy's code consists of an untrusted component (that interacts with the clients and the database server) and a trusted component (that implements the integrity checks).

The trusted enclave implements the integrity verification logic described in § IV and § IV-D, and a minimal TLS layer to establish a secure channel with the client. The integrity verification logic is implemented using 2500 lines of C code, which we plan to formally verify. The TLS channel is established using a Diffie-Hellman key exchange that is authenticated using Intel SGX's remote attestation primitive [14], which produces a hardware-signed quote containing the enclave's identity, that the clients can verify to ensure the genuineness of the proxy enclave. The TLS layer is implemented using 300 lines of C code and linked with the Intel SGX SDK [7].

The untrusted component (also called the host application) implements I/O between the clients and the proxy enclave, and between the proxy enclave and the database server. The client-proxy I/O is implemented entirely using ZeroMQ [2], a mainstream asynchronous messaging library. The proxy-server I/O is implemented using a thin wrapper (roughly 100 lines of code) over a database client library — we use hiredis [5] for interacting with Redis [8], DataStax driver [6] for interacting with Cassandra [4], etc. Not only does running the untrusted component outside of an enclave reduce the TCB, but we are also forced to do so because socket-based communication invokes system calls, and the OS cannot be trusted to modify enclave memory.

X. Evaluation

Using Visa transaction workloads and standard benchmarks, we evaluate VeritasDB by studying these questions:

- How much overhead in space (i.e., memory footprint) and time (i.e., throughput and latency) does integrity checking add, in comparison to a mainstream, off-the-shelf system with no integrity checking?
- How does overhead vary with the size of the database?
- How much improvement do the caching and concurrency optimizations provide? How does performance vary with the size of the caches and number of threads?

Experiment Setup: VeritasDB is implemented as a network proxy on the path between the client and the untrusted server. Since including network latencies while measuring VeritasDB's latency can suppress the measured overhead, we run the client, proxy, and server on the same machine (with sufficient hardware parallelism). Furthermore, while measuring throughput, we use a multiple clients to flood the proxy with requests so as to prevent the proxy from stalling — each client is synchronous, i.e., it waits for the response from the proxy before issuing the next operation.

We evaluate VeritasDB using the YCSB benchmarks [15] and Visa transaction workloads. We test with the following YCSB workloads: A (50% gets and 50% puts), B (95% gets and 5% puts), and C (100% gets), and D (95% gets and 5% inserts). For each workload, we first run a a setup phase where we insert between 5 million and 50 million keys into the database. During measurement, each operation accesses one of these keys, where the keys are generated using either the scrambled Zipfian (α = 0.95) distribution (denoted by suffix Zipf) or Uniform distribution (denoted by suffix Unif). For workload D, the insert operations generate new keys that do not exist in the database. The Visa workload captures access patterns encountered while processing over two billion payment transactions, where each transaction incurs one *get* operation to a NoSQL store.

All experiments are performed 10 times and then averaged. We use a machine with Ubuntu 16.04 LTS OS, running on Intel E3-1240 v5 CPU @ 3.50GHz, 32 GB DDR3 RAM, and 240 GB SSD. We anticipate even better performance once SGX is available in server-grade hardware.

A. Comparison with Prior Work

Before studying how individual optimizations impact performance of VeritasDB, we summarize the improvement we achieve over prior work, by enabling all optimizations: caching, compression, and concurrency (with 4 CPU threads). We setup the prior work system by implementing a separate proxy process that sends the MB-tree path (proof) to the client. The results are illustrated in Figure 6.

B. Measuring Space and Time Overhead

We first study the question of how much overhead in time (throughput, latency) and space (memory footprint) VeritasDB adds, compared to an off-the-shelf key-value store. For this experiment, we use a setup that enables all optimizations except concurrency, i.e., we use single CPU thread and enable compression and caching optimizations. This is because integrity verification is compute-bound, and adding threads trivially helps VeritasDB close the throughput performance gap (see § X-E); instead, disabling concurrency helps us study the performance overheads in fair light, showcasing both our algorithmic improvements and advantages of SGX hardware to the problem of integrity verification.

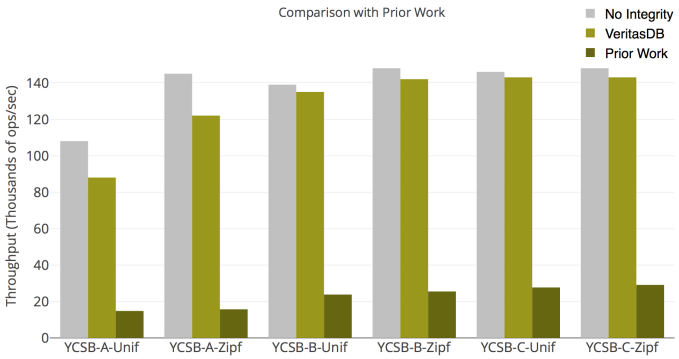

Fig. 6: Improvement over prior work: Throughput measurement, using RocksDB backend and 50 million keys.

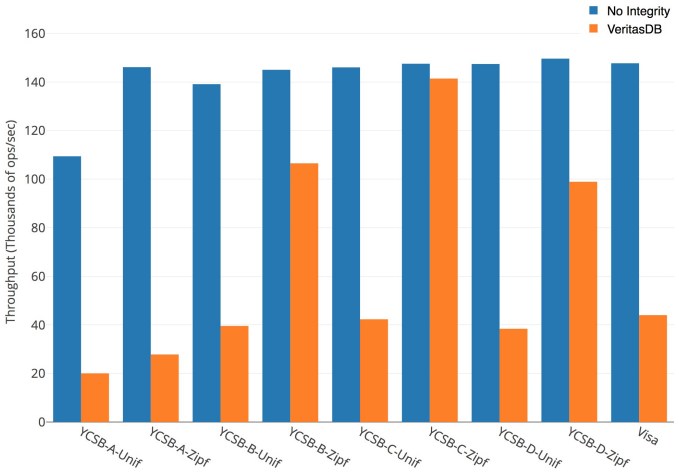

Fig. 7: Throughput on YCSB benchmarks, using 1 thread, RocksDB backend server, 5 million keys.

Throughput and Latency: Figure 7 illustrates the throughput measurements on standard workloads, and Figure 8 illustrates the latency overheads on micro-benchmarks. We fix the database size to 5 million keys, where each key is 64 bytes and each value is of length between 16 bytes and 256 bytes (chosen randomly). While VeritasDB can work with any NoSQL server, we use RocksDB [9] in these experiments.

Not surprisingly, we find that get operations execute significantly faster than others, leading to lower latency and higher throughputs — recall that hash computations are only required to ensure the integrity of the read nodes along the path from the deepest cached node to the node holding the requested key. All other operations modify the contents of one or more nodes in the MB-tree, forcing an additional sequence of hash computations from the deepest modified node back up the path to the root node. This phenomena is evident in Figure 7, where, as the proportion of get operations increases from 50% in YCSB-A to 100% YCSB-C, so does the throughput; Figure 8 also supports this phenomena as get operations have significantly lower latency.

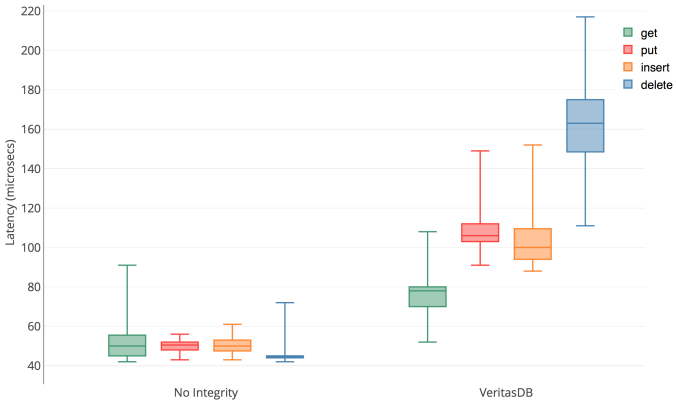

Fig. 8: Latency on micro-benchmarks, using 1 thread, RocksDB backend server, and 5 million keys.

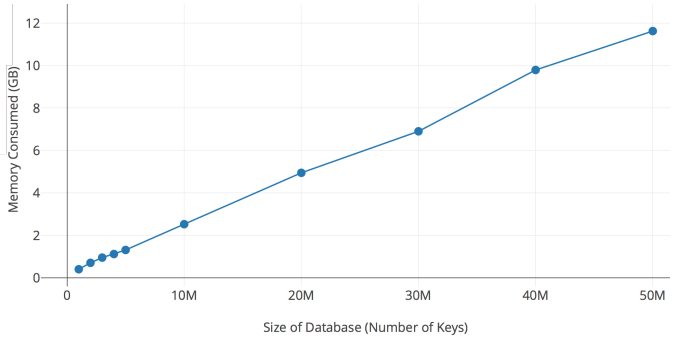

Fig. 9: Proxy memory usage by number of keys.

Not surprisingly, VeritasDB provides significantly higher throughput with Zipfian workloads because a tiny set of keys account for a large fraction of the accesses, which lends well to caching. Since 50 keys account for over 99% cumulative probability in this distribution, we find that either (even a tiny) value-cache stores the state for requested key (requiring no hash), or the hash-cache stores the hash of an immediate node (requiring 1 hash). Since the effect of caching is suppressed in update operations — update requires logarithmic number of hash computations, regardless of caching — this phenomenon is especially visible in YCSB-C-Zipf.

Memory Footprint: Characterizing the memory requirements is crucial because VeritasDB's throughput drops sharply (on uniform workloads) once the footprint exceeds available DRAM, most likely due to page thrashing. The MB-tree stores a 64-bit counter for each key in the database, along with a linear number of SHA-256 hashes and pointers to various MB-tree nodes. Therefore, the size of the MB-tree grows linearly with the number of keys in the database, which we measure empirically in Figure 9. Note that the memory footprint here only captures the size of untrusted memory, which holds the entire MB-tree. The space allocated to trusted memory is constant (at most 96 MB in modern SGX processors), and is dwarfed by the size of untrusted memory. We find that the footprint is lowered by 30%, across all database sizes and workloads, due to the compression optimization (§ VII).

C. Throughput with Varying Database Size

Since the size of the MB-tree grows with the database size, we study its negative impact on VeritasDB's throughput

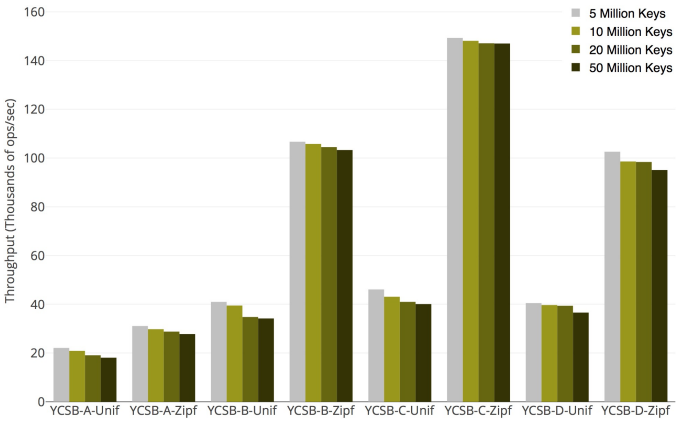

Fig. 10: Throughput with 1 thread for varying number of keys, using a RocksDB backend server.

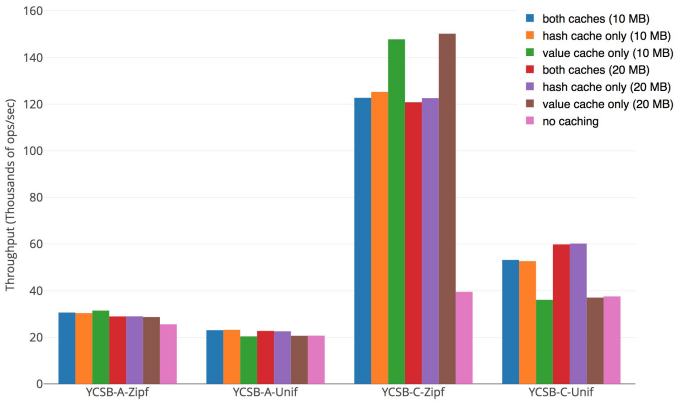

Fig. 11: Throughput with various cache sizes, using 1 thread, RocksDB server, and 5 million keys.

and illustrate our findings in Figure 10. We find that the decline in throughput from 5 to 50 million keys is only about 7% on average across all benchmarks, mainly owing to the slow growth of the B-tree structure — the rate of increase in height is exponentially lower at larger database size. For instance, with maximum branching factor of 9, we find that the MB-tree only grows in height by 4 when growing from 5 to 50 million keys, which results in only a few extra hash computations. While this may seem to be a small branching factor compared to a typical B-tree data structure, recall that VeritasDB stores this B-tree in memory as opposed to disk, and therefore is not incentivized by hardware to have larger branching factors; Furthermore, we empirically discover that while larger branching factors lead to a shorter tree, each node becomes wider, causing the hash computations to do a lot more wasted computation per operation — maximum branching factor of 9 was a sweet spot across each workload, but we omit this plot for space reasons.

As mentioned in § X-B, once the MB-tree exceeds the DRAM size (128 GB RAM stores MB-tree for over 500 million keys), performance drops to about 100 operations per second on uniform workloads; meanwhile Zipfian workloads remains unaffected, likely due to OS' caching of pages.

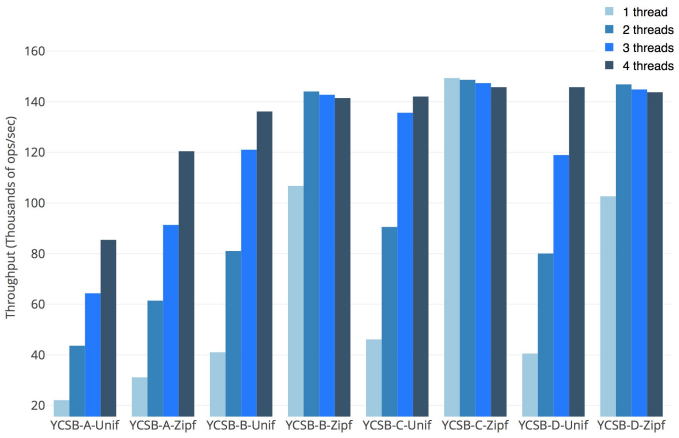

Fig. 12: Throughput varying number of threads, using RocksDB backend server and 5 million keys.

D. Measuring Impact of Caching

We measure the impact of caching by selectively disabling the hash-cache, the value-cache, or both, while enabling all other optimizations (except concurrency, for the same reason as § X-B). We also experiment with two different cache sizes: 10 MB vs. 20 MB per cache. Figure Figure 11 illustrates our findings on YCSB-A (update-heavy) and YCSB-C (read-heavy) benchmarks. As mentioned in § V, update operations have two phases: the first recovers authentic contents of MB-tree nodes and is similar to a *get*, while the latter modifies the MB-tree — this latter phase performs roughly equivalent work with or without caching, hence, *get* operations benefit more compared to *put*. This is observable in YCSB-C. In general, Zipfian workloads benefit more from caching; however, even uniform workloads show increase of 50% in throughput, especially due to the hash-cache.

E. Measuring Impact of Concurrency

Since integrity verification is compute-bound, we can close the performance gap with hardware parallelism. SGX allows all CPU cores on the processor to operate in enclave mode, and we leverage this feature in our concurrent design from § VI. Figure 12 illustrates how we achieve nearly the same performance as the baseline system (without integrity verification) by throwing enough CPU cores (worker threads) at the problem. Beyond a certain number of cores, the throughput becomes bound by the backend server, rather than the integrity checks. Furthermore, Figure 12 illustrates minor overhead from multi-threaded scheduling, as throughput isn't multiplied by the same factor as the ratio of number of threads — this overhead would have been even more severe had our experimental setup run out of hardware parallelism. Overall, we find that the parallel implementation of VeritasDB has a 5% overhead over the insecure system.

XI. Related Work

There is a large body of literature and systems built for verifying integrity of an outsourced database [21], [23], most of which use authenticated data structures. While several of these designs rely on variants of Merkle trees, the focus was either on security or on asymptotic costs, rather than performance on practical workloads—papers with implementation reported at most a few thousand operations per second, which is orders of magnitude lower than modern NoSQL databases. Additionally, many prior systems also required changes to both the client and the server, as the server produces a proof of correctness to be verified by the client; instead VeritasDB incurs no change to either.

Our work is inspired by Concerto [11], which addressed the concurrency limitations and computational cost of using Merkle trees by proposing deferred (batched) verification based on verified memory [13]. While this amortized costs (allowing them to achieve close to 500K operations per second on some workloads), we find that several classes of applications (especially payment transactions, where tampering poses high risk) require online verification.

The literature of authenticated data structures can be roughly organized along three dimensions. First, the types of data supported determine the integrity verification logic, where the data type ranges from large objects (files) to individual items in a data store. Second, the types of queries supported can range from key–value (point) queries, to partial, and to full SQL queries. Third, the integrity-verification mechanism can vary from purely cryptographic approaches to trusted-hardware approaches.

File System vs Database Integrity: Iris [32], an authenticated file system, reported throughput of 260 MB/sec with integrity of both file contents and metadata and with dynamic proofs of retrievability. Athos [20] authenticated an outsourced file system—not only the contents of each file, but also the directory structure—using authenticated skip lists and trees as building blocks, while reporting a modest overhead of 1.2x for writes and 2x for reads. We observe that the challenges with integrity verification of NoSQL stores are vastly different than file systems—Athos experimented with a file system with nearly 80K files of size 1.22 MB on average, and while this is acceptable for file systems, typical key-value stores host a much larger set of keys (typically several millions), which incurs larger overheads in checking integrity. File integrity has also been addressed using space-efficient techniques, such as entropy based integrity [28], but with the drawback of requiring linear time updates.

Key–Value vs SQL Integrity: IntegriDB [34] shows how a subset of SQL could support integrity verification with a limited throughput of 0.1 operations per second. A separate definition of integrity, transactional integrity, was addressed by Jain et al. [21], where the server has to prove that each transaction runs in a database state that is consistent with all transactions preceeding it. While they demonstrated maximum performance of few thousand operations per second for a commercial database (Oracle), we note that their approach to integrity was optimized for writes and offered no speedups to (and potentially even slows down) reads.

Cryptographic vs Trusted-Hardware Integrity: As an alternative to Merkle trees, some proposals use digital signatures attached to individual data items, but freshness is either not addressed or challenging to achieve without adding significant co-ordination between the clients and the servers. Li et al. [23] constructed a variant of Merkle hash tree over the data and used digital signature to add authentication.

VeritasDB is not the first database system to leverage trusted hardware. CorrectDB [12] used an IBM 4764 co-processor to provide authentication for SQL queries. While they also used authenticated data structures based on Merkle B+-trees, performance was limited owing to practical limitations of the co-processor (high latency link with DRAM) and the complexity of handling SQL queries.

XII. Conclusion

We designed a trustworthy proxy (called VeritasDB) to a key-value server that guarantees integrity to the clients, even in the presence of exploits or bugs in the server. The key takeaway is that while the standard approach of authenticating data from the untrusted server using a Merkle hash tree results in orders of magnitude reduction in throughput, recently developed trusted hardware (such as Intel SGX) provide larger amounts of protected memory and hardware parallelism, which benefits verification using Merkle hash trees without loss of security. We implement several optimizations (based on caching, compression, and concurrency) to achieve 10x improvement in throughput over past work.

References

[1] Intel software guard extensions developer guide. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf.

[2] ZeroMQ: distributed messaging. http://zeromq.org/.

[3] Spookyhash: a 128-bit noncryptographic hash. http://burtleburtle.net/bob/hash/spooky.html, 2012.

[4] Apache Cassandra. https://github.com/apache/cassandra, commit 1b82de8c9fe62cf78f07cf54fe32b561058eebe5.

[5] Hiredis. https://github.com/redis/hiredis, commit 3d8709d19d7fa67d203a33c969e69f0f1a4eab02.

[6] Datastax C/C++ driver for Apache Cassandra. https://github.com/datastax/cpp-driver, commit 3d88847002f33e8b236f52e8eb2a9f842394e758.

[7] Intel SGX SDK for Linux. https://github.com/intel/linux-sgx, commit 813960dbdd86b88b509b2946dbaa023e0ae8b1b9.

[8] Redis. https://github.com/antirez/redis, commit 813960dbdd86b88b509b2946dbaa023e0ae8b1b9.

[9] RocksDB: A persistent key-value store for Flash and RAM storage. https://github.com/facebook/rocksdb, commit dfbe52e099d0bfd7f917ca2e571a899bf6793ec1.

[10] C. Aggarwal. *Data Streams: Models and Algorithms*. Springer-Verlag, New York, NY, USA, 2007.

[11] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *Proc. 2017 ACM International Conference on Management of Data*, pages 251–266, New York, NY, USA, 2017. ACM.

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

[13] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2-3):225–244, 1994.

[14] E. Brickell and J. Li. Enhanced privacy id from bilinear pairing. Cryptology ePrint Archive, Report 2009/095, 2009. https://eprint.iacr.org/2009/095.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SoCC'10)*, pages 143–154, New York, NY, USA, 2010. ACM.

[16] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Algorithms*, 55(1):58–75, Apr. 2005.

[17] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proc. 25th USENIX Security Symposium*, pages 857–874, Austin, TX, 2016. USENIX Association.

[18] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *Proc. 6th Theory of Cryptography Conference (TCC'09)*, volume 5444 of *Lecture Notes in Computer Science*, pages 503–520. Springer, Mar. 2009.

[19] U. Erlingsson, M. Manasse, and F. McSherry. A cool and practical alternative to traditional hash tables. In *Proc. 7th Workshop on Distributed Data and Structures*, Santa Clara, CA, January 2006.

[20] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *Proc. International Conference on Information Security*, pages 80–96. Springer, 2008.

[21] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *Proc. 2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 529–540, April 2013.

[22] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Transactions on Computers*, (9):690–691, 1979.

[23] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proc. 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 121–132, New York, NY, USA, 2006. ACM.

[24] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1289–1306, Vancouver, BC, 2017. USENIX Association.

[25] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proc. 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, 2013.

[26] R. C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO'87)*, pages 369–378, London, UK, UK, 1988. Springer-Verlag.

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

[28] A. Oprea, M. K. Reiter, K. Yang, et al. Space-efficient block storage integrity. In *NDSS*, 2005.

[29] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 238–253, New York, NY, USA, 2017. ACM.

[30] O. Papapetrou, M. Garofalakis, and A. Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *Proc. VLDB Endowment*, 5(10):992–1003, June 2012.

[31] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proc. 2010 ACM Workshop on Cloud Computing Security Workshop (CCSW'10)*, pages 19–30, New York, NY, USA, 2010. ACM.

[32] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proc. 28th Annual Computer Security Applications Conference*, pages 229–238. ACM, 2012.

[33] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 875–892, Austin, TX, 2016. USENIX Association.

[34] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *Proc. 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pages 1480–1491, New York, NY, USA, 2015. ACM.

[35] C. Zhao, D. Saifuding, H. Tian, Y. Zhang, and C. Xing. On the performance of Intel SGX. In *Proc. of the 2016 13th Web Information Systems and Applications Conference (WISA)*. IEEE.