VeriDB: An SGX-based Verifiable Database

Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, Feifei Li
Motivation

- For cloud storage and cloud computing
  - The integrity of storage and computation relies on the “trust” from users to the cloud service provider.
Motivation

- For sensitive data
  - The cloud service provider needs to give some proof of the correctness.
- Or detect unexpected behaviors
Motivation

- The cloud may tamper with data.

What is my salary in March? It's $4000 😈

<table>
<thead>
<tr>
<th>Table “Salary”</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Month</strong></td>
</tr>
<tr>
<td>Jan</td>
</tr>
<tr>
<td>Feb</td>
</tr>
<tr>
<td>Mar</td>
</tr>
</tbody>
</table>
Motivation

- The cloud may even return falsified results without tampering with data

Tell me what kinds of fruits are worth more than $1.60!

Only apples 😈

<table>
<thead>
<tr>
<th>Item</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>2.00</td>
</tr>
<tr>
<td>Banana</td>
<td>1.50</td>
</tr>
<tr>
<td>Peach</td>
<td>1.80</td>
</tr>
</tbody>
</table>
Scenarios and Goals

Scenario

- Cloud-client
- Existence of TEE (SGX)
- Untrusted Cloud Service Provider: Byzantine Behavior

Goals

- For integrity
  - Endorsement of correct results
  - Detection of incorrect results
- For applicability
  - Support for general SQL queries
  - Low overhead
Strawman Solutions

- Use Merkle Hash Tree (MHT) to verify the integrity of data
  - The root hash would be a concurrency bottleneck
- Store all data in trusted memory
  - EPC (enclave page cache) is a scarce resource
  - Expensive swapping if EPC not enough
- Introduce significant overhead
Contribution

- Verifiable storage and execution
- Support verifiable general SQL queries
- Reasonable performance overhead
Outline

- Introduction
  - Motivation
  - Scenario and goals
  - Contributions

- VeriDB
  - Architecture
  - Verifiable storage and data access
  - Optimizations

- Evaluations
Architecture

- Data stored in untrusted memory
- Read/Write primitives are stored in trusted memory
  - Ensures the integrity of storage.
Architecture

- Interface between storage and execution
  - Reduce verifying results into verifying storage and trusted execution
- The client communicates with the query portal via a secure channel
Verifiable Storage

- Basic idea: read-write consistent memory\(^1\)
  - The contents got from “read” must be the contents of the latest “write”
  - Maintain a read set and a write set
  - Update two sets on memory operations
  - Check if the two sets are consistent

Verifiable Storage

- Construct a hash of tuple \( h(\text{addr}, \text{data}, \text{timestamp}) \) on each operation.
- Update the sets by xor the hashes\(^2\)
- Periodically,
  - The verifier reads each datum and adds to the read set.
  - Verify that ReadSet == WriteSet, otherwise throw an alarm.

Verifiable Data Access

- Key-chain of records in the table
  - Store (key, nextKey) tuples
  - Prove the existence / absence of a queried record
    - Absence of \( id_2 < qid < id_3 \) is proved by \((id_2, id_3, data)\)

<table>
<thead>
<tr>
<th>id</th>
<th>count</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>id_1</td>
<td>100</td>
<td>$100</td>
</tr>
<tr>
<td>id_2</td>
<td>100</td>
<td>$200</td>
</tr>
<tr>
<td>id_3</td>
<td>500</td>
<td>$100</td>
</tr>
<tr>
<td>id_4</td>
<td>600</td>
<td>$100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>nextKey</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>┐</td>
<td>id_1</td>
<td>(−,−)</td>
</tr>
<tr>
<td>id_1</td>
<td>id_2</td>
<td>(100, $100)</td>
</tr>
<tr>
<td>id_2</td>
<td>id_3</td>
<td>(100, $200)</td>
</tr>
<tr>
<td>id_3</td>
<td>id_4</td>
<td>(500, $100)</td>
</tr>
<tr>
<td>id_4</td>
<td>⊤</td>
<td>(600, $100)</td>
</tr>
</tbody>
</table>
Verifiable Data Access

- Three principles to ensure the integrity for range queries \([\text{startKey}, \text{endKey}]\)
  - We don’t miss anything in the beginning
    - Find the first row where \(\text{row.nextKey} \geq \text{startKey}\), and start from the next row
  - We reach the expected last row
    - \(\text{lastRow.nextKey} > \text{endKey}\)
  - All rows are chained
    - \(\text{thisRow.key} = \text{prevRow.nextKey}\)

- Example: \(\text{SELECT * FROM data WHERE key} \geq \text{id1} \text{ AND key} \leq \text{id3}\)
- Verifiable storage + verifiable data access = correct results

<table>
<thead>
<tr>
<th>key</th>
<th>nextKey</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>id₁</td>
<td>(−, −)</td>
</tr>
<tr>
<td>id₁</td>
<td>id₂</td>
<td>(100, $100)</td>
</tr>
<tr>
<td>id₂</td>
<td>id₃</td>
<td>(100, $200)</td>
</tr>
<tr>
<td>id₃</td>
<td>id₄</td>
<td>(500, $100)</td>
</tr>
<tr>
<td>id₄</td>
<td>⊥</td>
<td>(600, $100)</td>
</tr>
</tbody>
</table>
SELECT o.id, o.count
FROM orders as o, inventory as i
WHERE o.id = i.id, o.count <= i.count
SELECT o.id, o.count
FROM orders as o, inventory as i
WHERE o.id = i.id, o.count <= i.count

<table>
<thead>
<tr>
<th>id</th>
<th>count</th>
<th>price</th>
<th>nextid</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>200</td>
<td>$100</td>
<td>id2</td>
</tr>
<tr>
<td>id2</td>
<td>100</td>
<td>$200</td>
<td>+inf</td>
</tr>
<tr>
<td>-inf</td>
<td>-</td>
<td>-</td>
<td>id1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>count</th>
<th>desc</th>
<th>nextid</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>300</td>
<td>desc1</td>
<td>id3</td>
</tr>
<tr>
<td>id3</td>
<td>800</td>
<td>desc3</td>
<td>+inf</td>
</tr>
<tr>
<td>-inf</td>
<td>-</td>
<td>-</td>
<td>id1</td>
</tr>
</tbody>
</table>
Optimizations

- Use multiple RSWSs to avoid lock contention
  - Operations on addr1, addr2, and addr3
  - Separate the sets during update
  - Combine the sets and compare during verification

- Other optimizations
  - Avoid scanning unvisited pages
  - Excludes page metadata from verification
  - Compaction during verification
Outline

- Introduction
  - Motivation
  - Scenario and goals
  - Contributions

- VeriDB
  - Architecture
  - Verifiable storage and data access
  - Optimizations

- Evaluations
Each update of read set (RS) and write set (WS) introduces 1.5 – 2.2 μs overhead

- Hash operations make up most of the extra overhead

- “Insert” and “delete” need updates to the “nextKey” field, thus take longer time

- The verification process only introduces slight overhead
Evaluations – v.s. MB-Tree

- VeriDB significantly outperforms MB-Tree\(^3\), an MHT-based approach
  - MB-Tree involves more hash calculations
  - The root hash of MB-Tree becomes the bottleneck of concurrency

---

Evaluations – Macro-benchmark

- Queries (TPC-H)
  - Q1 and Q6, scan, filter, and aggregate;
  - Q19, scan, filter, and join
- The performance overhead mainly comes from the scan operators.
- Overall, VeriDB introduces 9%~39% overhead.

- Other macro-benchmark results: TPC-C
## Related Work

<table>
<thead>
<tr>
<th>System</th>
<th>Support</th>
<th>Trust Model</th>
<th>Overhead</th>
<th>Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concerto</td>
<td>Key-value</td>
<td>Cloud-user</td>
<td>Relatively Low</td>
<td>SGX + Verifiable memory</td>
</tr>
<tr>
<td>EnclaveDB</td>
<td>Relational</td>
<td>Cloud-user</td>
<td>High (All in SGX)</td>
<td>SGX</td>
</tr>
<tr>
<td>VeritasDB</td>
<td>Key-value</td>
<td>Cloud-user</td>
<td>High (MHT)</td>
<td>MHT + ADS</td>
</tr>
<tr>
<td>FalconDB</td>
<td>Relational</td>
<td>Multi-users</td>
<td>High (Blockchain)</td>
<td>Blockchain + ADS</td>
</tr>
<tr>
<td>VeriDB</td>
<td>Relational</td>
<td>Cloud-user</td>
<td>Relatively Low</td>
<td>SGX + Verifiable memory</td>
</tr>
</tbody>
</table>
Conclusion

- **VeriDB**: an SGX-based verifiable database that supports relational tables and general SQL queries.

- **Methods**: reduce the problem of providing verified results to ensuring verifiable storage and verifiable access.

- **Performance**: ≤ 2.2 μs overhead for read/write operators and 9%-39% for analytical workloads