VeriDB: An SGX-based Verifiable Database

Wenchao Zhou*
Georgetown University &
Alibaba Group
wzhou@cs.georgetown.edu

Yifan Cai†
University of Pennsylvania &
Alibaba Group
caiyifan@seas.upenn.edu

Yanqing Peng†
University of Utah &
Alibaba Group
ypeng@cs.utah.edu

Sheng Wang
Alibaba Group
sh.wang@alibaba-inc.com

Ke Ma†
Shanghai Jiao Tong University
whirlfoureye@sjtu.edu.cn

Feifei Li
Alibaba Group
lifeifei@alibaba-inc.com

ABSTRACT

The emergence of trusted hardwares (such as Intel SGX) provides a new avenue towards verifiable database. Such trust hardwares act as an additional trust anchor, allowing great simplification and, in turn, performance improvement in the design of verifiable databases. In this paper, we introduce the design and implementation of VeriDB, an SGX-based verifiable database that supports relational tables, multiple access methods and general SQL queries. Built on top of write-read consistent memory, VeriDB provides verifiable page-structured storage, where results of storage operations can be efficiently verified with low, constant overhead. VeriDB further provides verifiable query execution that supports general SQL queries. Through a series of evaluation using practical workload, we demonstrate that VeriDB incurs low overhead for achieving verifiability: an overhead of 1-2 microseconds for read/write operations, and a 9% - 39% overhead for representative analytical workloads.

ACM Reference Format:
Wenchao Zhou, Yifan Cai, Yanqing Peng†, Sheng Wang, Ke Ma†, and Feifei Li. 2021. VeriDB: An SGX-based Verifiable Database. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457308

1 INTRODUCTION

Recently there has been a paradigm shift to migrate data storage and associated computing tasks on to the cloud. While increasingly more routine or even mission-critical procedures are hosted on the cloud, data integrity still largely relies on the client's trust that the cloud service provider has faithfully maintained the data (and computed results) outsourced to them. It remains challenging for the client to *verify* whether the data or computation results retrieved from the cloud are correct; from the cloud service provider's perspective, such capability of *proving* to the client that its data is correctly handled is also a highly desirable security feature, encouraging hesitant clients to adopt cloud-centric solutions.

In practice, it is impossible to require that the results returned from the cloud be always correct: an adversarial cloud service provider can simply return empty results for any of the client's requests. *Verifiability* is a more practical and sufficiently strong guarantee in practice, that is, the correctness of any returned results is (cryptographically) verifiable. It allows the client to detect any faults with non-reputable evidence, and the cloud service provider to retain a formal proof for its correct operations.

In the past, there has already been a large body of work in the field of providing verifiability for cloud databases [1, 4, 12, 14, 19, 21, 24, 25, 31]. Notably, a classic approach leverages Merkle Hash Tree (MHT) [18] to ensure data integrity; the recursively defined hash structure of MHT reduces integrity verification to the reproducibility of its root hash. Such MHT-based solutions provide efficient verification for the presence (or absence) of a queried data record in a database. However, it has difficulties handling complex SQL queries. For example, the verification object size of JOINs is linear to the size of input relations. Additionally, MHT-based approaches fundamentally rely on the mutual agreement between the client and the cloud on the MHT's root hash. The root hash is essentially a concurrency bottleneck: updates to the database cannot be parallelized, and reads must wait until all preceding writes are completed. On the other hand, approaches that employ more advanced cryptographic primitives [30, 31] provide support for more general database queries, for example, vSQL allows clients to verify the query result of an arbitrary SQL query. However, most of these approaches fall in short in terms of performance, and may take hours or even days to generate a proof [30, 31].

Recent years, the emergence of trusted hardwares (such as Intel SGX [17]) provides a new avenue towards verifiable database. Such trust hardwares act as an additional trust anchor, allowing great simplification and, in turn, performance improvement in the design of verifiable databases. For example, Intel SGX provides a protected execution environment (called an enclave) within potentially compromised computers, shielding data and attested programs in the enclave from malicious manipulation. The client can therefore delegate some of the verification tasks to programs running in an SGX enclave. Concerto [1] introduces an SGX-based verifiable key-value store. It crucially removes the concurrency bottleneck in MHT-based designs, and adopts deferred verification that employs

*Work is completed during a visit at Alibaba DAMO Academy.
†Work is completed during a research internship at Alibaba DAMO Academy.

high-performance offline memory checking algorithm. However, it only supports key-value storage model and simple read/write interfaces (e.g., `get` or `put` of key-value pairs).

In this paper, we introduce the design and implementation of VeriDB, an SGX-based verifiable database that supports relational tables, multiple access methods and general SQL queries. In VeriDB, the client interacts with a query engine that resides in an SGX enclave. Therefore, the returned query result can be trusted and easily verified (by checking whether it is endorsed by the SGX), as long as the inputs to the query engine, i.e., the data retrieved from the storage, are correct. This effectively reduces the problem of verifying the query correctness to that of verifying the integrity of data retrieval from the storage.

A naïve approach is to maintain data within SGX's internal memory (i.e., EPC) which is naturally resistant against malicious manipulations. However, EPC is a scarce resource with a limited capacity; typical database sizes are well beyond its capacity. It renders the idea of having the complete database hosted within SGX impractical, since it requires frequent data swapping in and out of EPC, which is an expensive operation given the encryption/decryption and integrity checking needed. It significantly affects the database performance, causing orders of magnitude degradation [26].

VeriDB instead separates the execution logic and data storage: while the query execution engine is in an SGX enclave, the actual data is maintained in untrusted memory. The memory integrity is enforced by adopting an offline memory checking algorithm [1, 5]; this ensures the integrity of individual data records. The access methods in the query engine, which serve as the interface that directly interacts with data retrieval and modification in the untrusted memory, are further enhanced to ensure the results from index search, sequential and range scans are correct.

One key intuition of VeriDB is to partition the verification of a cloud database into two *separately running* tasks that verify 1) a data-intensive yet logically simple storage layer, and 2) a logically complex query engine that has a small memory footprint. The former leverages an efficient memory checking algorithm that only introduces constant overhead (1-2 microseconds) to `reads` and `writes`, and the latter relies on the protection provided by the memory-bounded SGX. Furthermore, these two components are connected through a thin interface (i.e., the access methods) that can be efficiently verified, ensuring the verifiability of the complete cycle of database queries and updates.

This paper makes the following contributions:

Page-structured verifiable storage layer. VeriDB adopts a page-structured storage design, to facilitate the integration with legacy relational databases and to support a wide-range of verifiable access methods. Built on top of write-read consistent memory [5], VeriDB provides verifiable results with low runtime overhead for storage operations, where the existence or absence of queried data is proved by a *single* record in the database.

Optimizations that improve storage performance. The page-structured design introduced auxiliary routines such as the maintenance of page metadata and space management within a page. The verification of these routines introduces non-negligible overhead. We present a set of optimizations tailored for the page-structured storage design for mitigating or even removing these overhead.

Verifiable query execution that supports general SQL queries. VeriDB hosts the query engine within an SGX enclave, and pushes the verification tasks to access methods, which can efficiently verify the correctness and completeness of retrieved data with the support of verifiable storage. VeriDB's design requires little changes to existing database execution engine to allow practical adoption.

System implementation and evaluation. Finally, we implement a prototype of VeriDB, and demonstrate, through a series of evaluations using practical workload, that VeriDB incurs reasonable overhead for achieving verifiability: VeriDB introduces an overhead of less than 2 microseconds for read/write operations, and a 9% - 39% overhead for representative analytical workloads.

The rest of the paper is organized as follows: Section 2 presents a brief background on SGX and classic approaches towards verifiable databases. Section 3 introduces the design goals and an architectural overview of VeriDB. We present in details the design of two key components in VeriDB, the verifiable storage layer and the query execution built on top of it, in Section 4 and Section 5 respectively. Section 6 shows a series of evaluations to demonstrate VeriDB's performance in practical workflows. Finally, we present a discussion of related work in Section 7 and conclude the paper in Section 8.

2 BACKGROUND

VeriDB relies on the support of trusted hardwares such as Intel SGX, and draws inspiration from prior work in the field of verifiable outsourced databases. In this section, we present a brief introduction of prior studies in these two fields.

2.1 Intel SGX

Intel Software Guard Extensions (SGX) is a state-of-the-art implementation of the trusted execution environment (TEE), which guarantees the confidentiality and integrity of computation and data. It can be used to protect the execution of applications in outsourced environments, where the host might be compromised. More implementation details can be found in [6, 10, 11].

SGX provides the protection via a virtual container called *enclave*. An *enclave* is an isolated virtual address space in a normal process, where both its code and data are stored in reserved memory pages called enclave page caches (EPC). The EPC cannot be accessed by the rest of the host, including the process outside of the enclave, other processes and the operating system. Hence, the integrity of any computation conducted by the enclave and data stored in EPC are guaranteed. An enclave is initialized by loading a special library whose authenticity can be verified, and can only be invoked via well-defined interfaces called `ECalls`. SGX also introduces `OCalls` to allow programs running in an SGX enclave to invoke libraries outside the SGX. SGX supports *remote attestation* that allows the client to verify the authenticity of an enclave and its loaded code/data on an untrusted host. Note that, while the rest of the host cannot access enclave's EPC, the enclave can access the entire address space on the host. This is an important property that can be exploited for fast data fetching.

When leveraging it in practice, there are several limitations of SGX that should be seriously handled. First, the preserved memory space for SGX is extremely limited, which is up to 128 MB in the current implementation [2, 3, 20] and the usable EPC capacity for enclaves is even lower. Although SGX provides virtual memory

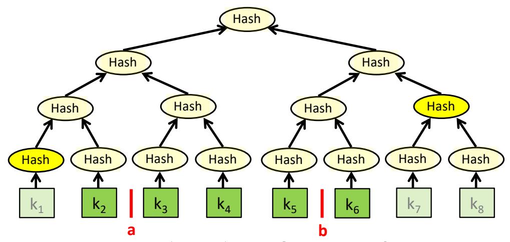


Figure 1: MHT-based verification of range scan.

(i.e., swapping unused EPC pages to unprotected memory) to hide this limitation, a page swapping can easily consume 40000 CPU cycles [2, 6]. Second, an `ECall` is expensive, which is about 8000 cycles as previously reported [20, 27]. This requires the application to carefully reduce its `ECalls`.

2.2 Verifiable outsourced databases

Since the inception of cloud databases, the verifiability of outsourced databases has been a high-value target. A first set of practical solutions adopt Authenticated Data Structure (ADS) based verification. Notably, Merkle Hash Tree (MHT) is a widely adopted ADS; MHT is a recursively defined tree structure that leverages irreversible hash functions to check the absence, presence of data records in an outsourced database. Briefly, an MHT's leaf nodes are the hash calculated from the data, and the internal nodes are recursively calculated as the hash of the contents of the children nodes. Any manipulation or omission of the data will fail to regenerate the root hash, and thus be detected. MHT can further efficiently verify the correctness of range queries.

Example 2.1. Consider a table consists of eight records $k_1 < k_2 < \ldots < k_8$, Figure 1 shows the MHT built on top the table. Given a range query that searches for records in $[a, b]$, we expect it returns $\{k_3, k_4, k_5\}$ as the query result. For verification purposes, the MHT-based solution will return records $\{k_2, k_3, .., k_6\}$ (highlighted in green): k_2 and k_6 are returned to prove that no records in range $[a, k_3)$ and $(k_5, b]$ are omitted. Two additional hashes (highlighted in yellow) are also returned to reconstruct the root hash.

However, it is hard to apply MHT-based approaches to verify more general SQL queries. For example, verification of multi-dimensional range query and JOIN queries are challenging. More critically, data updates require recalculation of the root hash. This effectively creates a concurrency bottleneck: verification of subsequent read operations have to wait until all preceding writes complete, even if they have no direct conflicts.

There have been several efforts to address these two limitations. One direction is to extend the applicability to more general SQL queries by adopting more advanced crypto primitives [30, 31]. For example, IntegriDB employs the bilinear accumulator primitive in its design, and uses interval trees as a building block to support a wider range of SQL queries. However, its proof generation is expensive, which can take hours or even days for large databases.

In another direction, to support high-performance verification in update-heavy databases, Concerto [1] adopts a drastically different approach. It abandons the use of MHTs, and instead reduce the verification of KV store to verifying memory integrity. This removes the concurrency bottleneck in MHT-based approaches, however, it supports only KV stores and simple read/write interfaces.

3 SYSTEM OVERVIEW

In this section, we describe the threat model (Section 3.1) considered in this paper, followed by the targeting design goals (Section 3.2). We further present in Section 3.3 an architecture overview that introduces the main components in VeriDB.

3.1 Threat model

We consider the typical cloud service scenario consisting of two participants, where a client interacts with a cloud database service provider. The client relies on the service provider to maintain its database and execute queries against the database. However, the client may not fully trust the service provider. In the paper, we focus on the threats to the *integrity* of the database and the query results; solution that provides data confidentiality, for example, through encryption and access control, may be layered on top of VeriDB.

Untrusted service provider. The security threat originates from an adversarial or compromised service provider, who may alter the database or the query result arbitrarily. For example, the service provider may 1) insert, alter or delete arbitrary data in the database, and/or 2) return incomplete, stale, or even fabricated query results. More generally, we conservatively assume Byzantine faults.

Existence of SGX. In this Byzantine threat model, the adversary has full control of the cloud database, except a protected TEE that holds a small set of data and attested programs. More specifically, we assume that the cloud database is equipped with Intel SGX, where the data and programs in the SGX enclave cannot be accessed or modified without the client's authorization, even if the adversary controls the rest of the computer.

3.2 Goals

The overarching goal of this paper is to design an SGX-based verifiable cloud database, which allows clients to detect incorrect query results returned by an adversarial or compromised cloud database service provider. we say a cloud database is *verifiable*, if it satisfies the following two correctness properties:

- **Endorsement of correct results.** When the database returns correct query results, no false alarms should be reported and the result should be endorsed, e.g., through cryptographic signatures, by the trusted hardware. Here, we say a query result is *correct* as if the query were performed on a database that runs on the client's own machine.
- **Detection of incorrect results.** When the database returns incorrect results, the client can detect such misbehavior with an associated evidence that proves this misbehavior.

Additionally, we consider the following two goals to ensure that the proposed design is practical and has wide applicability.

- **Support for general SQL queries.** The cloud database should support verifiability for general SQL queries. In this paper, we focus on SPJA queries, but the proposed solution can be straightforwardly extended to support other relational operators, such as nested queries.
- **Low overhead.** Finally, security enhancement to support verifiability should have small impact on the database's query processing performance. For example, it should not impose constraints that significantly impact the database's capability of support concurrent queries.

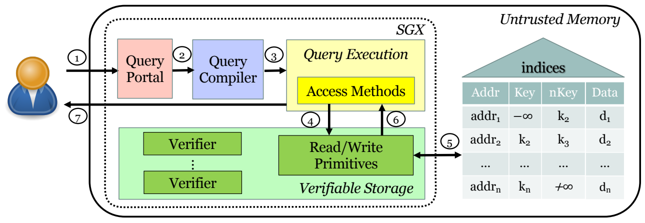


Figure 2: Architectural Overview of VeriDB.

Non-goals. A verifiable database allows users to *detect* misbehavior of a compromised cloud database. However, it doesn't *tolerate* misbehavior, that is, it doesn't return correct results when the database is compromised. For example, data may be lost if an adversarial service provider deletes or otherwise modifies the data stored in the cloud database. But we still consider that there is a sufficiently wide range of application scenarios, especially given that the service providers are big players in the cloud ecosystem: any detection of misbehavior would lead to serious penalties, such as reputation dent, loss of customer loyalty, or even lawsuits, which strongly incentivize these service providers against data tampering. This is significantly different compared to the typical blockchain application scenario where the goal is to tolerate a bounded number of misbehaving participants and participants may leave, become irresponsive or otherwise misbehave at an arbitrary time.

3.3 Architecture

At a high-level, VeriDB separately verifies the integrity of the storage layer and the query execution engine. The verifiable storage ensures the verifiability of individual read/write operations, and the query execution engine ensures the correctness of the query output if its inputs are correct.

Figure 2 presents the query workflow in VeriDB. A client's query Q is authenticated and communicated to the query portal, which resides in the SGX enclave, through a secure channel, to ensure that the query is indeed initiated by the client (Step 1). The query portal passes the query to the query compiler (Step 2), which converts the query into a physical query plan, and further passes it to the query execution engine (Step 3). While the execution logic resides in the SGX enclave, the access methods need to retrieve data from untrusted memory (Step 4). The verifiable storage layer protects the read/write operations (Step 5) and leverages an offline memory verification process to continuously check the integrity of individual data. Multiple verifiers may be employed to verify different (disjoint) sections of the memory for performance purposes. The data retrieved from the storage layer is verified by the access method for completeness (its integrity is guaranteed by the verifiable storage), and is used as the input to the query execution engine (Step 6). Finally, the execution engine returns and endorses the query result back to the client (Step 7).

Query compiler. Verification of an arbitrary SQL query under an untrusted environment is extremely challenging. Given a user-specified SQL Q and its corresponding compilation result $Plan_Q$, which is typically represented as a query execution plan tree, it is relatively straightforward to convert $Plan_Q$ to an equivalent representation in relational algebra (i.e., as a SQL query) Q'. However, given that the compilation and optimization may significantly change the execution plan, Q' may look very differently compared with Q. Verifying the equivalence of Q and Q' is provably difficult – equivalence of conjunctive queries is a classic NP-hard problem in database literature. Therefore, query compilation and optimization are performed within the trusted SGX environment.

Trusted storage in untrusted memory. The storage layer of VeriDB provides the basis for the verification of complex SQL queries: it ensures that individual read/write operations of any record in the database are verifiable, that is, each read/write operation is associated with an evidence that proves its correctness.

SGX provides a section of trusted memory that is within the protection of trusted hardware. However, its size is limited to 96 MB; data beyond this size will cause expensive memory swaps that are across enclave boundaries [26]. Therefore, a fundamental design decision of VeriDB is to place the database outside the SGX, and only maintain a small set of data within the SGX for verification purposes. This is achieved by adopting a *write-read consistent* memory implementation. Briefly, SGX ensures the integrity of a small synopsis of all the client-initiated writes; any tampering of the data that bypasses the operations in the enclave will cause inconsistent synopsis and be detected.

Such design has an important advantage: the correctness of a read/write operation can be proved by a single data record in the database. The construction of proofs does not require access to any shared data, and therefore allows for better concurrency; this is in contrast to MHT-based approaches where every proof requires the inclusion of the most updated MHT root.

Query execution. With the existence of SGX, the client can delegate part of the verification to the attested code resides in an SGX enclave, removing the need of shipping large-size proofs over the network and the client's burden of verifying the proofs. In VeriDB, the query execution engine resides in the SGX, this design decision is based on the goal of maximizing the system performance: entering or leaving an SGX enclave, through the use of `ECalls` or `OCalls`, adds non-negligible overhead; therefore, VeriDB avoids frequent `ECalls` (or `OCalls`) by colocating the query execution and the trusted storage.

The SGX-resident query engine ensures the integrity of the query result, provided that its inputs are correct. The verifiable storage layer provides APIs for verifiable access to individual records, that is, the integrity and freshness of the retrieved record are verified. The access method further verifies the completeness of inputs, and therefore ensures the correctness of inputs and, in turn, the outputs of the trusted query engine.

4 STORAGE LAYER

We adopt a storage design that aims to minimize the negative performance impact on the database system, while achieving the verifiability properties (see Section 3.2). In VeriDB, the storage supports relational tables and follows the widely adopted page-structured design; this benefits a stronger practicality, as it can be seamlessly integrated with most relational databases which also adopt page-structured designs for buffers, and in-memory and disk-based storages. Upon requests from the query execution engine, the storage layer returns results for read/write operations with associated proofs to prove the presence or absence of requested record. In this section, we present the construction of theses proofs.

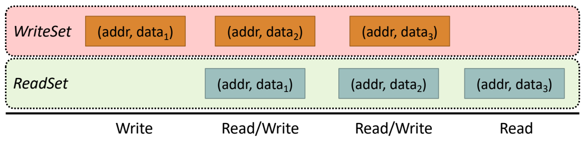


Figure 3: Example of ReadSet and WriteSet.

4.1 Preliminary: write-read consistent memory

As a fundamental building block, VeriDB relies on *trusted* memory to store data records. It employs a set of verification protocols [1, 5] to ensure that the memory is *write-read consistent*.

Definition 4.1. Memory is *write-read consistent*, if, for every read on address *addr*, it returns the same data as what is written by the most recent write on *addr*.

The definition is further simplified by requiring every write to check the integrity of *addr* before it overwrites the data, and every read to (virtually) write back the data after the read. This way, the reads and writes on *addr* are interleaving. Figure 3 shows an example of such interleaving writes/reads. Note that, such additional requirement does not add much burden to the storage system – data is loaded into cache anyways for the write/read.

Read and Write Sets. Blum et al [5] introduces an efficient implementation of write-read consistent memory, which is developed from a simple observation: if the memory is write-read consistent, at any given time, the set of all reads (or ReadSet, written as $\mathcal{RS}$) closely tracks the set of all writes (or WriteSet, written as $\mathcal{WS}$). The WriteSet contains exactly one more item for each memory location *addr* (i.e., the last write on *addr*). For the example shown in Figure 3, the WriteSet on *addr* contains the additional write of $data_3$. At the time of verification, we scan the memory to read the latest content of each memory location, after which the ReadSet becomes exactly the same as the WriteSet. Set equivalence can be efficiently evaluated [5] by checking the collision-resistant hash function h over the sets: for example, $h(\mathcal{RS}) = \sum^{\oplus}_{(addr,data)\in\mathcal{RS}} PRF(addr, data)$, that is, the set hash is the xor sum of the keyed pseudo-random functions of all set elements. $\mathcal{RS} = \mathcal{WS}$ implies $h(\mathcal{RS}) = h(\mathcal{WS})$, and $h(\mathcal{RS}) = h(\mathcal{WS})$ implies $\mathcal{RS} = \mathcal{WS}$ with high probability.

Following this intuition, SGX enclave maintains $h(\mathcal{RS})$ and $h(\mathcal{WS})$, as two byte arrays (currently with length of 64-byte), and require that all read/write operations be performed by dedicated procedures in the SGX which complete these operations and update $h(\mathcal{RS})$ and $h(\mathcal{WS})$ accordingly. Algorithm 1 presents the pseudocode of the protected Read and Write procedures: during a Read process, in addition to the actual fetch of data (Line 2), we include this read operation to $\mathcal{RS}$ and update $h(\mathcal{RS})$ (Line 3). To ease the handling of consecutive reads, we *virtually* write the data at its current location (no actual I/O is performed), and include this write in $h(\mathcal{RS})$. $h(\mathcal{WS})$ is updated accordingly (Line 5). The process of Write is similar and omitted due to space constraints.

Non-quiescent verification. The Blum algorithm introduces a small overhead at runtime: $h(\mathcal{RS})$ and $h(\mathcal{WS})$ are calculated incrementally during each read/write operation. Concerto [1] further improves the verification performance using a *non-quiescent* verification process, where the verification runs in parallel with routine reads and writes (shown in Algorithm 2). The system concurrently maintains the ReadSet and WriteSet of *two* adjacent epochs; the

Algorithm 1 Verifiable Read and Write (protected by SGX)

```
1:  function Read($addr$)
2:      $data \leftarrow Mem[addr]$          ▷ $Mem[addr]$ is the content of $addr$
3:      $h(\mathcal{RS}) = h(\mathcal{RS}) \oplus PRF(addr, data)$          ▷ Update ReadSet
4:      ▷ (Virtually) write the same data back
5:      $h(\mathcal{WS}) = h(\mathcal{WS}) \oplus PRF(addr, data)$          ▷ Update WriteSet
6:      return $data$
7:  function Write($addr, data^{new}$)
8:      $data^{old} \leftarrow Mem[addr]$
9:      $Mem[addr] \leftarrow data^{new}$
10:     $h(\mathcal{RS}) = h(\mathcal{RS}) \oplus PRF(data^{old})$          ▷ Update ReadSet
11:     $h(\mathcal{WS}) = h(\mathcal{WS}) \oplus PRF(data^{new})$          ▷ Update WriteSet
```

Algorithm 2 Non-quiescent Verification (protected by SGX)

```
1:  function Verify
2:      $h(\mathcal{WS}^{new}) = 0$
3:      for each $page \in memory$ do          ▷ Scan through the pages
4:          $page.lock$ = true          ▷ Lock the page
5:          for each $(addr, data) \in page$ do
6:              $h(\mathcal{RS}^{current}) = h(\mathcal{RS}^{current}) \oplus PRF(addr, data)$
7:              $h(\mathcal{WS}^{new}) = h(\mathcal{WS}^{new}) \oplus PRF(addr, data)$
8:          $page.lock$ = false          ▷ Release the lock
9:      if $h(\mathcal{RS}^{current}) \neq h(\mathcal{WS}^{current})$ then return false
10:     else return true
```

start of the memory scan indicates the start of a new epoch. As the verification scans through the memory, it reads all data exactly once (Line 5). For each data it scans, it includes the read in the $\mathcal{RS}$ of the current epoch (Line 6) and initiates the $\mathcal{WS}$ of the next epoch (Line 7). When the scan of the memory completes, the verification process checks the equivalence of $\mathcal{RS}$ and $\mathcal{WS}$) of the current epoch (Line 9); any data tampering will be detected then. Note that only the page that is currently being scanned is locked and temporarily stops routine reads and writes. For a large storage system, we expect that such contentions happen rarely and unlikely affect the performance.

4.2 Page-structured verifiable storage

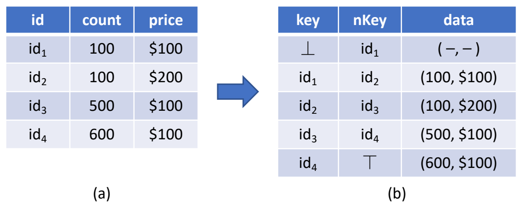
Based on the write-read consistent memory, we develop a page-structured verifiable storage, where the existence or absence of a queried data is proved by a single record in this storage. To achieve this, we extend the storage of a relation $\mathcal{R}$ by requiring each record to store, in addition to its primary key, the next smallest key:

Definition 4.2. Consider a relation $\mathcal{R} = \{r_i = \langle f_1, f_2, ..., f_n\rangle\}$ consisting of n columns. Without loss of generality, suppose column 1 is the primary key, i.e., r_i's key is f_1. The tuple r_i is stored as

$$r_i = \langle key, next(key), data\rangle$$

where *data* is the aggregated data that contains $f_2 ... f_n$; $next(key)$ (or *nKey* for short) is the smallest key greater than *key*, or $\top$ if such key doesn't exist (i.e., f_1 is the largest key). An additional record $\langle \perp, \min(keys), -\rangle$ is also inserted, where $\min(keys)$ is the minimal primary key in the table and $-$ indicates null value.

With this extended model, a single record is sufficient to prove the presence or absence of a queried data. For example, the record $\langle k, nKey, data\rangle$ itself is an evidence of its existence (since the record is read from a trusted memory). The absence of a tuple with primary key k', $k_1 < k' < k_2$, can be proved by the record $\langle k_1, k_2, data\rangle$.



id	count	price
id_1	100	\$100
id_2	100	\$200
id_3	500	\$100
id_4	600	\$100

(a)

key	nKey	data
$\perp$	id_1	$(-,-)$
id_1	id_2	(100, \$100)
id_2	id_3	(100, \$200)
id_3	id_4	(500, \$100)
id_4	$\top$	(600, \$100)

(b)

Figure 4: An example relation (Figure a) stored in the extended storage model in VeriDB's storage layer (Figure b).

Example 4.3. For example, consider the relation shown in Figure 4. $\langle id1, id2, (100, \$100)\rangle$ proves the existence of $\langle id1, 100, \$100\rangle$; A query for a tuple keyed by $id > id4$ returns *null* with an evidence $\langle id4, \top, (600, \$100)\rangle$ that proves the returned result.

Page structure. These records are stored in VeriDB as pages; the structure of a VeriDB page resembles classic page designs in database systems like Postgres. Each page contains in the page header a set of metadata such as the total page capacity, space that is still available, number of records in the page, etc. The core of the page is a list of records referenced by pointers that record their offset relative to the start of the page. Using the pointers, records are easily accessible by $(page, index)$—a pointer to the page and the record's index in the page.

Supported operations. VeriDB provides the following read and write interfaces for interacting with the page-structured storage. Algorithm 3 shows their pseudocode (we omit the pseudocode of update, as it is similar to a delete followed by an insert):

- **Get(page, index)** reads the offset of the queried record from the corresponding pointer, which is then used to retrieve the data from the page.
- **Insert(page, data)** first checks whether the page still has enough free space to store the data. If it does, a new pointer is added to point to the newly inserted data. The insert further identifies (e.g., by checking the index) the record whose primary key right precedes the current one, and updates its *nKey* to the key of *data*.
- **Delete(page, index)** resets the pointer to the deleted data, and reclaims the space previously allocated for the record. It also needs to identify the record whose primary key right precedes the current one, and updates its *nKey* to the deleted data's *nKey* (i.e., removing the delete record from key chain). Reclaiming the space may be expensive as multiple records may be moved within the page to maintain contiguous unused space. However, we show that this can be optimized by delaying the space reclamation (see Section 4.3).
- **Update(page, index, data^{new})** updates the specified record at the given page. There is no need to update the key chain, as it only updates the *data* field in $\langle key, nKey, data\rangle$. However, in cases where $data^{new}$ occupies more space than the current *data*, the update will need to perform a delete followed by an insert, which may happen on a different page.

Algorithm 3 Interfaces of Page-structured Storage

```
1:  function GET($page, index$)
2:      $addr \leftarrow$ READ($page.ptr[index]$)
3:      return $data$ = READ($page[addr]$)
4:  function INSERT($page, data$)
5:      $addr \leftarrow$ ASSIGNADDR($page, data.length$)
6:      if $addr < 0$ then return $-1$                ▷ Not enough space
7:      WRITE($page.ptr[index], addr$)                ▷ Add a new pointer
8:      WRITE($page[addr], data$)                     ▷ Write data
9:      $(page', index') \leftarrow$ PREV($data.key$)
10:     UPDATENEXTKEY($page', index', data.key$)
11:     return $index$
12: function DELETE($page, index$)
13:     WRITE($page.ptr[index], -1$)
14:     RECLAIMSPACE($page$)              ▷ Reclaim space of the delete record
15:     $(page', index') \leftarrow$ PREV($data.key$)
16:     UPDATENEXTKEY($page', index', data.nKey$)
```

As we can notice in Algorithm 3, these operations rely on the verifiable read and write discussed in Section 4.1. For example, *Get*$(page, data)$ performs two verifiable reads (Line 2 and 3) on the write-read consistent memory, one for reading the record's offset in the page, and the other for retrieving the actual data. The overhead of each verifiable read or write (compared to their non-verifiable counterparts) comes mainly from the evaluation of two cryptographic hash functions (see Algorithm 1). While this overhead is reasonable—the overhead of verifiable read/write is consistently between 1.4-4.2 microseconds, we can adopt optimizations (see Section 4.3) to further reduce the number of hash functions needed, and in turn reduce the read/write overhead.

Besides these read/write interfaces, VeriDB further provides two additional interfaces for handling space management needs:

- **Register(page)** includes the specified page into the verification process. This is called by memory management modules, and allows user to gradually add data that needs to be protected. The *Register* process also updates $h(\mathcal{WS})$ based on the initial data in the page.
- **Move(page^{old}, $\text{index}^{\text{old}}$, page^{new})** moves data to a new location. In theory, the move operation can be simulated by a delete and an insert. But, we may want to make it atomic and protected by the SGX. For example, when an index algorithm run in untrusted environment decides to merge or split pages for space management purposes, the data move should be protected, such that the integrity of actual data, which serves as evidence in verification, is still protected.

4.3 Optimizations

Exclude page metadata from verification. The main functionality of page metadata is to provide addressing service for the actual data record. We in fact only need to guarantee that the read of the data record is consistent with its corresponding write. Therefore, the ReadSet/WriteSet operations related to the page metadata (e.g., check availability the space, find relative offset of a record within the page, etc) can be omitted. It significantly reduces the number of ReadSet/WriteSet operations (by 50%-65%), which translates to a 20% reduction in the overall overhead.

The tradeoff is that some misbehavior that only manipulate the page metadata cannot be detected. For example, a compromised server can waste space in a page (falsely report the capacity of availability space in a page). However, 1) it does not impact the correctness of the goals presented in Section 3.2, and 2) the service provide does not have incentive to so, as it is lowering how efficiently it is using its own physical storage.

Compact page during verification. The default page implementation assumes that unused space is a contiguous region, for higher space utilization and easier management. Such design requires significant data relocation during deletion: since records can have different sizes, we cannot simply swap the deleted records with the last record in a the page. On average, half of the records in the page needs to be read and written to another location in the page. To address this issue, we remove the assumption that unused space is contiguous and delay the space reclamation, such that unnecessary data move is completely removed during the deletion process. A separate *compaction* process is employed to periodically perform space reclamation and gather unused space into a contiguous region again. While having a separate compaction process removes the need of data relocation during each delete, it still needs to update $h(\mathcal{RS})$ and $h(\mathcal{WS})$ at compaction time.

As another step further, we find that the compaction process resembles the verification process of write-read consistent memory (see Algorithm 2): suppose there are N records in a page, both need to read N records, and update $h(\mathcal{RS})$ and $h(\mathcal{WS})$ N times. Therefore, we can perform the compaction process as a side-task during the verification process. This way, the compaction introduces little, if any, additional overhead, while retaining high space utilization in the page-structured storage.

Avoid scanning unvisited pages during verification. Additionally, by organizing data in pages, we can further optimize the verification process: the verification process requires the scan of the whole memory to make the read and write sets consistent; scan the complete memory is likely an overkill and can be optimized by scanning the pages that have been *touched*, i.e., read or written, after the last verification. Such information needs to be maintained within SGX. Assuming a total memory capacity of 32 GB and a page capacity of 8 KB, it requires one bit for each of the 4 million pages, that is, it consumes 0.5 MB memory within the SGX enclave. While it is non-negligible, the storage cost of such a structure is affordable given that the total capacity of SGX memory is 96 MB. If VeriDB is required to support memory space well beyond 32 GB, we can further adopt coarser granularity (e.g., consider 16 pages as a group) to reduce the memory utilized for tracking the pages that are touched between two verifications.

Use multiple RSWSs to avoid lock contention. To improve the performance of concurrent queries, VeriDB introduces RSWS locks. A higher concurrency means more likelihood to have multiple workers updating the ReadSet/WriteSet, i.e., a higher contention for the RSWS locks. Therefore, as an optimization, VeriDB allows multiple ReadSets/WriteSets (corresponding to disjoint memory sections), with a lock for each set. On updating or checking a ReadSet/WriteSet, the worker or verifier grabs only the corresponding RSWS lock. Compared with MHT-based solutions, which require global lock that halts the system for updating the MHT's root hash, the RSWS locks allow for much finer granularity concurrency management.

5 QUERY EXECUTION

VeriDB's storage layer ensures that reads and writes of individual tuples in the database are verifiable; the presence and absence of a queried tuple is proved by a single record in the verifiable storage. In this section, we show the design of a secure query engine that leverages these proofs for individual reads and writes to generate verifiable results of complex SQL queries. In VeriDB, the query execution engine resides in an SGX enclave, colocating with the storage layer's read and write interfaces. This eliminates the needs of frequent `ECalls` and `OCalls` during the interaction among the storage layer and different operators in the query execution engine.

Section 3.2 presents a general correctness definition: the returned result is correct, if it is consistent with what would have been returned by a database instance running by a trusted entity (e.g., by the client). More specifically, given a query Q, we consider the following properties of the returned query result:

Integrity: Any tuple returned should satisfy Q.
Completeness. All tuples that satisfy Q are returned (not omitted).
Freshness. The query is executed on the latest state of the database.

In the rest of the section, we describe the query portal (Section 5.1) that performs query authorization and recovery from power failure, and the support for access methods such as point and range scans (Section 5.2). We then present how it can be further extended to support general SQL queries in Section 5.4 and provide a brief security analysis in Section 5.5.

5.1 Query portal

The query portal runs insides an SGX enclave, and is the entry point of queries submitted from the users. Notably, it performs the following tasks to ensure data integrity.

Query authorization. VeriDB should only execute queries (and the corresponding memory accesses) that are indeed initiated by the clients; otherwise, an adversarial service provider can launch a SQL query to modify the database in any way it wants. This is enforced by a query authorization process (as a part of Step 1 in Figure 2). Briefly, the client and its trusted query execution engine (resides in the SGX) maintains a pre-exchanged key k. Each query issued by the client is associated with a unique query id *qid* and a message authentication code (MAC) generated using k; upon receiving a query, the query portal verifies that *qid* has never been seen before and that the query matches its MAC. Similarly, the query and verification results generated by the query engine should also be authenticated by the SGX and verified by the client (as a part of Step 7 in Figure 2).

Recovery from failure. VeriDB crucially relies on states securely stored in SGX, such as $h(\mathcal{RS})$ and $h(\mathcal{WS})$, to achieve verifiability. It is inherently vulnerable to power or machine failure: these data are permanently lost. However, it is worth noting that, as VeriDB focuses primarily on in-memory database, power failure wipes out not only the data inside SGX enclaves, but also the *complete* database. Therefore, re-establishing the states in SGX can be performed as a side-task during the ordinary database recovery used in typical databases: the query portal starts the recovery by copying data from a designated source, e.g., from a remote replica, as how a typical in-memory database operates. These repeated writes use the same interfaces introduced in Section 4.2, and naturally update the states stored in SGX. The always-running verification process further ensures that data tampering during the recovery can be detected.

Defense against rollback attack. An adversarial service provider may introduce intentional power failure to launch *rollback attack*, that is, revert the system to an old state which *was* valid in the past. For example, the compromised server can wipe out the SGX state

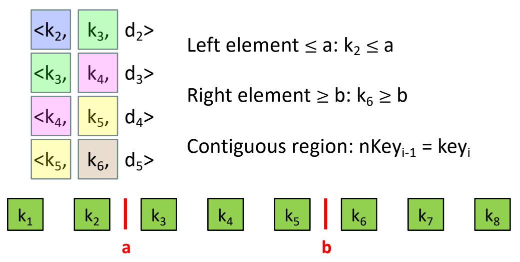


Figure 5: VeriDB support for verification of range scan.

by an intentional power failure and then replay selected queries. Previous research has shown that defending against rollback attack crucially relies on a *trusted persistent* storage[9]. Data can be protected by a consensus protocol[16] or storage controlled by a trusted entity (e.g., the user)[13]. In VeriDB, we require the user to maintain a small piece of data, which effectively serves as a synopsis of query history, to defend against rollback attacks.

More concretely, the query portal maintains a strictly increasing counter: the counter is incremented for each received user query; at the same time, the value of the counter is assigned to the query as its sequence number. When a query is completed, the associated sequence number is returned back to the user along with the query result. Given that the counter increases strictly, the user should never see a sequence number twice. However, any rollback attacks will inevitably decrease the value of the counter, and cause the user to receive a repeated sequence number[1]. The user maintains a list of received sequence numbers, and verifies that there is no repetition in the list. VeriDB leverages optimizations such as maintaining intervals of successive sequence numbers (instead of individual numbers) to help reduce user's storage cost.

5.2 Secure access methods

Access methods interact directly with the storage layer, and rely on the returned proofs, in the format of $\langle key, nKey, data\rangle$, to verify the correctness of query result.

Index Search. One of the most basic query types is to search for a specific record in a *table* based on its primary key:

```
SELECT * FROM table WHERE key = keyValue
```

Such query leverages the $Get(page, index)$ API provided by the storage layer, where $(page, index)$ is retrieved from an index stored in untrusted memory (the index does not need to be verifiable). The index returns the $(page, index)$ pair for the largest key that is less than `keyValue`: this supports the generation of evidence for searches that do not have a match record for the queried `keyValue`. The returned result of $Get(page, index)$ is $\langle key, nkey, data\rangle$, VeriDB verifies that one of the two conditions holds:

(1) key = `keyValue`, in which case, the index search returns a tuple (i.e., $\langle key, data\rangle$) that matches the query; or
(2) key < `keyValue` < $nKey$, in which case, the index search returns null as the result.

Otherwise, the verification fails; for example, the untrusted index may return a tampered $(page, index)$ pair.

Range Scan. Range scan returns *all* records whose primary keys are in a specified range $[start, end]$:

```
SELECT * FROM table WHERE start<=key and key<=end
```

Before we introduce VeriDB's support for verifying range scans, we briefly revisit the classic MHT-based solution for range scan verification. The query result is returned as an ordered list of records $r_1, ..., r_k$ that satisfies the following three conditions (1) $r_1 \leq start$, (2) $end \leq r_k$, and (3) r_{i+1} is the successive record of r_i in the MHT. In addition, it returns necessary metadata to allow reconstruction of the MHT's root hash. VeriDB verifies the result of range scan by checking conditions 1 - 3 as well.

Example 5.1. In VeriDB, the range query relies on the index to locate the largest key not exceeding a, and then traverses the ordered records until reach a record that is greater than or equal to b. By calling $Get(page, index)$ repeatedly, the query engine receives $\langle k_2, k_3, d_2\rangle, ..., \langle k_5, k_6, d_5\rangle$. Figure 5 illustrates VeriDB's verification of this range scan. It verifies the following conditions hold:

(1) The key of the first record (i.e., k_2) is less than or equal to a.
(2) The $nKey$ of the last record (i.e., k_6) is greater than a.
(3) Each record's key is the same as its predecessor's $nKey$ (highlighted with same color), to ensure that the records form a contiguous region in the table (ordered by the key).

5.3 Verifiable scan on multiple columns

Verification of the access methods heavily rely on the $\langle key, nKey\rangle$-structured evidence. These $\langle key, nKey\rangle$ structures effective form a key chain—The storage model presented in Section 4 maintains such a chain on the table's primary key. However, it does not provide good support when search condition of a point search or a range query is related to a column other than the primary key [2].

A naïve solution is to maintain multiple tables in the storage, one for each column. However, this leads to large redundancies. In fact, the ordering of the records' physical locations is inconsequential, instead, the verifiability roots from the integrity and completeness of the key chains. Therefore, we maintain one single copy of the data, but extend the storage model to maintain multiple $\langle key, nKey\rangle$ chains, one for each column that appears in a range scan (or a key search) as the range (or search) key.

Definition 5.2. Given a relation $\mathcal{R} = \{d = \langle f_1, f_2, ..., f_n\rangle\}$ consisting of n columns. Suppose we provide access methods on k columns (without loss of generality, suppose they are columns 1 through k), we model the data as

$$\langle key_1, nKey_1, key_2, nKey_2, ..., key_k, nKey_k, data\rangle$$

where $data$ is the aggregated data that contains $f_{k+1}...f_n$.

Effectively, this creates k chains, one for each column, such that the access method on that column can be verified for the correctness.

Example 5.3. As an example, consider a relation that supports access methods on two columns. In the extended storage model, each record has the format $\langle key_1, nKey_1, key_2, nKey_2, data\rangle$. Figure 6 shows the table status during the insertion of two tuples $\langle 1, 4, data_1\rangle$ and $\langle 3, 2, data_2\rangle$.

The relation is initialized to contain two records: $\langle \bot, \top, -, -, -\rangle$, $\langle -, -, \bot, \top, -\rangle$ which indicates there are no keys inserted for column 1 and column 2 respectively, that is, the table is initialized

[1]Note that, due to factors such as processing delays or network congestion, sequence numbers received by the user may not be in an increasing order.

[2]It is still possible, but with higher cost, to support verifiable query execution without creating a $\langle key, nKey\rangle$ chain for the relevant field: one can perform a sequential scan on the primary key, followed by a Select to find the records satisfying the condition.

(a)

key_1	$nKey_1$	key_2	$nKey_2$	data
$\perp$	$\top$	–	–	–
–	–	$\perp$	$\top$	–

(b)

key_1	$nKey_1$	key_2	$nKey_2$	data
$\perp$	**1**	–	–	–
–	–	$\perp$	**4**	–
1	$\top$	4	$\top$	$data_1$

(c)

key_1	$nKey_1$	key_2	$nKey_2$	data
$\perp$	1	–	–	–
–	–	$\perp$	**2**	–
1	**3**	4	$\top$	$data_1$
3	$\top$	2	4	$data_2$

Figure 6: An example relation that support access methods on multiple column.

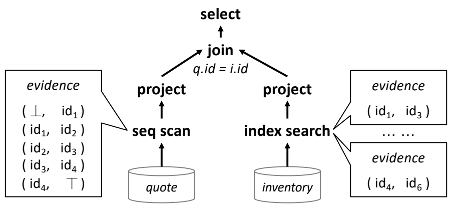


Figure 7: Example query execution plan.

quote

key = id	nKey	data = (count, price)
$\perp$	id_1	(–, –)
id_1	id_2	(100, \$100)
id_2	id_3	(100, \$200)
id_3	id_4	(500, \$100)
id_4	$\top$	(600, \$100)

inventory

key = id	nKey	data = (count, desc)
$\perp$	id_1	(–, –)
id_1	id_3	(50, $desc_1$)
id_3	id_4	(200, $desc_3$)
id_4	id_6	(100, $desc_4$)
id_6	$\top$	(100, $desc_6$)

Figure 8: Example tables.

as two empty key chains. After the insertion of $\langle 1, 4, data_1 \rangle$, the two key chains are updated: the chain for column 1 is updated to $\perp \longrightarrow 1 \longrightarrow \top$, and the chain for column 2 is updated to $\perp \longrightarrow 4 \longrightarrow \top$. Note that, similar to how insertion is executed in a linked list, the records preceding the newly insert record need to update their *nKey* (highlighted in red in Figure 6). Finally, after the insertion of $\langle 3, 2, data_2 \rangle$, the two chains are updated to $\perp \longrightarrow 1 \longrightarrow 3 \longrightarrow \top$ and $\perp \longrightarrow 2 \longrightarrow 4 \longrightarrow \top$.

Performance implication. While the solution is generically applicable to any number of columns, it comes with performance implications. For each write operation (e.g., INSERT and DELETE) that introduces or removes keys, it needs to perform extra operations to update predecessor's *nKey* for each of the maintained key chains. It also increases the overall storage size, as each record needs to maintain both its *key* and its successor's *key*. However, the storage overhead is bounded – in the worst case, where we support access methods on all columns, the storage cost is doubled.

5.4 Execution of SQL queries

VERIDB adopts the classic volcano query execution model: a given SQL query is compiled into a tree-structured execution plan consisting of relational operators, where the leaf nodes are the access methods described in Section 5.2, and the output of a lower-level operator is passed as an input to its parent operator. In VERIDB, the query execution engine resides inside an SGX enclave, and therefore, the output of an operator (such as SELECT, PROJECT, and JOIN) can be trusted if its input is trusted or passes verification. This effectively pushes the verification to the leaf operators, i.e., the access methods.

Furthermore, since the client can offload the verification process to attested programs running in an SGX enclave, s/he can simply verify that the received query results are endorsed by the query execution engine and sent through a secure channel. The query result can be trusted as long as neither the query execution engine nor the storage layer raises verification failure alarms.

Example 5.4. Consider an example query operating on the input database depicted by Figure 8. The query checks the `quote` and `inventory` tables to find all sale quotes that exceed the current balance in the inventory.

```
SELECT q.id, q.count, i.count
FROM quote as q, inventory as i
WHERE q.id = i.id and q.count > i.count
```

Figure 7 depicts the execution plan of the query, and the verification status of the access methods. The JOIN pulls data from the outer relation `quote` through SEQSCAN (on `quote.id`) and from the inner relation `inventory` through INDEXSEARCH (on `inventory.id`). The SEQSCAN, treated as RANGESCAN for range $(\perp, \top)$, first pulls records from the storage. It retrieves $\langle \perp, id_1, - \rangle$ and verifies that $\perp$ is less than or equal to the left-end of the range $(\perp, \top)$.

As the key $\perp$ is outside the range $(\perp, \top)$, it continues to retrieve $\langle id_1, id_2, (100, \$100) \rangle$ and verifies that its $key = id_1$ is the same as its predecessor's $nKey = id_1$. The output $\langle id_1, 100, \$100 \rangle$ is passed to the PROJECT (to remove the `price` column), and then the downstream JOIN. After the JOIN operator receives $\langle id_1, 100 \rangle$ from the outer relation, it pulls the tuple with $key = id_1$ from the inner relation using the INDEXSEARCH operator. The INDEXSEARCH retrieves from the storage layer an `inventory` record $\langle id_1, id_3, (50, desc_1) \rangle$; the record itself proves its existence (i.e., passes the verification). Finally, the tuples pulled from the outer and inner relations are JOINed together and generate $\langle id_1, 100, 50 \rangle$ which satisfies the SELECT conditions and is included in the final output.

This process continues until the SEQSCAN exhausts all records in the `quote` table. Figure 7 shows all the records retrieved by the SEQSCAN. As they satisfy the three verification conditions of range scans, they pass the verification as well. The client considers the the query result as correct, since no verification failure is reported.

The operators in the execution engine, when triggered, output *one* tuple. Therefore, the intermediate state during a query execution is minimal and can be maintained within SGX. However, when the intermediate state is large (e.g., because of introduction of materialization points or aggregations) and beyond the capacity of EPC, it needs to be offloaded to untrusted memory. We can rely on the secure swap of SGX, however, the secure swap can be expensive (because of encryption and decryption). Alternatively, we can reuse the trusted storage of VERIDB for storing the intermediate results (i.e., treat the intermediate state as additional external data). Such approach avoids heavy-weight secure swap, and we plan to explore this direction as a future work.

5.5 Security analysis

As the write-read consistent memory adopts deferred verification, VERIDB does not provide online verification, instead, it achieves a *delayed* version of the *integrity*, *completeness*, and *freshness* properties. We provide a brief security analysis of the integrity and completeness properties; the analysis of the freshness property closely resembles the one of the integrity property, and is omitted in the paper due to space constraints.

Theorem 5.1 (Integrity). *Given any client-specified query Q, each tuple returned to the client satisfies Q; otherwise the breach of the integrity will be (eventually) detected.*

Proof. The result received by the client is authenticated using MAC, any compromise of its integrity will be detected when the client verifies the MAC (see Section 5.1). Therefore, the proof reduces to show the integrity of the query result generated by the execution engine. Given that the execution engine resides in the SGX, the integrity of the output of an operator depends *solely* on the integrity of its inputs. The physical plan of a query forms a tree consisting of relational operators, where the leaf nodes are access methods. The integrity of the query result is reduced to the integrity of the inputs of the access methods (see Section 5.2).

For an access method, it retrieves records from the verifiable storage layer using the SGX-protected APIs (see Section 4.2). Since the actual data is stored outside of the trusted environment, it is susceptible to attacks. However, the write-read consistent memory (see Section 4.1) ensures that any unauthorized modification to the storage (e.g., by bypassing the provided APIs and making direct modification to the memory) will be detected. Therefore, unless the verification reports an alarm, the data access will be performed on a correct database state, that is, the result of following the sequence of writes submitted through the SGX-protected APIs. □

Theorem 5.2 (Completeness). *Given any client-specified query Q, all tuples that satisfy Q are returned to the client; otherwise the omission will be (eventually) detected.*

Proof. The proof of the completeness property follows the similar strategy as the one of integrity. The completeness of the query result is reduced to the completeness of the input of the access methods. For index searches, the completeness is guaranteed by the verifiable storage layer: the absence of a queried record needs to be proved by a non-forgeable evidence stored in the write-read consistent memory. For range scans, the completeness is guaranteed by checking the coverage of the start and end points of the range, and the evidence that the returned results form a no-omission keychain (see Section 5.2). Fundamentally, the non-forgeability of the evidences is guaranteed by the write-read consistent memory. □

6 EVALUATION

In this section, we present a series of experiments to answer the following questions: As micro-benchmarks (Section 6.1), 1) *how much overhead does* VeriDB *incur compared to a baseline that does not support verifiability*, and 2) *how does the internal of non-quiescent verification affect* VeriDB*'s performance, and whether it can support verification frequent enough to be practical?* As a comparative study (Section 6.2), 3) *how does* VeriDB *perform compared against the classic MHT-based approach?* 4) Finally, as a macro-benchmark (Section 6.3), *how is the end-to-end query performance affected by the support of verifiability?*

6.1 Micro-benchmarks

The performance overhead of VeriDB mainly roots from the verification of the write-read consistent memory, which is the foundation of the verifiable storage layer. The main source of overhead comes from 1) the need to update the ReadSet and WriteSet (i.e., $h(\mathcal{RS})$ and $h(\mathcal{WS})$) during each read/write operation, and 2) the need to perform non-quiescent verification. Therefore, we focus on evaluating the overhead from these two sources.

Experiment setup. Each experiment loads an initial state of the database and then runs a mixed set of read/write operations on the database. We use 4-byte integers as keys and 500-byte strings as values. The initial database consists of N key-value pairs, where the keys are in the range of 1 ... N and the values are generated randomly. Unless otherwise stated, we used $N = 1$ million pairs as the initial database size. In our experiments, we use the following four kinds of operations:

- Update: Given a specified key, it updates the corresponding data record in the database.
- Insert: It inserts a data record to the database, and updates the *nKey* field of the preceding record.
- Delete: It deletes a data record (with a specified key) from database, and updates the *nKey* field of the preceding record.
- Get: Given a specified key, it retrieves the corresponding data record from database.

The detailed description of these operations are presented in Section 4.2. Unless otherwise specified, we run the experiments as follows: each experiment is repeated three times and in each of these three runs 10 thousands operations in total, where the number of four kinds of operations are approximately the same. The latency is measured as the average latency over time and across the three runs. Our experiments are carried out on a server equipped with Intel Xeon CPU E3-1270 v6 @ 3.80GHz CPU and 64 GB of RAM.

Overhead for maintaining ReadSet/WriteSet. Our first set of experiments evaluate the latency of the aforementioned four kinds of read/write operations. We consider four configurations: a) Baseline which performs the read/write operations without additional mechanisms to guarantee verifiability; b) RSWS which provides verifiability guarantee for data records but not page metadata; and c) RSWS incl. metadata which additional ensures the verifiability of page metadata. Figure 9 presents our evaluation results.

We make the following observations: First, we notice that the cost introduced by the update of the ReadSet and WriteSet is relatively smaller (ranging between 1.5-2.2 us), and can be reduced if we exclude the page metadata from verification (recall that the correctness verification of client's query relies only on the integrity of actual data records). The average cost for updating the ReadSet and WriteSet is reduced by around 20% after we removed the page metadata verification. The operations of ReadSet and WriteSet add around 1.5 us for Get and Delete, and around 2.2 us for Insert and Delete. This difference is mainly because the Insert and Delete operations need to update the *nKey* field of preceding records, causing additional updates on the ReadSet and WriteSet.

Second, we notice that this latency overhead for supporting verifiable storage layer is dominated almost exclusive by PRF operations (required by updating $h(\mathcal{RS})$ and $h(\mathcal{WS})$). By adopting hardware solutions such as FPGA, the hash speed can be significantly improved (potentially by orders of magnitude), making VeriDB a more appealing solution that provides verifiability with little performance penalty.

Verification. Our next set of experiments focus on the performance impact introduced by the non-quiescent verification. The cost of the non-quiescent verification mainly comes from two sources: a) the need to lock a page when it scans that page, preventing race conditions with routine read/write operations, and b) the resources

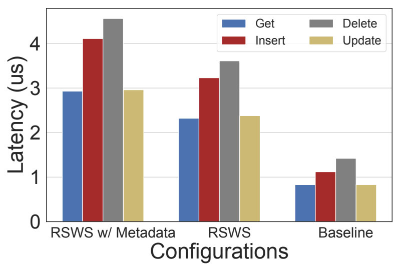


Figure 9: Latency of reads/writes with different system config.

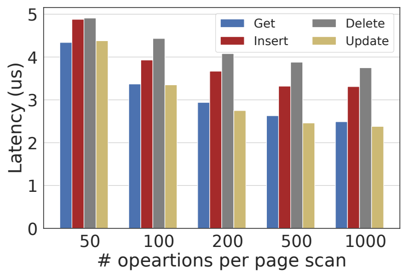


Figure 10: Latency of reads/writes with different verification freq.

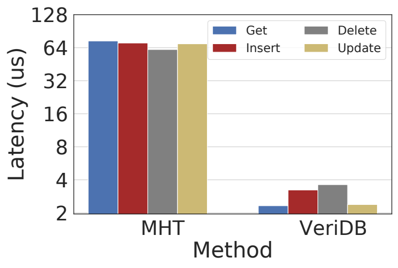


Figure 11: Latency of reads/writes for MB-tree and VeriDB

needed to update the ReadSet during the scan. Figure 10 presents the latency of read/write operations when the non-quiescent verification is performed with different frequencies. The configurations x have the background verification thread always running, and perform a memory scan after *x* operations.

We observe that, as the verification is invoked more frequently, the latency of the read/write operations also becomes larger. It is understandable, a more frequent memory scan requires more resources to read page data, compact the records (see Section 4.3) and update the ReadSet; it also locks pages more often and creates a higher chance of data contention that prevents routine read/write operations. We posit that a verification frequency of 1,000 operations introduces a small overhead (1-4% overhead compared to the BASELINE, i.e., the RSWS data in Figure 9) is frequent enough in practice. To summarize, VERIDB provides verifiability for read-/write operations with reasonable overhead that is insignificant enough to be adopted in practical scenarios.

6.2 VERIDB vs. MB-Tree

We next present a comparative evaluation that compares VERIDB's performance with that of MB-Tree [14], a classic MHT-based approaches for database verification. It is a representative example that reveals the general limitation of MHT-based approaches – the concurrency bottleneck at the root hash in an MHT. In an MB-Tree, read/write operations in data records are not performed in an SGX environment; the system does not maintain the ReadSet and WriteSet, instead, it maintains a MHT on top of the data records. For example, each write operation requires updating the path from the MHT's leaf node affected by the write to the root, and each read operation requires the return of an ADS that allows regeneration of the MHT's root hash. We continue to use workload introduced in Section 6.1, that is, performing 100K mixed read/write operations on a database initialized with an init state. Figure 11 shows the latency of the four kinds of read/write operations executed on VERIDB and MB-tree (note the logarithmic scale on the y-axis). In this experiment, we enable non-quiescent verification with a frequency of verifying one page every 1,000 operations.

We observe that VERIDB significantly outperforms MB-Tree, yielding a 94-96% reduction in read/write latency. This agrees with evaluation results reported in prior work [1]: the concurrency bottleneck in MHT-based approaches largely affect system performance. Admittedly, MB-Tree provides online verification, meaning the returned result can be directly verified against its associate ADS. We posit that, while VERIDB adopts a delayed verification (detection happens at the end of a memory scan), VERIDB is an appealing alternative given its significant performance boost, and its support for a more general class of SQL queries.

6.3 Macro-benchmark

TPC-H benchmark. We evaluate the query performance of VERIDB using the TPC-H benchmark. The TPC-H benchmark contains 22 SQL queries and a dataset, and is widely used by the database community to evaluate the performance of new systems. We evaluate the performance of VERIDB using Query #1, #6 and #19 in the TPC-H benchmark; take Query #19 as an example: it is a SUM on the result of a JOIN query applied to two multidimensional range queries on tables `lineitem` and `part`. In our evaluation, we consider two configurations: BASELINE that excludes all needed security enhancements for verifiability (e.g., the maintenance of the ReadSet and WriteSet, the non-quiescent verification, and the enhanced access methods), and VERIDB that provides verifiability.

Figure 12 shows the evaluation results, we make two observations: 1) VERIDB introduces a consistently reasonable overhead across different queries, and 2) the overhead is dominated by the overhead of table scans, during which VERIDB performs the update of ReadSet and WriteSet. For Queries #1 and #6 which both scan the *lineitem* table once, the overhead on both queries is about 6.7 seconds. Query #19 is a JOIN over two tables. We considered two query plans: one uses MERGEJOIN, and the other uses NESTEDLOOPJOIN and materialize the SELECT result on inner loop. The cost of data access remains the same: VERIDB scans both the *part* and *lineitem* tables once and the overhead is about 7.1 seconds. On the other hand, the computation cost of MERGEJOIN is significantly lower, although it introduces a larger intermediate state to store sort results. Importantly, we observe that the query execution engine, while resides in an SGX enclave, introduces no additional overhead. Therefore, for queries with computational bottleneck like Query #19 (with NESTEDLOOPJOIN), the relative overhead is only 9%; for simple SELECT queries like Query #1 and #6, the relative overhead can be up to 39%.

TPC-C benchmark. We also evaluate the performance of VERIDB when processing concurrent reads/writes using the TPC-C benchmark. We measure VERIDB's average throughput on a 20-warehouse configuration when varying the number of clients and the number of ReadSets/WriteSets (RSWSs). Figure 13 shows our evaluation result. We observe that VERIDB has a similar performance trend compared to an ordinary database which does not protect data integrity: both systems reach a peak throughput when the number

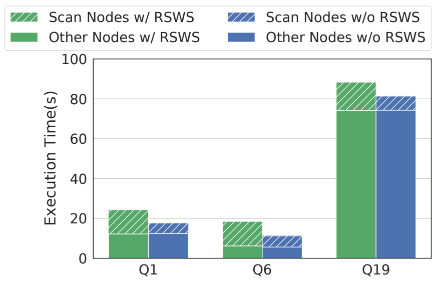


Figure 12: Execution time on TPC-H dataset.

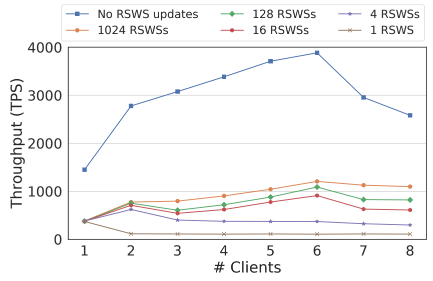


Figure 13: Transaction throughput on TPC-C dataset.

of clients is 6, and the throughput decreases afterwards. Unlike MHT-based approaches, VeriDB, with sufficiently many RSWSs, does not introduce extra concurrency bottlenecks. We further observe that increasing the number of RSWSs significantly reduces the performance overhead caused by the lock contention as we expect in Section 4.3. Overall, VeriDB with 1024 RSWSs introduces an overhead of around 3x-4x on transaction throughput, as VeriDB needs to update the hashes of the read and write sets.

7 RELATED WORK

The problem of verifiable cloud databases has been extensively studied to ensure result integrity against an untrusted cloud service provider. Generally speaking, the techniques to achieve integrity can be categorized into three groups: Authenticated Data Structure (ADS), Succinct Arguments of Knowledge (SNARK), and Trusted Execution Environment (TEE).

ADS-based approach. ADS is a classic technique to ensure query integrity. There are two commonly-used building blocks for ADS: digital signatures that use asymmetric cryptography to authenticate any piece of data, and Merkle Hash Trees (MHT) that use hierarchical hashes to verify the existence of any record(s) in an array of data. The ADS-based approach typically handles only a specific class of computations on the data, e.g., range queries [14, 22], joins [29, 32], pattern matching [7, 23], and a subset of SQL queries [31]. They have limited applicability and cannot serve as full-functional databases.

SNARK-based approach. SNARK is a technique that verifies the execution result of arbitrary non-deterministic arithmetic circuits. Therefore, SNARK-based approach can support arbitrary computation tasks, but at the expense of an extremely high performance overhead. In particular, vSQL [30] designs a verifiable database system that can support arbitrary SQL queries with an interactive protocol. However, it is limited to relational databases with a fixed schema. Nevertheless, the impractical overhead prevents these solutions from being widely adopted in real world scenarios.

TEE-based approach. TEE serves as an additional trust anchor, and allows simplification and performance improvement in the design of verifiable database. For example, CorrectDB [4] considers moving the database into an environment protected by a trusted CPU, and VeritasDB [26] proposes a network proxy running in an SGX enclave that mediates communication between the client and the database server. They both rely on MHT-based ADS for integrity verification, and suffer from the problem that MHT root inevitably becomes a concurrency bottleneck. Concerto [1] adopts a radically different approach and removes the concurrency bottleneck in MHT-based designs. It reduces the problem of verifying data integrity to that of verifying memory integrity, which can be efficiently achieved using a high-performance memory checking algorithm. However, it only supports key-value storage model. Opaque [33] is an SGX-based oblivious distributed analytics platform that also ensures integrity in query processing, but its main focus is on access pattern leakage protection which introduces large overhead and therefore not comparable to our system.

Verifiable databases on multi-parties. There are also related work that aims for data and query integrity in an untrusted cloud environment, but assume that there are multiple clients and servers in the scenario. Some approaches achieve a similar goal as our paper that the clients want to outsource the database without storing the full data. These works typically use ADS to achieve integrity and use blockchain to synchronize the digest of ADS [24, 28]. Therefore, they also suffer from the functional limitation of the underlying ADS, and in addition, the performance overhead introduced by blockchain consensus protocol.

Another way to achieve integrity is to keep a local copy of the full data and ensure that the local data is up-to-date. This goal can be achieved by either blockchain [8] or TEE [15]. As a result, they can achieve fault tolerance in addition to fault detection, at the cost of large per-node storage overhead.

8 CONCLUSION

To conclude, we introduce in this paper the design and implementation of VeriDB, an SGX-based verifiable database that supports relational tables, multiple access methods and general SQL queries. VeriDB adopts a page-structured storage design, to facilitate the integration with legacy relational databases and to support a wide-range of verifiable access methods. By moving the query engine into an SGX enclave, VeriDB reduces the problem of verifying the integrity of query results to verifying the correctness and completeness of retrieved data, which can be efficiently achieved with the support of the verifiable storage. Using our prototype implementation, we demonstrate that VeriDB incurs reasonable overhead for achieving verifiability: VeriDB introduces an overhead of 1-2 microseconds for read/write operations, which allows it to be adopted for logically simple queries that require low latency. VeriDB introduces a 9%-39% overhead for analytical workloads. Overall, We posit that such overhead is a reasonable tradeoff for achieving verifiability in practical scenarios.

ACKNOWLEDGMENTS

This work was supported by Alibaba Group through Alibaba Research Fellowship Program.

REFERENCES

[1] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 251–266.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 689–703.

[3] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, 173–190.

[4] Sumeet Bajaj and Radu Sion. 2013. CorrectDB: SQL Engine with Practical Query Authentication. *Proc. VLDB Endow.* 6, 7 (2013), 529–540.

[5] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. 1991. Checking the Correctness of Memories. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. IEEE Computer Society, 90–99.

[6] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 86 (2016), 1–118.

[7] Premkumar T. Devanbu, Michael Gertz, April Kwong, Charles U. Martel, Glen Nuckolls, and Stuart G. Stubblebine. 2001. Flexible authentication of XML documents. In *ACM Conference on Computer and Communications Security*. ACM, 136–145.

[8] Johannes Gehrke, Lindsay Allen, Panagiotis Antonopoulos, Arvind Arasu, Joachim Hammer, James Hunter, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Srinath T. V. Setty, Jakub Szymaszek, Alexander van Renen, Jonathan Lee, and Ramarathnam Venkatesan. 2019. Veritas: Shared Verifiable Databases and Tables in the Cloud. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org.

[9] Intel. [n.d.]. Intel(R) SGX SDK Developer Reference for Windows. https://software.intel.com/content/www/us/en/develop/download/sgx-sdk-developer-reference-windows.html. March 2020.

[10] Intel. [n.d.]. Intel(R) Software Guard Extensions (Intel SGX). https://software.intel.com/sites/default/files/332680-002.pdf. June 2015.

[11] Intel. [n.d.]. Intel(R) Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. 2014.

[12] Rohit Jain and Sunil Prabhakar. 2013. Trustworthy data from untrusted databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 529–540.

[13] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. 2016. Verena: End-to-End Integrity Protection for Web Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*. 895–913.

[14] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. ACM, 121–132.

[15] Kai Mast, Lequn Chen, and Emin Gün Sirer. 2018. Enabling Strong Database Integrity using Trusted Execution Environments. *CoRR* abs/1801.01618 (2018). arXiv:1801.01618 http://arxiv.org/abs/1801.01618

[16] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, USA, 1289–1306.

[17] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*. ACM, 10.

[18] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science)*, Vol. 293. Springer, 369–378.

[19] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. 2004. Authentication and Integrity in Outsourced Databases. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*. The Internet Society.

[20] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*. 238–253.

[21] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. 2005. Verifying Completeness of Relational Query Results in Data Publishing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. ACM, 407–418.

[22] Dimitrios Papadopoulos, Stavros Papadopoulos, and Nikos Triandopoulos. 2014. Taking Authenticated Range Queries to Arbitrary Dimensions. In *ACM Conference on Computer and Communications Security*. ACM, 819–830.

[23] Dimitrios Papadopoulos, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2015. Practical Authenticated Pattern Matching with Optimal Proof Size. *Proc. VLDB Endow.* 8, 7 (2015), 750–761.

[24] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. 2020. FalconDB: Blockchain-based Collaborative Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 637–652.

[25] Sarvjeet Singh and Sunil Prabhakar. 2008. Ensuring correctness over untrusted private database. In *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings (ACM International Conference Proceeding Series)*, Vol. 261. ACM, 476–486.

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

[27] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 81–93.

[28] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases. In *SIGMOD Conference*. ACM, 141–158.

[29] Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. 2009. Authenticated join processing in outsourced databases. In *SIGMOD Conference*. ACM, 5–18.

[30] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 863–880.

[31] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2015. IntegriDB: Verifiable SQL for Outsourced Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. ACM, 1480–1491.

[32] Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. 2012. Efficient query integrity for outsourced dynamic databases. In *CCSW*. ACM, 71–82.

[33] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*. USENIX Association, 283–298.