VeritasDB: High Throughput Key-Value Store with Integrity

Rohit Sinha
Visa Research
rohit.sinha@visa.com

Mihai Christodorescu
Visa Research
mihai.christodorescu@visa.com

ABSTRACT

While businesses shift their databases to the cloud, they continue to depend on them to operate correctly. Alarmingly, cloud services constantly 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 30x in throughput) and place additional responsibility on clients.

We present VeritasDB, a key-value store that guarantees data 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 unmodified database server, which can be any off-the-shelf database engine (e.g., Redis, RocksDB, Apache Cassandra). The proxy transforms each client request before forwarding it to the server and checks the correctness of the server's response before forwarding it to the client.

To ensure the proxy is trusted, we use the protections of modern trusted hardware platforms, such as Intel SGX, to host the proxy's code and trusted state, thus completely eliminating trust on the cloud provider. To maintain high performance in VeritasDB while scaling to large databases, we design an authenticated Merkle B+-tree that leverages features of SGX (modest amount of protected RAM, direct access to large unprotected RAM, 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 down to 1.05x.

1. 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. The possibility of one rogue administrator accessing and tampering sensitive data poses significant threat, one that cannot be fully mitigated by encrypting and/or signing data at rest. Moreover, remote attackers constantly try to exploit the infrastructure software layers of the cloud platform, including the privileged OS and hypervisor, to gain privileged access to the victim's databases. A cloud user (or database client) desires the security guarantee that queries return values that are consistent with the history of interaction between that client and the database server.

To that end, there is a large volume of work on database integrity verification [9, 10, 28, 18, 19, 27], 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 in trusted storage—the hashes of individual database records (key-value pairs) are arranged in a tree structure, where the hash of the root node forms a concise summary of 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 the hash function. We refer the reader to Mykleton et al. [24] for a concise introduction to MHT, though § 4.2 of this paper is a standalone account of how the MHT (B+-tree variant) enables verifying integrity of key-value stores.

Not only does this approach force redesign of the client and server, it incurs significant performance overheads. First, 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. 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 which paths of the MHT are accessed. This limits concurrency, which is problematic for a compute-heavy task. Our initial experiments and work by Arasu et al [9] measured overheads of up to 30x 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 [22], 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 in larger DRAM (albeit unprotected), and multicore parallelism — 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 immediately observe an average 3x improvement in throughput 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, we implement 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 [14]) and cuckoo hashing [17]. 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. This is already an order of magnitude improvement over prior work, which have overheads close to 30x in throughput. Finally, to avail hardware parallelism, we use sharding of 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.

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 [7], Cassandra [3], 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 server — it is important to note that while integrity verification detects incorrect results, it cannot correct them (this would fall under fault tolerance). 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 leveraging features of SGX processors to implement novel optimizations based on caching, concurrency, and compression.
- We evaluate VeritasDB on standard YCSB benchmarks and Visa transaction workloads, and present the impact of these optimizations.

2. OVERVIEW

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 book-keeping to track versions of objects written per key and to enforce freshness on all results sent back to the client. The server can be hosted on any untrusted machine, and the proxy must be hosted within a SGX process on the server machine. The proxy consists of the following components: enclave code

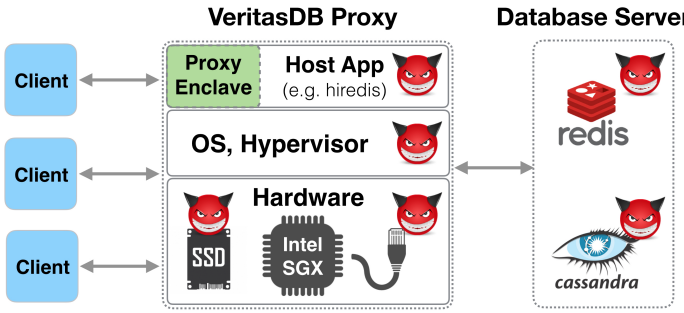

Figure 1: Architecture and Threat Model.

for performing the integrity checks, unprotected memory (containing a Merkle B+-tree), and enclave-resident state (for authenticating the Merkle-tree and other bookkeeping). The unprotected part of the proxy performs untrusted tasks, such as socket communication with the server and buffering of requests from multiple clients—the bulk of this code is implemented as a database driver (e.g., the hiredis library [4] for communicating with Redis [7], etc.).

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 § 3—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 [12] 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 a sequential consistency property in addition to integrity (more on this in § 3).

2.1 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] [22] and RISC-V Sanctum [15], 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, occupying a contiguous range of the host application's virtual address space and running with user-level privileges (ring 3 in x86). The host application can invoke code inside an enclave via statically defined entry-point, and

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

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 of RAM required for the Merkle B+-tree), and this paper demonstrates an efficient design and optimizations to overcome this limitation.

2.2 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 that runs the enclave program, the adversary may fully 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 protected enclave memory. The adversary may also control 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 the attacker can destroy the entire database.

3. 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 (since the integrity property is well-understood), 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 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 1 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 1 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 1 refer to prior interactions in the session, and integrity requires 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 1, 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 1).

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 1. 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 back-end database server. However, it is important to note that VeritasDB cannot fix an incorrect response, and therefore 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 (e.g. tampering to the database), the proxy stops accepting new requests and terminates the session. Therefore, integrity forms a safety property: VeritasDB never returns an incorrect result to the client, but the system may fail to make progress (i.e., liveness is not guaranteed). Our logical formalization of integrity is aligned with the definitions in prior work [21]. 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

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

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 1: 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.

message queue, and the clients enjoy a sequential consistency property [20, 9].

4. INTEGRITY VERIFICATION

This section describes VeritasDB's core mechanism (built on SGX protections) and justifies its design decisions; while it lays the groundwork for describing our contributions (§ 5 and beyond), this section combines existing ideas and is not entirely novel by itself. As stated in § 3, integrity is composed of authenticity and freshness guarantees, and VeritasDB uses the proxy to enforce both. We enforce authenticity using a cryptographic MAC, and freshness using enclave-resident state for keeping track of the latest version of each key-value binding — this is the focus of § 4.1). Furthermore, since the proxy enclave's state grows linearly with the size of the database size (specifically, the number of keys), and current SGX CPUs only provide bounded protected memory, § 4.2 describes an authenticated Merkle B+-tree data structure for protected access to this state.

4.1 Basic Design

To commence a session with the proxy, the client first establishes a TLS channel whose remote endpoint terminates within the proxy enclave, i.e., the interactions are secure against attacks on the network or the host OS. The channel establishment protocol 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 *get*, *put*, *insert*, and *delete* operations over this TLS channel, by serializing the API opcode and operands, and then encrypting them using the AES-GCM cipher. VeritasDB piggybacks on the server, which is assumed to support these operations as well (true for Redis [7], RocksDB [8], Cassandra [3], etc.).

To perform the integrity checks, the proxy maintains the following state: 1) 128-bit AES-GCM key for encrypting communication with the client; 2) 128-bit HMAC key (called `hmac_key`) for authenticating server-bound values; 3) a map (called `version`) from keys to 64-bit counters for tracking latest version for each key; 4) a map (called `present`) from keys to boolean values for tracking deleted keys (needed to prevent a subtle attack explained later).

For the purpose of § 4.1, the reader can assume that all of the proxy's code and is stored within enclave memory (alternative discussed in § 4.2). Storing 64-bit version counters in `version` leads to constant space overhead for each key in the database. Alternatives for ensuring freshness include storing a pre-image resistant digest, such as a cryptographic hash, of the stored value (which would require at least 256 bits for sufficient security), or storing the entire value itself (which is clearly wasteful duplication). We use the present map, in lieu of simply adding / removing entries from the version map, to avoid an attack explained below in the section explaining *insert* operations. We now describe the integrity checks and the proxy-server interaction below.

Put Operations. The proxy performs some book-keeping and transforms the arguments to a $put(k, v)$ request before forwarding the request to 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 version[k].

```
if (present[k]) {
  tag := HMAC(hmac_key, k || version[k] + 1 || v);
  res := server.put(k, version[k] + 1 || v || tag);
  if (res == ok) { version[k] := version[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 version, thus guaranteeing freshness.

```
if (present[k]) {
  res := server.get(k);
  if (res == fail) { return fail; }
  ctr || v || tag <- res; /* deconstruct payload */
  /* failure of following assertions causes halt */
  assert tag == HMAC(hmac_key, k || ctr || v);
  assert ctr == version[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 creates a binding for key k in the version and present maps. While one might expect that a new binding should initialize with the version counter 0, this approach 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 $[\ldots, insert(k, v), \ldots, delete(k), \ldots, insert(k, v'), \ldots]$. If the version were to reset to 0 on $insert(k, v')$, then the attacker can start supplying older values from the session on *get* requests but still satisfy the integrity checks. To prevent this attack, we check whether the key k was previously inserted, and resume the counter; this is ascertained by checking membership of k in version, but with present[k] set to false.

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

Delete Operations. The proxy simply forwards the delete command to the server. If the server deletes successfully, we remove the binding by setting present[k] to false, yet retaining the counter version[k] for the reason explained above.

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

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

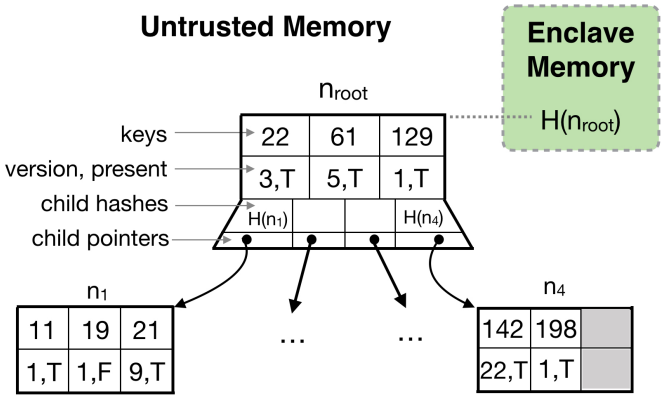

Figure 2: Merkle B+-Tree for authenticating version and present. Hash of root node is stored in enclave.

4.2 Merkle B+-Tree and Operations

The security of VeritasDB rests on being able to perform the integrity checks securely, which necessitates protected memory to store the proxy's code and data. As described in § 4.1, the proxy's state includes the version and present maps, which requires 65 bits for each key: 1 bit for the present flag, and 64 bits for the counter, though its bitwidth may be adjusted. Hence, the proxy's state grows with the database size (more accurately, the number of keys), while SGX CPUs only offer fixed size enclave memory — 96 MB for current generation CPUs. For any database of reasonable size, we would quickly exhaust the enclave's memory.

A strawman solution is to piggy-back on SGX's secure paging which allows for larger than 96 MB enclaves, relying on the OS and CPU to co-operate together on page-level management of enclave memory. Here, on accessing an unmapped page, the CPU automatically encrypts (using AES-GCM authenticated encryption) a 4KB page from enclave memory, swaps it out to non-enclave memory, and decrypts the requested page before loading into the enclave — note that this is orthogonal to the virtual memory paging between DRAM and persistent storage. This preserves both integrity and authenticity of the proxy's state because the adversarial OS is unable to observe or tamper the plaintext contents of those pages. However, this approach 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 [23, 21], 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 [24] for an introduction to using Merkle Hash Trees for providing integrity in outsourced databases. VeritasDB employs a Merkle B+-tree (MB-tree from here on), which offers logarithmic time update and retrieval operations, and implements novel performance optimizations that take advantage of Intel SGX features. Before diving into those optimizations (focus of § 5), we use this section to discuss the specific application of a MB-tree in VeritasDB.

Figure 2 illustrates the MB-tree structure for a sample database. Each intermediate node contains several keys (depending on

the configured branching factor), a version counter and present bit 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, version and present 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, 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 checks as follows.

Get. The algorithms presented in § 4.1 assume trusted access to version[k] and present[k]. Since this state is stored in the MB-tree in unprotected memory, we prepend the algorithm for $get(k)$ in § 4.1 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 */
version[k] := n_local.version[j] /* authenticated */
present[k] := n_local.present[j] /* authenticated */
...
```

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 $\mathsf{H}(n_{root})$ 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 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 10 million keys. A key contribution of this paper is a set of optimizations (presented in § 5) for bridging this gap to roughly 2.8x on average, and 1.05x with 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 version[k] while a $delete(k)$ operation updates present[k]. For both 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 to the root. Hence, for any given key, update operations require at least twice as much computation as *get* operations. We omit pseudocode due to space constraints.

An $insert(k, v)$ operation creates a new entry in the MB-tree, which is either placed in an empty slot in an existing node or placed in a newly allocated node — the latter causes 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 after the insert (similar to *put* and *delete* operations).

5. OPTIMIZATIONS

VeritasDB witnesses a significant performance improvement by using the SGX architecture, which provides direct access to untrusted memory (for storing the MB-tree) from the integrity-verifying enclave code. Prior systems that sent all nodes along a path in the MB-tree from the untrusted server to the verifying client over the network had upwards of 30x overhead for modest-size databases [9]. In contrast, on a direct implementation of our scheme from § 4.2, we measure an average 10x overhead in throughput (see "no caching" in Figure 10), which is already a 3x reduction in overhead compared to past systems.

We present three optimizations, caching, concurrency, and compression, which provide overhead reductions in addition to the gains from using SGX. Note that while these optimizations provide an order-of-magnitude speedup in practical workloads, they do not improve the worst case. Dwork et al. [16] 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 Intel CPUs).

5.1 Caching

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 $\mathsf{H}(\mathsf{n}_{\mathsf{root}})$. By caching these bits from the MB-tree, 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 version and present bits. 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 (§ 7.3) that skewed workloads benefit more from the value-cache, while the hash-cache benefits workloads with higher entropy distributions. We leave it to future work to tune the size of the two caches based on runtime monitoring of the access patterns. Figure 3 illustrates these two cache structures.

5.1.1 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

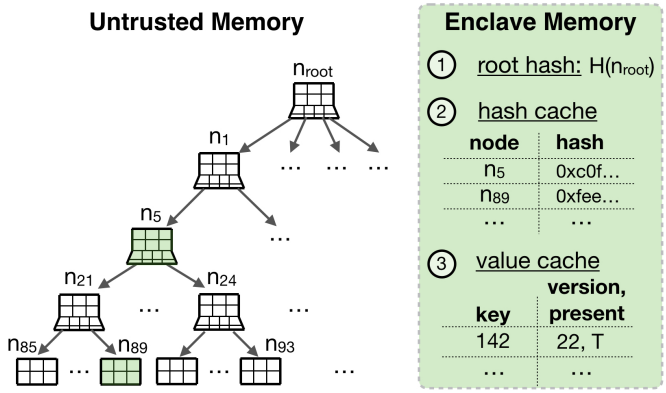

Figure 3: Caches optimize reads of MB-tree's state.

up read operations significantly (§ 7.3). 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.

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, thereby urging us to introduce approximation — this is acceptable in VeritasDB's setting because caching can withstand inclusion of a few light hitters. VeritasDB implements an algorithm based on count-min-sketch [14] (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. 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 § 4.2.

```
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 [26], also provides a sliding-window count-min-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 slightly different problem because our point query computes an aggregate over a sliding window whose length is somewhere between N-1 and N epochs, based 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,

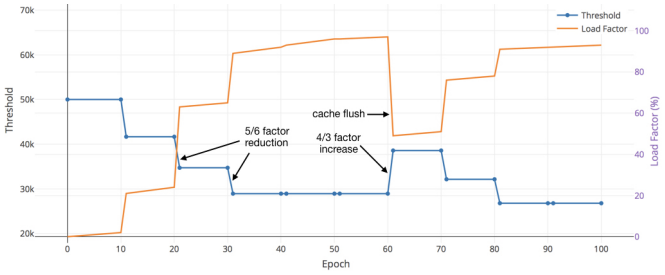

Figure 4: Threshold tuning during a run of YCSB-C

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.

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]) $\geq$ 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) 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 [17]. We can also use finer grained adjustments at smaller occupancy increments, but we defer this exploration to future work.

5.1.2 Value Cache

We also reserve some protected enclave memory to cache entries in the version and present maps, 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, version[k] and present[k] bits, 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 version[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 [1] 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, which can be up to 64 bytes in VeritasDB.

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

5.2 Concurrency

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.

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 version and present maps 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.

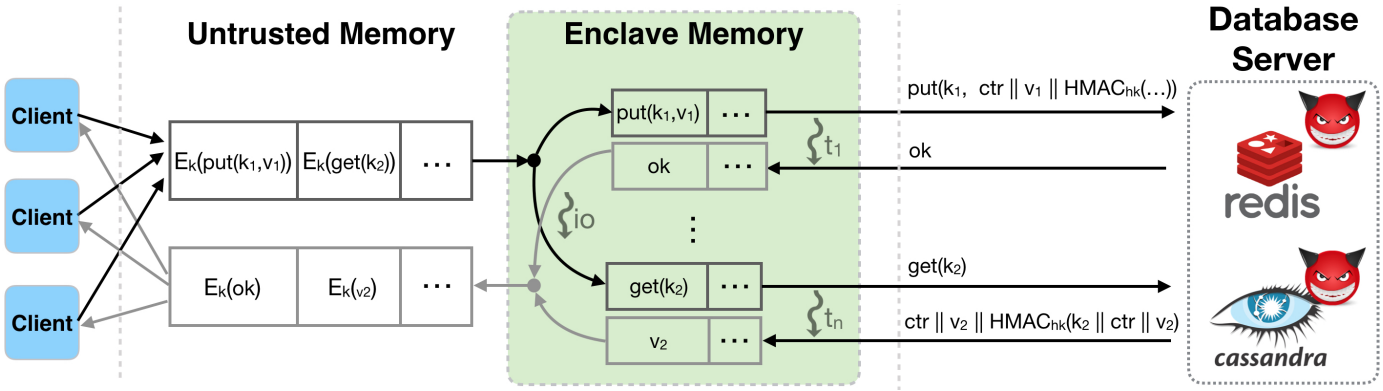

Figure 5: Proxy architecture for concurrent query processing.

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 [4] for Redis [7]). We do not implement asynchronous buffered I/O 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 — [30] 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 Concerto 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 § 7.4).

5.3 Compression

B+-trees require that keys are stored in order within a node. This creates an empirical phenomenon where keys within a MB-tree node have several bits in common, starting from the most significant bit. For instance, the keys in the root node in Figure 2 share all but the lowest 8 bits. Instead of duplicating the other 248 bits (where maximum key length is set to 64 bytes) 3 times, VeritasDB stores these common bits once. This form of compression reduces memory footprint by an average of 30% across our benchmarks.

6. 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 § 4 and § 5, and a minimal TLS layer to establish a secure channel with the client. The integrity verification logic is implemented using 4000 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 [12], 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 [6].

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 [4] for interacting with Redis [7], DataStax driver [5] for interacting with Cassandra [3], 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.

7. 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 [13] 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 distribution ($\alpha = 0.99$) or Uniform distribution; 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 payment transactions during a period of several weeks, amassing over two billion transactions — each transaction incurs one *get* operation to a NoSQL store. All experiments are performed 10 times and then averaged.

All experiments are performed on a machine with Ubuntu 16.04 LTS, running on Intel E3-1240 v5 CPU @ 3.50GHz (with 4 physical, 8 logical cores), 32 GB DDR3 RAM, and 240 GB SSD. We anticipate even better performance from VeritasDB once SGX is available in server-grade hardware.

7.1 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 § 7.4); 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.

Throughput and Latency. Figure 6 illustrates the throughput measurements across all YCSB and Visa workloads, and Figure 7 illustrates the latency overheads on micro-benchmarks. For each YCSB workload, we experiment with both the scrambled Zipfian distribution (suffix Zipf in Figure 6) and the Uniform distribution (suffix Unif). 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).

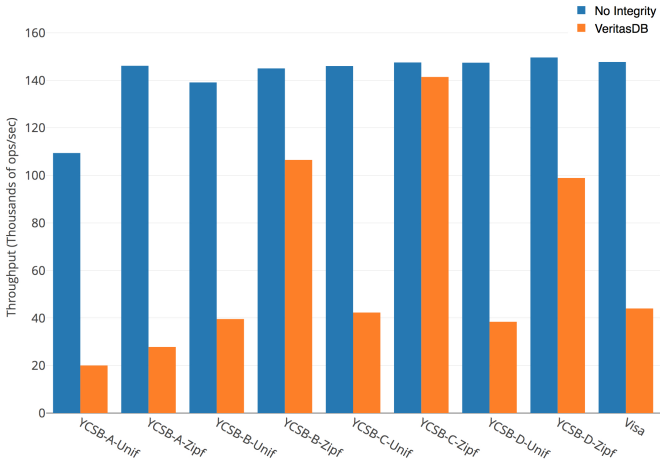

Figure 6: Throughput on YCSB benchmarks, using 1 thread, RocksDB backend server, 5 million keys.

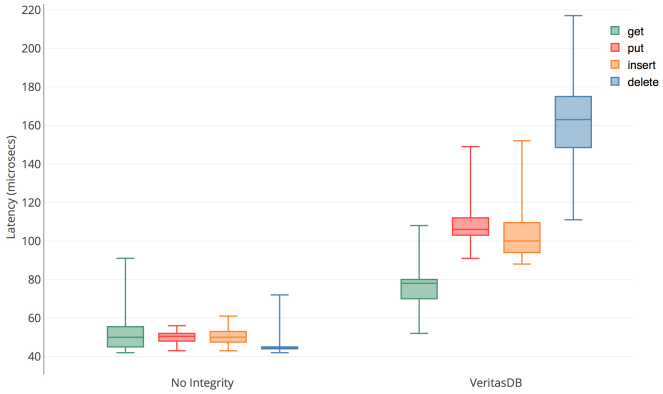

Figure 7: Latency on micro-benchmarks, using 1 thread, RocksDB backend server, and 5 million keys.

While VeritasDB can work with any NoSQL server, we use RocksDB [8] 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 6, where, as the proportion of get operations increases from 50% in YCSB-A to 100% YCSB-C, so does the throughput; Figure 7 also supports this phenomena as get operations have significantly lower latency.

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.

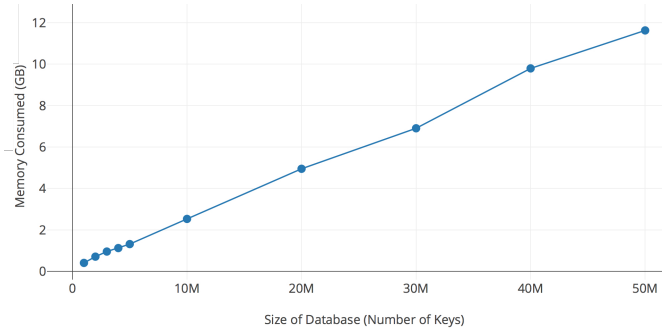

Figure 8: Proxy memory usage by number of keys.

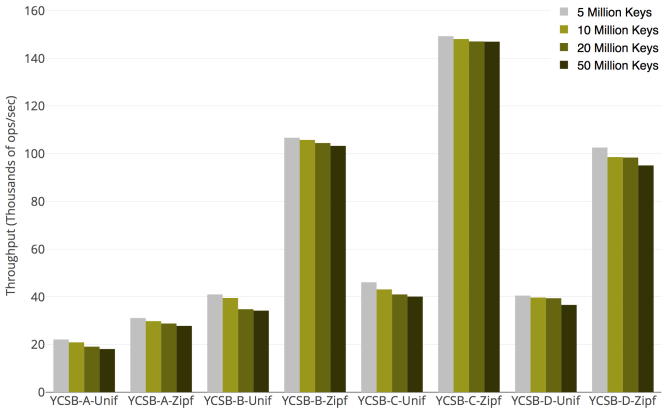

Figure 9: Throughput with 1 thread for varying number of keys, using a RocksDB backend server.

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 8. 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 (§ 5.3).

7.2 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 and illustrate our findings in Figure 9. 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

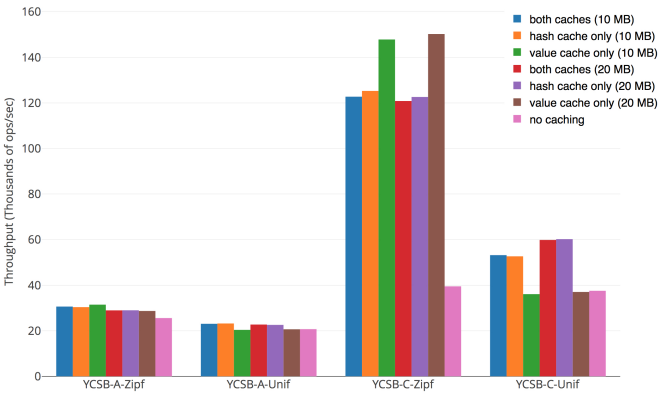

Figure 10: Throughput with various cache sizes, using 1 thread, RocksDB server, and 5 million keys.

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 § 7.1, 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. We leave it to future work to explore sharding across machines.

7.3 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 § 7.1). We also experiment with two different cache sizes: 10 MB vs. 20 MB per cache. Figure Figure 10 illustrates our findings on YCSB-A (update-heavy) and YCSB-C (read-heavy) benchmarks. As mentioned in § 5.1, 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.

7.4 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 § 5.2. Figure 11 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 11 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.

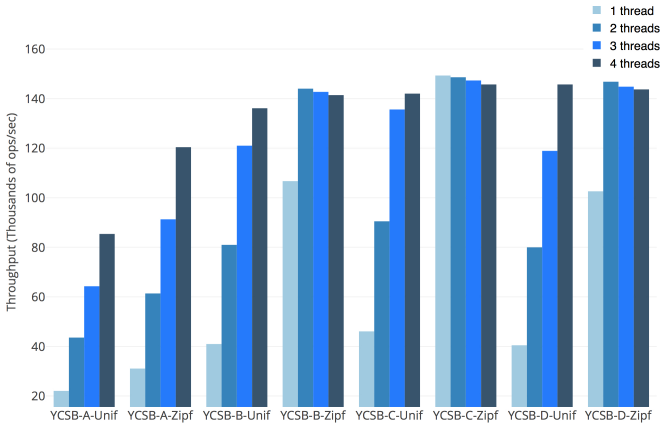

Figure 11: Throughput varying number of threads, using RocksDB backend server and 5 million keys.

8. RELATED WORK

There is a large body of literature and systems built for verifying integrity of an outsourced database [19, 21], 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 empirical 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 (e.g., hashes along a path of the MB-tree) which the client verifies; instead VeritasDB incurs no change to either.

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.

Our work is inspired by Concerto [9], which addressed the concurrency limitations and computational cost of using Merkle trees by proposing deferred (batched) verification based on verified memory [11]. 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.

Outsourced File vs Outsourced Database Integrity. A recent authenticated file system called Iris [28] reported throughput of 260 MB/sec with integrity of both file contents and metadata and with dynamic proofs of retrievability. Athos [18] 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 (in the order of hundreds of millions) with smaller sized values, which leads to larger overheads in checking integrity. When applying techniques from these papers on NoSQL stores (with YCSB benchmarks), we observed a low average throughput of 5000 operations per second (30x overhead) on a moderately sized database with 50 million keys.

Digital-signature aggregation has also been used, but while they guarantee authenticity, freshness was either not addressed or challenging to achieve without adding significant co-ordination between the clients and the servers. File integrity has also been addressed using other space-efficient techniques, such as entropy based integrity [25], but with the drawback of requiring linear time updates.

Key–Value vs SQL Integrity. Zhang et al.'s IntegriDB [29] showed 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. [19], 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 (tuples) 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. Merkle trees are a popular mechanism for data integrity. Alternatively, cryptographic signatures attached to individual data items were also proposed as building blocks for authenticated data structures. Li et al. [21] constructed a variant of Merkle hash tree over the data and used digital signature to add authentication. Our MB-tree is based on their approach, and we extend it by adding the version counters (which saves space) and present bits (to prevent rollback attacks).

VeritasDB avails several features offered by Intel SGX to achieve high throughput, but is not the first database system to leverage trusted hardware. CorrectDB [10] used an IBM 4764 co-processor as a hardware root of trust 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.

9. CONCLUSION

We designed a trustworthy proxy (called VeritasDB) to a key-value store 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 execution environments (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.

10. REFERENCES

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

[2] ZeroMQ: distributed messaging. http://zeromq.org/, 2018.

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

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

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

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

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

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

[9] 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 (SIGMOD'17)*, pages 251–266, New York, NY, USA, 2017. ACM.

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

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

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

[13] 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.

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

[15] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proc. 25th USENIX Security Symposium (Security'16)*, pages 857–874, Austin, TX, 2016. USENIX Association.

[16] 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.

[17] 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 (WDAS'06)*, Santa Clara, CA, January 2006.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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 (ACSAC'12)*, pages 229–238. ACM, 2012.

[29] 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.

[30] 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.

APPENDIX

A. EXTENSIONS

Key Rotation. VeritasDB uses 64 bits for the counter for each key in the version map, although this configuration can be altered. After some duration, this counter is bound to overflow (back to 0), which would allow the attacker to supply stale values, with the same counter as the current value within the version map, and yet pass the integrity checks. Nevertheless, we may never encounter this situation in practice — even in the pathological case where all operations target the same key, at a hypothetical throughput of 1 million operations per second, an overflow occurs once every 584942 years. Despite the unlikelihood of this attack, for provable correctness of VeritasDB, we implement a simple key rotation scheme that pauses operation on encountering an overflow, chooses a new HMAC key `hmac_key` at random, and re-populates the database server with values that are authenticated under the new key. We measured this process to incur approximately 10 minutes of downtime for a database of 50 million keys, at a throughput of 100,000 operations per second.

Fault Tolerance. The proxy instance is a single point of failure in the design. Not only does the proxy's failure make the database unavailable to the client (until it restarts), but it also prevents integrity checks on future operations because its entire state is stored in volatile memory. Here, we discuss standard techniques for improving availability of VeritasDB, though we leave their implementation to future work.

First, replication offers a standard approach for fault tolerance, where the proxy service is distributed onto a cluster of nodes in a primary-replica (leader-follower) setup. Here, the clients connect to a primary node, which propagates all update operations (*put*, *insert* and *delete*) to the replica nodes so that they can update their local state. In the event of failure of the primary node, the client (or some intermediate hop) establishes the connection with a replica node, which becomes the new primary node. To avoid round-trip communication between the primary and replica nodes, we can use a distributed messaging system (such as Kafka) to allow the primary node to "fire and forget", and also allow the replica nodes to recover from their failures. As a next line of defense, we can periodically back up the proxy's state to persistent storage, much like the backend NoSQL store is wont to do.