VeriTxn: Verifiable Transactions for Cloud-Native Databases with Storage Disaggregation

ZHANHAO ZHAO*, Renmin University of China, China
HEXIANG PAN, National University of Singapore, Singapore
GANG CHEN, Zhejiang University, China
XIAOYONG DU, Renmin University of China, China
WEI LU, Renmin University of China, China
BENG CHIN OOI, National University of Singapore, Singapore

Cloud-native databases become increasingly popular while exposing to greater data security and correctness risks. Existing verifiable outsourced databases overlook either the correctness risk of transactions, or the disaggregation architecture: a key design consideration of cloud-native databases for performance and elasticity, or both. We present VeriTxn, a novel cloud-native database that efficiently provides verifiability of transaction correctness. VeriTxn relies on the trusted hardware (i.e., Intel SGX) to enable verifiable transaction processing. We build a page-structure cache in the trusted domain, where transactions can be verified with low, constant overhead. VeriTxn further optimizes the read-only transactions by exploiting disaggregation to fit the read-heavy workload in the cloud. We also integrate our proposal into MySQL, a popular open-source database. We conduct extensive experiments to compare VeriTxn against state-of-the-art verifiable databases and evaluate the performance of VeriTxn on MySQL. The results show that VeriTxn introduces tolerable performance degradation for verifiable transactions, while achieving up to 7.03× and 7.93× higher throughput than Litmus and LedgerDB, and its sustainable performance when integrated with MySQL.

CCS Concepts: • **Information systems** → **Database transaction processing**; • **Security and privacy** → **Database and storage security**.

Additional Key Words and Phrases: Cloud Database, Disaggregation, Verifiable Transaction

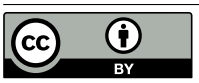
ACM Reference Format:
Zhanhao Zhao, Hexiang Pan, Gang Chen, Xiaoyong Du, Wei Lu, and Beng Chin Ooi. 2023. VeriTxn: Verifiable Transactions for Cloud-Native Databases with Storage Disaggregation. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 270 (December 2023), 27 pages. https://doi.org/10.1145/3626764

1 INTRODUCTION

In recent years, organizations such as banks [21, 49] and governments [9] are increasingly moving their sensitive and critical data into cloud databases. Such outsourcing requires cloud databases to guarantee data integrity and ensure the correctness and safety of each transaction. However, the results returned from the cloud can be incorrect or without any evidence on their correctness. For example, an attacker can intercept clients' requests and pretend to be a cloud service provider by

* This work was done while this author was at National University of Singapore.

Authors' addresses: Zhanhao Zhao, Renmin University of China, China, zhanhaozhao@ruc.edu.cn; Hexiang Pan, National University of Singapore, Singapore, panh@u.nus.edu; Gang Chen, Zhejiang University, China, cg@zju.edu.cn; Xiaoyong Du, Renmin University of China, China, duyong@ruc.edu.cn; Wei Lu, Renmin University of China, China, lu-wei@ruc.edu.cn; Beng Chin Ooi, National University of Singapore, Singapore, ooibc@comp.nus.edu.sg.



2836-6573/2023/12-ART270
https://doi.org/10.1145/3626764

simply returning empty results for any requests. To this end, transactions in the cloud have to be *verifiable* so that the correctness of any returned results is verifiable.

Cloud-native databases are databases specifically designed to exploit the elasticity, scale, resiliency, efficiency, and flexibility provided by the cloud. To achieve good efficiency and elasticity, the database design has to feature a disaggregation architecture, where the computation and storage are decoupled as two distinct components connected by the high-speed network [1]. A number of cloud-native databases adopt such an architecture, including Azure SQL [4], Aurora [68], Taurus [32], PolarDB [17], and Snowflake [27]. Various techniques have been proposed to speed up these disaggregated databases, including customized storage management [16, 75], and query processing with caching and pushdown [71, 74]. For verifiability, several approaches have been proposed to ensure data integrity by encrypting queries and verifying their results [3, 5]. However, such approaches cannot verify whether transactions execute correctly without compromising ACID properties, and therefore, the problem of ensuring the verifiability of transactions in the cloud remains open.

To support verifiable transactions, existing solutions either provide serializability checking ability [2, 64], or heavily customize the transaction processing protocols [69] and data structures [42]. Consequently, they are not able to exploit the efficiency provided by disaggregation. On the one hand, serializability checking relies on tracing dependencies among the transactions. Extracting dependencies from logs is costly, especially in cloud-native databases where multiple nodes can handle transactions individually, and hence may affect the database scalability. On the other hand, customization-based approaches introduce certain constraints to the storage and transactions. For example, Litmus [69] requires the data to be stored in memory to enable verification by the memory integrity checker. Several classic approaches propose to organize the data using Merkle Hash trees [50], where each update requires the reconstruction of the tree. However, the cloud storage typically has its own implementation and cannot be modified easily, and hence the Merkle tree reconstruction is hard to implement. All these constraints introduce complexity and performance penalty when these techniques are directly employed in cloud-native databases.

Different from existing works, we turn to trusted execution environments (TEE) such as Intel SGX (Software Guard Extensions) [24] to efficiently support verifiability for transactions. Among the available hardware technologies that support TEE implementation, we shall use SGX as the default hardware in this paper. Succinctly, SGX provides a protected execution environment, known as an enclave, within potentially compromised nodes in the cloud, shielding data and attested programs in the enclave from malicious manipulation. Due to its functionalities, various data management systems including key-value stores [7] and outsourced databases [58, 60] are built with SGX for security and data integrity. However, in the past years, TEE suffers from a limited memory capacity of up to 256 MB and high overheads of performance. As a consequence, researchers proposed various methods to overcome these limitations, e.g., by only putting required data in this restricted environment [7, 58, 63, 78]. However, with the GB capacity of current SGX [34], these optimizations become less effective. Besides, the earlier designs do not exploit the disaggregation architecture. In agreement with the recent database self-assessment report [1], it is important to design a verifiable transaction processing protocol for a disaggregation architecture that exploits TEE for efficiency.

In this paper, we propose VeriTxn, a cloud-native database that efficiently supports verifiable transactions. We disaggregate the transaction layer and encapsulate it in SGX, i.e., transactions are entirely executed in the enclave. To this end, we design a page-structure cache, called a verified cache, which is maintained in the enclave. Each transaction only accesses the verified cache for reading/writing the data items. We subsequently propose two techniques to ensure efficient and verifiable transaction processing by exploiting the memory capacity of today SGX. First, we achieve verifiability by introducing a server-side verification mechanism. We delegate the verification

to the attested code residing in the trusted domain, removing the need to ship large-size proofs over the network between the server side and the client, as well as offloading the client's burden of verifying the proofs. This server-side verification is coarse-grained, i.e., we verify each data page instead of each data item accessed by a transaction. Consequently, each transaction has a bounded cost corresponding to the number of pages it accessed. Enabled by this page-level verification mechanism, our proposed verifiable transaction protocol is generally applicable to databases employing a page-structured storage model. Second, we design a double-layer cache management, consisting of the verified cache in the enclave and the data cache in the untrusted domain. The verified cache interacts with the data cache, while the data cache interacts with the disaggregated cloud data storage. We employ a hybrid cache replacement policy, i.e., a lazy policy for the verified cache, while an eager policy for the data cache. With this approach, we cache data in the verified cache and ensure the freshness of the data cache as much as possible.

By disaggregating the storage and computation, we ensure the scalability for running verifiable read-only transactions. VeriTxn can add more individual compute nodes to handle read-only transactions, which is well-suited for read-heavy workloads in the cloud. The main challenge is then to ensure the verified cache coherence in these nodes, of which we address by leveraging an asynchronous replication mechanism to periodically replicate the verified cache between compute nodes, making a tradeoff between data freshness and transaction latency.

In summary, we make the following contributions:

- We present VeriTxn, a cloud-native database that guarantees verifiable transactions. VeriTxn processes transactions in SGX with a disaggregation architecture for elasticity and efficiency.
- We propose the server-side verification mechanism to ensure efficient verification of transactions. The key data structure is the SGX-bounded verified cache, which enables transactions to be verified with low, bounded overhead.
- We introduce optimization strategies for read-only transactions while providing verifiability guarantees. This enables VeriTxn to scale out linearly.
- We conduct extensive evaluations on two popular benchmarks, namely YCSB and TPC-C, and compare VeriTxn against state-of-the-art verifiable databases. The results show that VeriTxn is efficient, and it outperforms the baselines by up to 7.93×.
- We integrate VeriTxn into MySQL and the performance evaluation confirms its robustness in MySQL variants.

The remainder of the paper is structured as follows. The next section provides relevant background on cloud-native databases and Intel SGX, and presents the problem statement. Section 3 describes the threat model and an overview of VeriTxn. Section 4 details the design of VeriTxn, including the verifiable transaction processing, etc. Section 5 introduces the tampering recovery technique and a comprehensive security analysis. Section 6 describes the system implementation, and Section 7 presents the experimental results. Section 8 discusses the related works, and Section 9 concludes.

2 BACKGROUND AND PROBLEM DEFINITION

In this section, we describe cloud-native databases and Intel SGX, and state the problem of supporting verifiable transactions.

2.1 Cloud-Native Databases

Modern cloud-native databases decouple the computation from storage and manage them as two separate layers of services [4, 17, 27, 32, 68]. Such disaggregation can reduce operational costs and improve resource utilization because it allows computation and storage to scale and be charged

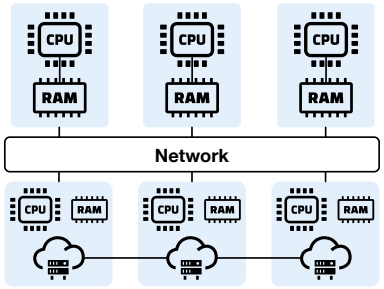


Fig. 1. Cloud-Native Databases with Disaggregation

independently. Figure 1 illustrates the architecture of a cloud-native database, which consists of two components: (1) a persistent storage layer hosts the actual data and write-ahead logs for achieving fault tolerance; (2) a compute layer that is responsible for SQL execution, transaction management, recovery daemon, etc. Unlike legacy databases, the compute layer must access the storage layer through the network because of the disaggregation. In our design, we minimize inter-layer communication through customizations in transaction processing and cache management, which will be detailed in Section 4.

2.2 Intel SGX

Intel Software Guard Extensions (SGX) [24] is a hardware-based implementation of the trusted execution environment (TEE), which enables the trusted processing of private data. SGX can be used to protect the execution of applications and ensure data integrity in cloud environments because of the following two key features: **Isolation:** SGX reserves several private memory regions called *enclave*, which are isolated from the rest of the host. Both the code and the data in the enclave are protected from being accessed by processes outside the enclave, even by the operating system or hypervisor. SGX achieves this by locating the enclave memory in protected memory pages called enclave page caches (EPC). **Attestation:** SGX enables a remote client to verify that the code is executed as expected inside the enclave, which is called attestation. Upon successful attestation, the remote client can bootstrap a secure communication channel with the enclave. As an example, a client that initiates a transaction can use such a secure channel to guarantee the safe delivery of the transaction to the enclave.

In the latest generation of SGX, the EPC capacity has increased from the previous 256 MB to 512 GB per socket, making the memory limitation, as reported in previous works [34, 63, 78], no longer a major constraint. However, a critical limitation remains: interactions between host processes and enclaves are still expensive. Hosts can only interact with the enclave via pre-defined functions (`ECalls`/`OCalls`). That is, hosts call into the enclave via `ECalls`, while the enclave calls the code outside (such as system calls) via `OCalls`. However, these functions are costly. As an example, an `ECall` incurs about 8000 CPU cycles as reported in [56, 63]. Moreover, issues like paging overhead occur when the EPC is exhausted [34]. Consequently, we design an SGX-friendly transaction processing protocol to minimize costly intersections between the enclaves and hosts, which will be presented in Section 4.2.

2.3 Verifiable Transactions

Broadly, a database is deemed supporting verifiable transactions when it meets these two requirements: (1) It guarantees the ACID properties [38], and (2) it ensures that these guarantees can be verified. Due to our disaggregation architecture design, the verifiable durability is hence considered

a property guaranteed by the storage layer. In this section, we shall therefore focus on the formulation of verifying isolation, atomicity, and consistency. It is well accepted that serializability is the gold-standard isolation level [13, 14, 38], which enables each transaction to move the database from one consistent state to another [38]. Further, atomicity means that each state is stable. Based on these intuitions, we formulate the verifiable transaction scheme by specifying the state and state transition.

Formally, we model a database state as $\mathbb{S}$ with respect to a set of versions for all the data items (x_i represents the i-th version of the data item x) in the database. A transaction T_i is a sequence of operations, which are either read $R_i(x_j)$, write $W_i(x_i)$, commit C_i or abort A_i. We use $R_i(x_j)$ to denote a T_i's read that obtains the version x_j written by T_j. We use $\mathbb{R}_{T_i}$ ($\mathbb{W}_{T_i}$) to denote the read set (write set) of T_i, which contains the versions read (wrote) by T_i. Let us consider the database state when a transaction T_i starts as $\mathbb{S}$, and a new state $\mathbb{S}^+$ is generated once T_i commits. We model a transaction T_i as the function shown in Equation 1, indicating that the new state $\mathbb{S}^+$ is identical to a union set of the old state $\mathbb{S}$ and the write set of T_i.

$$T_i : \mathbb{S} \rightarrow \mathbb{S}^+ \equiv \exists\, x_i, x_i \in \mathbb{S}^+ \wedge x_i \notin \mathbb{S} \Rightarrow x_i \in \mathbb{W}_{T_i} \tag{1}$$

Example 1. Consider a transaction T_1 with two operations $R_1(x_0)$ and $W_1(y_1)$, $\mathbb{S} = \{x : x_0, y : y_0\}$. It can be obtained that $\mathbb{R}_{T_1} = \{x_0\}$, $\mathbb{W}_{T_1} = \{y_1\}$. After T_1 commits, the new state is $\mathbb{S}^+ = \{x : x_0, y : y_1\}$.

A verifiable transaction scheme consists of three main operations:

- $\mathbb{D} \leftarrow \texttt{Digest}(\mathbb{S})$. Return a digest value that is computed over the database state $\mathbb{S}$.
- $(\mathbb{S}^+, \mathbb{D}^+, \mathbb{O}_{T_i}, \pi) \leftarrow \texttt{Execute}(\mathbb{S}, T_i)$. It takes as input the current state $\mathbb{S}$ and a transaction T_i, executes T_i, and returns the updated state $\mathbb{S}^+$, a digest value $\mathbb{D}^+$ computed over the new state, an execution result $\mathbb{O}_{T_i} = \mathbb{R}_{T_i} \cup \mathbb{W}_{T_i}$, and a proof π of correctness.
- $\{0, 1\} \leftarrow \texttt{VeriTxn}(T_i, \mathbb{O}_{T_i}, \pi, \mathbb{D}, \mathbb{D}^+)$: Given a transaction T_i, the execution result $\mathbb{O}_{T_i}$, the proof π, and $\mathbb{D}$ ($\mathbb{D}^+$) that correspond to the state before (after) T_i is executed. It checks the proof and returns 1 if and only if the proof is valid.

Definition 1 (Verifiable Transaction)**.** A transaction T_i can be verified to be correct if and only if it ensures the following properties.

- **State integrity.** After executing T_i, the database state $\mathbb{S}^+$ cannot be tampered without being detected. More precisely, given a proof π corresponding to $\mathbb{S}^+$ such that `VeriTxn()` returns 1, it is infeasible to generate another proof π' and state $\mathbb{S}'$ such that `VeriTxn()` returns 1 and $\mathbb{S}^+ \neq \mathbb{S}'$.
- **Serializability.** The new state $\mathbb{S}^+$ can be verified to be identical to a union set of the old state $\mathbb{S}$ and the write set of T_i. Besides, the read set of T_i is verified to be a subset of $\mathbb{S}$.

$$\{\mathbb{D}^+ = \texttt{Digest}(\mathbb{S} \cup \mathbb{W}_{T_i})\} \wedge \{\mathbb{R}_{T_i} \in \mathbb{S}\} \tag{2}$$

Example 2. Given current database state $\mathbb{S}_0 = \{x : x_0, y : y_0\}$, and two transactions, including T_1 with an operation sequence $\langle R_1(x_0), W_1(y_1)\rangle$, and T_2 with operations $\langle R_2(y_0), W_2(x_2)\rangle$. Suppose T_1 and T_2 run concurrently, and $R_2(y_0)$ executes before T_1 commits, so $\mathbb{R}_{T_2} = \{y_0\}$. After that, T_1 commits first and makes a new state $\mathbb{S}_1 = \{x : x_0, y : y_1\}$, where $\texttt{VeriTxn}(T_1, \mathbb{O}_{T_i}, \pi_1, \mathbb{D}_0, \mathbb{D}_1) = 1$, and $\mathbb{D}_1 = \texttt{Digest}(\mathbb{S}_0 \cup \mathbb{W}_{T_1} = \{y_1\}) \wedge \mathbb{R}_{T_1} = \{x_0\} \in \mathbb{S}_0$. Hence, T_1 is said to have been verified. Then, T_2 performs $W_2(x_2)$. Finally, T_2 commits and attempts to have a new state $\mathbb{S}_2 = \{x : x_2, y : y_1\}$. However, T_2 is verified to be not serializable, because $\mathbb{R}_{T_2} = \{y_0\} \notin \mathbb{S}_1$.

3 SYSTEM OVERVIEW

In this section, we describe the threat model and then present the architecture overview of VeriTxn.

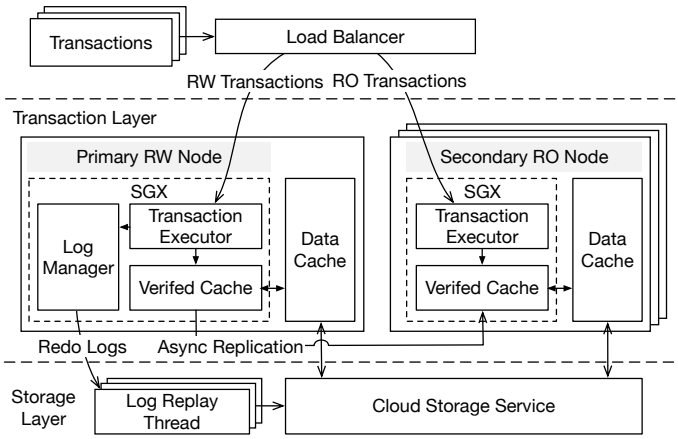


Fig. 2. System Architecture of VeriTxn

3.1 Threat Model

In a typical cloud database system, clients interact remotely for the services. All the client-side components, such as SQL clients, are trusted. This is typically realized by hosting the client in a trusted on-premises environment (e.g., behind user-controlled firewall) or in enclaves [6, 58]. In this paper, we focus on the threats to the data integrity and transaction serializability of the database. Solutions [3, 63] that provide data confidentiality, through encryption and access control for examples, may be layered on top of VeriTxn.

Servers hosting the database are typically untrusted, and consequently, the adversary may be able to have complete control over the server. We consider the strong adversary, who can cause the server to exhibit arbitrary adversarial behavior. For example, the adversary can tamper with the server's memory and disk, replicating and overwriting data pages of the host database. Based on the design principles of TEE, an adversary cannot access the enclave provided by SGX. In particular, data and computation inside an enclave are protected and attested to be correct. We require transactions to be completely processed in the enclave, achieved by the SGX attestation. Like many other SGX-based works [58, 63, 78], we exclude side-channel attacks from our scope. We note that several studies attempt to counter side-channel attacks by eliminating record-level leakage [26, 29, 35, 52, 77]. However, these methods incur significant overhead due to the need to obscure specific read/write operations in their custom software design. In our approach, we mitigate the issue of side-channel attacks by preventing record-level access pattern leakage, which is achieved by loading data at the page level, thereby hiding the precise records accessed by transactions. Although the page-level side-channel attacks remain, we consider these attacks as implementation-specific and we can opt for more secure TEEs or integrate the aforementioned methods if necessary.

3.2 Architecture

VeriTxn is designed as a disaggregated database, with the overview illustrated in Figure 2. In its disaggregated architecture, VeriTxn consists of two layers: the transaction layer and the storage layer. The transaction layer consists of multiple compute nodes, with each executing incoming transactions and returning results to users. The storage layer is a shared storage that can be accessed by every compute node. All the read-write (RW) transactions (transactions with read and write operations) are sent to the primary RW node. In contrast, the secondary RO nodes handle read-only (RO) transactions. Following modern cloud-native databases [17, 68], there is only one RW node, but a number of RO nodes in VeriTxn.

Transactions sent by clients are first forwarded through the **load balancer** to the RW node or RO nodes. Note that the load balancer is running in an enclave. In this manner, connections between the clients and the load balancer, and between the load balancer and RW/RO nodes, are protected by the remote attestation provided by SGX. Either the RW node or RO node is equipped with SGX, consisting of a trusted enclave and an untrusted data cache. The untrusted **data cache** is used for fast data access. It interacts with the storage layer to temporally cache hot data on purpose. We place the following three components in the enclave: The **transaction executor** is responsible for handling transactions' operations, including read $R_i(x_j)$, write $W_i(x_i)$, commit C_i or abort A_i, etc. Each read-write transaction generates redo logs before the commit. The **log manager** transfers redo logs to the storage layer for persistency, which is only equipped in the RW node. The **verified cache** is a pre-allocated EPC memory to cache data in the enclave. For verification purposes, it organizes the data into pages and assigns each page a verified hash.

With the large memory capacity of the latest SGX, we execute transactions entirely in the enclave through the collaboration of these components. However, achieving efficient and verifiable transaction processing is challenging due to the necessary interaction between SGX and the untrusted host. We address this challenge by introducing an SGX-friendly transaction processing protocol to minimize the intersections between the transaction executor and other components. We now discuss how our protocol efficiently executes transactions while ensuring the verifiability of both serializability and data integrity.

For a read-write transaction, the transaction executor on the RW node receives the transaction and follows the paradigm in the single-node databases to execute transactions. Specifically, the transaction executor performs the read/write operations and employs a concurrency control algorithm to ensure serializability. In VeriTxn, instead of decomposing a transaction T_i into sub-transactions to read/write data items from remote storage nodes, T_i directly reads/writes data items from the local verified cache. By so doing, we eliminate the costly distributed transactions [40, 48, 66]. If the required data does not reside in the verified cache, we load it from the untrusted data cache with a safe loading technique. During each data loading, we load and verify a data page containing the required data, rather than loading and verifying each individual data item. To verify integrity, we calculate the page hash and compare it with the corresponding hash in the verified cache. This verification, conducted at the page level, is coarse-grained and serves to lessen the communication overhead between SGX and the untrusted host. VeriTxn makes no assumptions about the concurrency control algorithm. Common-used algorithms, such as OCC and 2PL, can be applied to VeriTxn. For illustration purposes, we employ a customized OCC by default, which is naturally combined with our proposed verification mechanism. By using OCC, any two conflict transactions would be detected, and one must abort for serializability. We will elaborate on the transaction processing with verification in Section 4.2.

We ensure that read-only transactions always retrieve consistent data with verifiable integrity, by applying the proposed verification mechanism to RO nodes. To maintain the verified hash on RO nodes, we introduce an asynchronous replication mechanism, which periodically transfers the modifications of the verified cache on the RW node to RO nodes. Furthermore, we propose a consistent reading technique to perform read-only transactions on a consistent snapshot, while making a trade-off between data freshness and transaction latency. We will discuss the read-only transaction optimization in detail in Section 4.4.

4 THE DESIGN OF VeriTxn

In this section, we present the design of VeriTxn, including its key data structure called the verified cache, verification mechanism, and transaction processing protocol. We also introduce how VeriTxn handles storage disaggregation and read-only transactions.

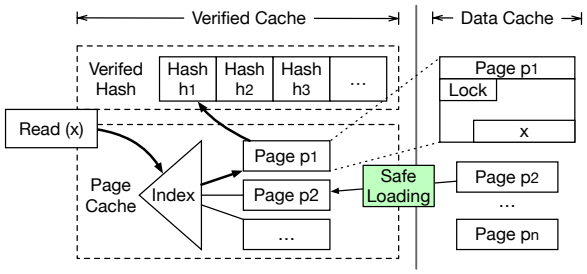


Fig. 3. Verified Cache Structure

4.1 Verified Cache

To facilitate verifiable transaction processing, we maintain a verified cache in the SGX enclave of each node. The structure of the verified cache is shown in Figure 3, which consists of two parts: 1) verified hash, including a set of hash corresponding with each data page; 2) page cache, containing a number of verified data pages. The former is used to do verification, and the latter is responsible for interacting with transactions' operations.

In VeriTxn, we organize data into pages and index them with in-memory indexes. Our system supports both hash indexes and B$^+$-tree indexes. These indexing schemes can serve as either primary or secondary indexes, depending on the application requirements. The structure of a page resembles classic page design in conventional database systems [55, 57]. Each page p contains a page header, a set of metadata including the page ID ($p.pid$), the page lock ($p.lock$), the timestamp ($p.ps$), and the number of records on the page, etc. We use $p.ps$ to represent the commit timestamp of the latest transaction that writes p. The rest of the page p is a list of data items. Each data item x contains three fields: the primary key ($x.pk$), data ($x.val$), and $x.cts$, the commit timestamp of the latest transaction that writes x.

For verifiability, each page is assigned a hash, calculated by the collision-resistant hash function $\mathcal{H}$. Only the RW node is responsible for updating the pages and their verified hashes. Note that we do not assign each data item with a hash because doing this may cause high memory overhead when the database is large. In contrast, we store the hash for each page separately in the verified hash structure. Each hash item h in the verified hash comprises: 1) $h.pid$, ID of the corresponding page p; 2) the hash value $h.hash$; 3) $h.cts$, the commit timestamp of the latest transaction that updates h. By putting it in SGX, we establish a synopsis of all data pages for integrity verification. Specifically, we calculate $p.hash$ using $\mathcal{H}(p) = \sum^{\oplus}_{x \in p} PRF(x)$, where $p.hash$ is the xor sum of the keyed pseudo-random functions of all data items in the page. Given two pages p and p', $p = p'$ implies $\mathcal{H}(p) = \mathcal{H}(p')$, and $\mathcal{H}(p) = \mathcal{H}(p')$ implies $p = p'$ with high probability.

We propose a *safe-loading* technique to safely transfer pages from the data cache to the verified cache. A page in the verified cache may be offloaded to the data cache when the verified cache is full. By safe-loading, any tampering of the data that bypasses the enclave will cause inconsistent synopsis and be detected.

Example 3. Let us refer to Figure 3, and consider an operation that reads the data item x. To fetch x, it first accesses the index and locates a corresponding page p_1. If p_1 is not in the verified cache, p_1 will be loaded in if the examination of $\mathcal{H}(p_1') = h_1$ passes. Otherwise, this page might have been modified by malicious attackers.

4.2 Transaction Processing with Verification

We now detail how to run a verifiable transaction. As discussed, we run a transaction in the SGX enclave with a server-side verification mechanism. In general, a transaction can be completed

Algorithm 1: Read/Write of a transaction T_i

```
1  Function Read(T_i, key):
2      if ∃ x_i ∈ T_i.ws ∪ T_i.rs, x_i.pk = key then return x_i
3      Read the page p_j containing x_j, where x_j.pk = key
4      Read the hash item h_j, where h_j.pid = p_j.pid
5      T_i.rs ← T_i.rs ∪ ⟨h_j.hash, p_j⟩
6      return x_j
7  Function Write(T_i, x_i):
8      T_i.ws ← T_i.ws ∪ x_i
```

Algorithm 2: Verification and commit of a transaction T_i

```
1  Function Verification(T_i):
2      for x_i ∈ T_i.ws do
3          Locate p_i containing x_i
4          if p_i.lock ≠ T_i.tid and ¬ CAS(p_i.lock, 0, T_i.tid) then
5              return false
6      for ⟨h_j, p_j⟩ ∈ T_i.rs do
7          if p_j.lock ≠ T_i.tid and p_j.lock ≠ 0 then return false
8          p'_j ← Read the page p_j
9          if H(p'_j) ≠ h_j.hash then return false
10     return true
11 Function Commit(T_i):
12     T_i.cts ← assign a commit timestamp
13     Send redo log to the storage layer
14     for x_i ∈ T_i.ws do
15         Update x_i into its corresponding page p_i
16         h_i.hash ← H(p_j)
17         p_i.lock ← 0
```

through: 1) the read/write phase, where the transaction performs read/write operations by accessing the verified cache and verifying the loaded data pages; 2) the commit phase, during which the transaction update hashes and data, as well as transfer the redo log to the storage layer. VeriTxn does not rely on a specific concurrency control algorithm and can function with most algorithms such as OCC and 2PL. For illustration purposes, we introduce how we extend Silo [67], a centralized OCC variant, to exemplify processing transactions with verification in SGX. We integrate the verification phase with the validation of the read/write set for simplicity. Each transaction T_i is performed by a thread in the enclave, maintaining a read set ($T_i.rs$), a write set ($T_i.ws$), and a unique transaction ID ($T_i.tid$). Given a transaction T_i, it executes in three steps below:

- In the read/write phase, we process read/write operations of transaction T_i. Algorithm 1 shows the pseudo-code of `Read()` and `Write()` functions. `Read()` takes T_i and a search key key as the input (line 1). We directly return x_i ($x_i.pk = key$) if it is already in $T_i.ws$ or $T_i.rs$ (lines 2). Otherwise, we add the page p_j containing the data item x_j whose primary key is key, and p_j's corresponding hash h_j to $T_i.rs$ (lines 3–5). Function `Write()` takes T_i and a new data version x_i to be written as the input (line 7), and adds x_i into the write set $T_i.ws$ directly (line 8).

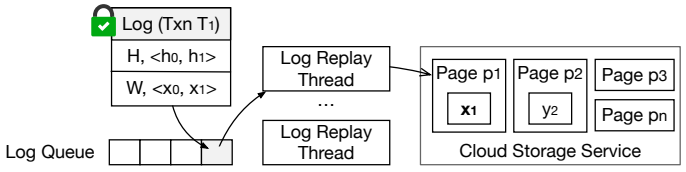


Fig. 4. Disaggregated Storage Design

- We then verify T_i by examining its read/write set in two steps. First, we lock all the pages containing x_i in the write set $T_i.ws$ (line 2). In other words, for each data item $x_i \in T_i.ws$ and $x_i \in p_i$, we set the lock $p_i.lock = T_i$ to prevent other concurrent transactions from modifying it. If the lock of p_i is held by another transaction ($p_i.lock = T_j$), T_i fails to acquire the lock and needs to abort (lines 4–5). Second, we check data pages in the read set to ensure correctness (line 6). In particular, for each data page p_j in the read set, we examine whether p_j is locked by another transaction (line 7). If it is locked, the transaction needs to abort. Besides, we check whether p_j is modified by other transactions or is tampered, through re-reading the page $p_j \in T_i.rs$ and calculating its hash (lines 8–9). If all these examinations pass, we commit T_i (line 11); otherwise, T_i needs to abort.
- After passing the verification, T_i enters the commit phase (line 11). In this phase, we first allocate a commit timestamp $T_i.cts$ for the transaction T_i (line 12). Then, redo logs are generated and sent to the storage layer (line 13). Finally, we update all the hash and data pages written by T_i, and then release all the granted locks of the pages in $T_i.ws$ (lines 14–17).

4.3 Disaggregated Storage

The storage layer of VeriTxn is constructed based on cloud storage services, such as Microsoft Azure Storage [10], Amazon S3 [8], etc. Each read-write transaction sends its encrypted redo log to the storage layer over the network, transferring data modified by it to the storage layer. For performance, we introduce the double-layer cache management to ensure that the data is cached in the verified cache as much as possible. To make the data cache in RO nodes fresh, we eagerly update the data cache by retrieving the data from the cloud storage. Due to space constraints, more details about VeriTxn's storage design, such as handling page structure modification and data vacuum, can be found in the extended version [76].

4.3.1 Redo Logging. We use redo log [54] to record the transaction's modifications. In our design, we additionally record the hash corresponding to the modified pages in the log. To ensure the security of logs, we introduce a secure redo log storage mechanism by encrypting the log with a signature-based mechanism. Specifically, each entry in the log is a pair <type, change>, where the type field is either W or H, indicating it is a data or hash log entity, respectively. The change field of a data entry stores both the old and new values, and that of the hash entry stores the old and new verified hash. Given an operation modifying the data item x from x_i to x_j, it produces a data entry $\langle W, < x_i, x_j > \rangle$, and a hash entry for the modified page p_i, denoted as $\langle H, < h_i, h_j > \rangle$. Considering the transaction T_1 in Example 1, its redo log consists of two entries $\langle W, < x_0, x_1 > \rangle$, $\langle H, < h_0, h_1 > \rangle$, as shown in Figure 4. Moreover, we assign a unique log sequence number (LSN) [54] for each log to maintain the sequential order of logs. We generate the redo log in the enclave. Each redo log is assigned a signature to ensure it cannot be tampered by attackers without being detected. This signature is a hash generated based on the log entries. After encrypting the log with a private key stored in the enclave, the log manager then sends the log to the storage layer. Each transaction ensures that its redo logs are persistent in the storage before committing. Note that the log can be decrypted with the public key in the storage layer.

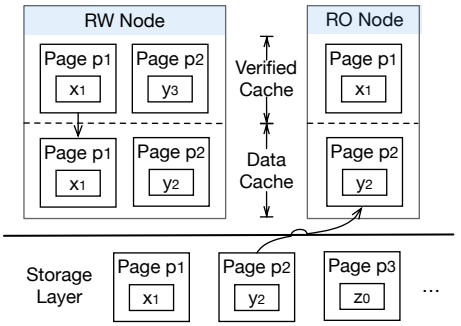


Fig. 5. Double-layer Cache Management

We employ the storage layer to handle the log replay instead of the compute node. This approach reduces network overhead as each transaction communicates with the storage layer only once. For example, as shown in Figure 4, the redo log of T_1 is generated in the RW node and sent to the storage layer. There are multiple log replay threads in the storage layer that continuously replay incoming logs to the cloud storage. Upon receiving a redo log in the storage layer, it is first stored in a log queue. The log threads fetch the queue, and write to a cloud-shared disk or call the APIs (i.e., `Put()`) provided by the cloud storage to replay logs. Note that we organize the cloud storage into pages, the same as in the caches, to simplify data transferring. By replaying redo logs in their determined serialization order indicated by LSN, we ensure the page structure is consistent with that created within the enclave during the original transaction execution. Let us refer back to Figure 4. The page $p1$ in the cloud storage is updated by the incoming log entry of T_1, where the data item x is set to the new version x_1. Further, this page p_i is asynchronously transferred to the RO nodes through gossiping or heartbeat to ensure data freshness of the RO node.

4.3.2 Double-layer Cache Management. For the best performance, since each transaction only accesses the data in the verified cache, it is therefore best to put all the data in the verified cache to avoid communicating with the cloud storage. However, this is infeasible because the database is often much larger than the memory capacity of SGX. We therefore make use of the data cache and design a hybrid caching policy to speed up instead. First, we store as much data as possible in the verified cache, i.e., only offload data from the verified cache to the data cache when the verified cache is full. Second, we eagerly refresh the data cache to ensure that the hot data can be fetched from the data cache instead of the cloud storage.

Lazy offloading. When the verified cache is full, it triggers the data offloading, which retires data pages into the data cache. The offloading is costly because it copies data from the enclave to the untrusted domain. Therefore, we utilize the commonly used least-recently-used (LRU) policy for cache replacement to avoid frequently loading and offloading the same data page. Other optimized cache replacement policies [28, 43, 62] are also applicable in this case, which are however orthogonal to this paper. Note that we do not offload data from the data cache to remote storage to reduce network overhead. Instead, we rely on the log replay to update the remote storage asynchronously.

Eager refreshing. We regard the data cache as a secondary cache, which maintains the data pages that are not stored in the verified cache. Consider a transaction T_i requires a data page p_i. T_i first searches the verified cache, then the data cache, and finally the cloud storage. In this manner, T_i can directly reach p_i if p_i is in the cache. Otherwise, T_i fetches the page p_i from the cloud storage, resulting in increased latency. To ensure the freshness of the data cache, we assign each page a lease at the time it is stored in the data cache. The lease is a timestamp allocated by the wall clock of the server. When the lease is expired, the corresponding node calls the `Get()` API to retrieve the

page from the cloud storage. We assign a larger lease if there is no update to reduce the network overhead. In this way, we use the lease to eagerly refresh the data cache with minimal overhead. We regard this page as only containing cold data if the lease exceeds a threshold. Therefore, we discard such a page from the data cache because it is seldom accessed.

We use the following example to clarify the data workflow through the double-layer cache and the cloud storage.

Example 4. In reference to Figure 5, let us consider the page p_1 and p_2. Suppose the verified cache is full, the page p_1 is offloaded to the data cache because there are no transactions that manipulate p_1. In addition, suppose the lease of the page p_2 expires in an RO node. The RO node sends a request to the cloud storage and gets the page p_2 with a fresher data item y_2. Considering the page p_2 has been modified by some concurrent transactions in the RW node, the page p_2 cached in the RO node is still not up-to-date. If there is a transaction T_2 that wants to read p_2, it may keep waiting until it gets the latest version of p_2. Alternatively, we ensure to read a consistent state instead of the latest to make a trade-off between performance and data freshness, which will be discussed in Section 4.4.

4.4 Read-Only Transactions

Many existing works [15, 23, 47] show that read-only transactions are one of the common workloads in the cloud. When there are extensive read-only transactions, a widely used solution is to deploy additional RO nodes to achieve scalability. VeriTxn extends such a design, and ensures that read-only transactions always retrieve consistent data with verifiable integrity. As discussed in Section 3.2, we first perform asynchronous replication of the verified hash to enable the proposed verification mechanism on RO nodes. We then introduce a consistent reading technique to perform read-only transactions on a consistent snapshot, while making a trade-off between data freshness and transaction latency.

Replication of the verified hash. We also use the verified hash to support verifiable transactions on the RO nodes. However, since the hash is generated in the RW node, we transfer it to RO nodes through asynchronous replication. Specifically, we combine the hash updated by multiple transactions into a batch, and then broadcast a batch to all the RO nodes. The RO node duly updates its local hash with the incoming hash. We apply the Thomas write rule [65] to skip outdated hash. That is, for each hash entry h_i, we check whether it is outdated by comparing its timestamp with the timestamp of local hash h'_i. We skip the update if $h_i.ts < h'_i.ts$ to avoid using a stale hash to update a newer hash. We perform the hash replication via a secure communication channel, established through remote attestation, as mentioned in Section 3.2. The RW node's enclave encrypts both the replication message and its hash before transmission to the RO node, thereby protecting the message from unauthorized access. Upon receipt, the RO node decrypts the message, computes the corresponding hash, and compares it with the received hash to verify integrity.

Read-only transaction processing. We propose a consistent reading technique for read-only transactions to ensure correctness and efficiency. Each transaction is assigned with a read timestamp rts. We set rts to the replication checkpoint φ, a timestamp indicating all the verified hash written before φ is already reserved in this RO node. We preform the visibility checking [13], examining rts with the timestamp on each accessed page and hash item, to ensure that read-only transactions are able to obtain data and verify its integrity from a consistent state. Moreover, since hash replication is asynchronous, this assignment allows the transaction to retrieve data from a consistent but not necessarily the latest state. By adjusting the replication batch size, we realize the trade-off between data freshness and transaction latency. Due to space constraints, the pseudo-code for the consistent reading technique can be found in the extended verision [76].

5 RECOVERY AND SECURITY ANALYSIS

In this section, we first present the recovery mechanism of VeriTxn to handle both adversarial errors and crash failures. We then conduct a thorough analysis of potential threats and demonstrate the ability of VeriTxn to effectively manage these risks.

5.1 Tampering Recovery and Fault Tolerance

We now introduce our tampering recovery technique, which ensures the availability of VeriTxn in the face of integrity attacks. As previously discussed, VeriTxn ensures the verifiability of transaction results by executing transactions within the enclave and detecting any data tampering occurring outside of SGX. However, as indicated in our threat model (Section 3.1), the cloud vendor cannot fully prevent the risk of adversarial misbehavior, and hence, the potential for data tampering always exists. To prevent the system from becoming inaccessible due to the impact of data tampering, we perform online tampering recovery for transactions that encounter tampered data, which allows them to continue executing after repairing this tampered data.

We conduct tampering recovery based on the durable secure redo log. As discussed in Section 4.3.1, we sign and encrypt the secure redo log to ensure its integrity. Following the methodology presented in [7], our basic idea is to reconstruct a correct data page by replaying all the operations on this page stored in the log. However, the log required for replay could be redundant, potentially causing lengthy replay times and blocking the system. To address this issue, we employ "anchor logs" for efficient and online recovery. We extend the recorded hash log to create an anchor log after every n modification to a page (n is a tunable parameter). The anchor log additionally captures the full content of the updated page. For example, if a transaction T_i modifies page p_i to p_i', and this is the n-th modification on p_i since the last anchor log is recorded, we register an anchor log with p_i'. These assignments allow us to quickly recover a correct page by applying subsequent operations based on the page stored in the latest anchor log.

We use the information in the verified hash to efficiently locate the required log during recovery. We introduce two additional attributes to each hash item h_i in the verified hash: 1) $h_i.anchor\_lsn$, denoting the LSN of the latest anchor log entry; 2) $h_i.current\_lsn$, representing the LSN of the most recent transaction that updates the corresponding page. When a transaction T_j loads a required page p_i into the enclave and detect it is compromised, T_j initiates a recovery process within the enclave to restore the page with the correct data through the following four steps: 1) We directly fetch the anchor log based on $h_i.anchor\_lsn$, because LSN can be converted into an exact position in the log files [54]. 2) We parse the anchor log and examine the log hash to ensure its integrity; 3) Assuming that p_i' is stored in the anchor log, we obtain subsequent modifications on p_i' by retrieving the redo log between the LSN of p_i' and $h_i.current\_lsn$; 4) We reconstruct a correct page by applying subsequent changes to p_i'. Once the recovery process is completed, we load the repaired page into the verified hash. To ensure the safety of this process, we block T_j and any concurrent transactions from accessing page p_i until the recovery finishes. If the recovery process takes too long, we abort these transactions based on the timeout to avoid deadlocks. After T_j commits, we utilize log replay to recover the page on the disk. Note that the log replay on the storage could be further tampered with. Our recovery process can be invoked again to manage this scenario. If the issue persists, we alert the user to review their security settings such as access control.

Fault Tolerance. We follow the standard mechanisms to handle RW and RO node crash failures in cloud-native databases [54, 68]. The difference in VeriTxn is that it requires additional recovery steps for verified hashes. When an RW node fails, we propagate an RO node to be the new RW node for availability. During this process, we reconstruct the verified hash based on the secure log. In case of an RO node failure, we synchronize the verified hash from another RO node, and

Table 1. Security Analysis of VeriTxn against Various Threats

Category	Specific Threat	Strategy	Case ID
Attacks on storage	Data tampering	Online verification and recovery	1a
	Log tampering	Encryption & signature Duplication	1b
Attacks on the network	Data loading/flushing attack	Online verification and recovery	2a
	Log flushing attack	Encryption & signature	2b
	Transaction request attack	Secure channel	2c
	Hash replication attack		2d
Attack on RW/RO nodes	Data cache tampering	Online verification and recovery	3

therefore, RO nodes can recover independently without blocking the RW node. Note that all the recovery process is performed within the enclave. Due to space constraints, we leave the detailed crash recovery process in the extended version [76].

5.2 Security Analysis

As presented in Section 3.1, our threat model primarily concerns data integrity and transaction serializability. Since the transaction processing component is entirely protected by the enclave for serializability, we focus on comprehensively analyzing the guarantee of VeriTxn on data integrity. Given the system architecture of VeriTxn, we identify three primary types of integrity threats: attacks on storage, attacks on the network, and attacks on RW/RO nodes. As shown in Table 1, we list all potential threats to VeriTxn based on these three categories. We now analyzing these cases in detail to sketch the proof that VeriTxn provides the verifiability of integrity:

- We first consider data tampering (case 1a) and data cache tampering (case 3), where an attacker directly alters data on the storage or memory outside the enclave. If such attacks occur, VeriTxn detects the tampering when the compromised page is loaded into the enclave, as outlined in Section 4.2. Subsequently, we initiate an online recovery to restore the tampered pages with the correct data, following the technique described in Section 5.1. Thus, we ensure that VeriTxn can ensure integrity and stay accessible during these threat occurrences. The same methodology applies to handle data loading attacks (case 2a), where an adversary disrupts the data loading process to compromise data pages.
- We next analyze the log tampering (case 1b) and log flushing attack (case 2b), where an attacker attempts to directly alter logs on the storage or modify logs during their transfer to the storage. As presented in Section 4.3.1, the secure redo log is signed and encrypted, making the adversary cannot falsify and tamper with the log. However, given the adversary's full control over the server, arbitrary adversarial errors could occur, such as deleting all data and log files. In our implementation, we mitigate this problem by maintaining multiple backups of the log. During tampering recovery, we rely on the backups to obtain the correct log. More extreme cases, like all backups being invalid, can be handled by implementing replication strategies focusing on Byzantine fault tolerance [18], which is orthogonal to our paper.
- We finally address potential attacks on transaction requests (case 2c) and hash replication (case 2d). As indicated in Section 3.2 and Section 4.4, we establish a secure communication channel based on SGX remote attestation to protect interactions between SGXs. Each message, whether a transaction request or hash replication, is assigned a unique hash and encrypted. Upon receiving

Table 2. Comparison of VeriTxn with Existing Systems

	Key Designs	Integrity	Serializability	Verification
Enclage [63]	Encryption	✗	✗	✗
VeriDB [78]	Consistent Memory	✓	N/A	Offline
LedgerDB [70]	Merkle Tree	✓	✓ (by auditing)	Offline
FastVer [6]	Hybrid	✓	N/A	Offline
Litmus [69]	Authenticated Dictionary	✓	✓	Online (in batches)
VeriTxn	Verified Cache	✓	✓	Online

a message, the enclave within the RW/RO node examines its hash to verify integrity. If tampering is detected, the receiving node issues an error and requests the sender to resend the message.

We then discuss the security of VeriTxn and compare it against that of five state-of-the-art systems on four aspects, as shown in Table 2. Enclage [63] is an encrypted storage engine designed to support data confidentiality, which however cannot ensure integrity and serializability. Confidentiality is outside the scope of VeriTxn and the other four systems. VeriDB [78] verifies the data integrity based on the deferred memory verification [7], without explicitly considering serializability verification [78]. Merkle-tree-based approaches, such as LedgerDB [70], in contrast, achieve integrity through Merkle tree verification. Additionally, serializability verification can be attained by auditing the database ledger. However, maintaining such a ledger for tracing the transaction history is costly. FastVer [6] combines Merkle tree and deferred memory verification for performance optimization. However, as a consequence, FastVer is unable to achieve online verification. Litmus [69] is a state-of-the-art verifiable database using a software-based method to ensure verifiable transactions. Litmus runs and verifies transactions in batches based on deterministic reservation [66], ensuring enhanced security by supporting online verification. However, running transactions in batches may result in higher transaction latency. In contrast, VeriTxn does not require transactions to be executed in batches. By employing the proposed verifiable transaction processing mechanism, VeriTxn achieves comparable security with Litmus, and ensures online verification of integrity and serializability. Due to space constraints, the formal correctness proof of VeriTxn is provided in our extended version [76].

Discussion on confidentiality. As outlined in Table 1, we prevent unauthorized access to logs (case 1b, 2b), transaction requests (case 2c), and hashes (case 2d) as part of our efforts to ensure data integrity and transaction serializability. While data confidentiality is not our primary concern, we do not explicitly consider data encryption. We would like to clarify that if data encryption is employed before data is transferred out of the SGX enclave, or if we utilize encrypted data storage, VeriTxn could provide additional security against unauthorized access, thereby preserving confidentiality.

6 IMPLEMENTATION

In this section, we present the implementation details of VeriTxn, and elaborate on how to extend VeriTxn into MySQL.

6.1 Implementing VeriTxn

We implement VeriTxn based on the codebase of DBx1000 [72]. Our system is written in C++, and is publicly available at [37].

In our implementation, each compute node can either serve as an RW node or an RO node. We first employ the verified cache and use indexes to organize data in the enclave of each node. We

implement both hash indexes and B$^+$-tree indexes, serving as either primary or secondary indexes based on the application requirements. In the verified cache, we use a vector to store the verified hashes, each linked to its corresponding page. We run multiple threads of three types on each node: worker, messenger, and logger. A worker thread is responsible for executing transactions with multiple operations, and for handling read/write requests. A messenger thread sends and receives network messages such as hash replication messages. In the RW node, a logger thread handles sending redo logs to the storage layer, while in the RO node, it updates the data cache with data pages received from the storage layer. These threads are pre-allocated when the enclave starts, and are managed by a specified stack called Thread Control Structure (TCS) in SGX. During transaction execution, there are several `OCall` functions that can possibly be invoked. If the page is not in the verified cache, we use the function `safe_loading_ocall()` to load required data pages from the data cache, as discussed in Section 4.1. In addition, the function `async_hash_ocall()` is used by the RW node to broadcast updated hashes to all RO nodes. The RW node does not wait for acknowledgments from all RO nodes for asynchronous replication. Lazy offloading could be implemented as an `Ocall` function. However, the memory inside SGX, being limited in size, can be exhausted easily. Thus, we directly write the offloaded page to specific data cache positions to reduce `OCall` invocations.

For the storage layer, we implement an individual log replay component to consume redo logs produced by the RW node. It consists of several log replay threads, with each being pinned to a virtual CPU for performance. We deploy the log replay component in several virtual machines with limited CPUs to get costs under control, making it suitable for the pay-as-you-go charging policy used by cloud vendors. We organize the cloud storage into pages and create two interfaces, called `Get()` and `Put()`, to make the cloud storage services transparent to compute nodes and log replay threads. That is, VeriTxn can run on various cloud storage through this abstraction. For example, when replaying a log entry, a log replay thread first calls `Get()` to retrieve the corresponding data page. Following which, this thread applies modifications, generates an updated page, and calls `Put()` to write back.

6.2 Extending VeriTxn to MySQL

To confirm the feasibility and real-world applicability of VeriTxn, we implement our proposed verification mechanism on MySQL. We choose MySQL as a representative of full-fledged open sourced DBMS. We first implement the verified cache by extending the buffer pool of MySQL. We introduce an additional hash table into the buffer pool management component, maintaining the verified hash of every page accessed through the buffer pool, including data pages, index pages, undo pages, etc. We rely on the inherent buffer management strategy to cache data in the verified cache. Each item in the hash table contains four attributes: page_id, page hash, anchor_lsn, and current_lsn. We adjust the page cleaner to insert/update the corresponding hash item when a dirty page is flushed out to disk. We modify the function buf_read_page_low() to validate the integrity of a page during its loading into the buffer. In terms of recovery, we introduce a new redo log type for the verified hash. Each verified hash log entry contains the following information: 1) all the attribute values recorded in the corresponding verified hash; 2) page data (only included if this entry is an anchor log); 3) a hash of this log entry (signed with a private key). We record a hash log when the page is flushed out to the disk, using the MTR interface provided by MySQL. We integrate the tampering recovery technique outlined in Section 5.1 based on the modified redo log. We modify the function recv_parse_log_rec() to recover hashes from the log into the verified cache during crash recovery. We leverage the redo log encryption and redo log backup offered by MySQL to ensure the log is encrypted and duplicated appropriately.

We deploy our MySQL variant into SGX using the LibOS provided by Occlum [61], without specific optimization in the MySQL codebase for the SGX environment. Our implementation of the double-layer cache on MySQL is based on the interface provided by Occlum, which allows us to mount a tmpfs serving as the double-layer cache. This arrangement enables that when data is loaded into the enclave, it is first accessed through memory outside the enclave, and then the swapfile on the disk. We would like to clarify that the focus of our implementation on MySQL is to assess the real-world applicability of our proposed verification protocol. Therefore, we aim to provide specific performance optimizations in future work, including a complete reconstruction of the MySQL codebase for SGX and the development of a fully functional double-layer cache.

7 PERFORMANCE EVALUATION

In this section, we evaluate the performance of VeriTxn against five baselines. After introducing the experimental setup, we evaluate VeriTxn in a range of settings to show its throughput.

7.1 Experiment Setup

We run the experiments in a cluster of up to 8 nodes running Ubuntu 20.04 on Microsoft Azure. Each node is a standard DC16s v3 server, equipped with an Intel(R) Platinum 8370C CPU at 2.8GHz (2×8 cores), with 128GB of DRAM. The EPC size is limited to 64GB.

7.1.1 Baselines. We compare VeriTxn with five baselines, including a cloud-native database baseline with disaggregation, a ledger database, two SGX-based storage engines, and a verifiable database.
VeriTxn w/o verification (VeriTxn (w/o)): A cloud-native database baseline excluding the verifiable ability of VeriTxn. VeriTxn (w/o) omits the use of SGX and removes all the verification logic.
LedgerDB [70]: A state-of-the-art cloud ledger database that guarantees data integrity by maintaining a database ledger using the Merkle tree. We use the open sourced implementation [30] of LedgerDB, denoted as **LedgerDB***, to conduct our experiments.
Enclage [63]: A state-of-the-art storage engine designed to support confidentiality. Since we cannot obtain the source code of Enclage, for a fair comparison, we integrate its buffer structure into our prototype system. Moreover, as our threat model emphasizes integrity and serializability, we implement an optimized Merkle tree, as discussed in Enclage's paper [63], to ensure integrity. We refer to our implementation of Enclage as **Enclage+**.
FastVer [6]: A state-of-the-art storage engine supports integrity through hybrid verification, which combines deferred verification with the Merkle tree, and employs caching to enhance performance. Since FastVer is not open sourced, we implement its hybrid verification mechanism into our codebase to exclusively compare its verification performance with that of VeriTxn. We configure the deferred verification to be executed every 1 second.
Litmus [69]: A state-of-the-art verifiable database using a software-based method to ensure verifiable transactions. We use the open-source implementation [36] of Litmus to conduct our experiments.

To facilitate a systematic and fair comparison purely on transaction processing performance, all the compared systems exclude the SQL layer (e.g., cursor) because it is orthogonal to transaction processing. Since the concurrency control algorithm of VeriTxn is extended from Silo [67], we configure VeriTxn (w/o), LedgerDB*, Enclage+, and FastVer to use the traditional Silo algorithm for a fair comparison. Following Litmus's paper [69], we set Litmus to use deterministic reservation [66] for concurrency control to achieve its best performance. Except for Litmus, all these systems support range queries with the use of B^+-tree indexes. For the fact that range-based queries may potentially introduce phantom reads and violate serializability, we therefore adopt the approach originally

proposed in Silo [67], which records the involved leaf node of the B$^+$-tree index into the read set for serializability validation. There are other alternative solutions to this problem, such as next-key locking [53], range-locks [46], and Bw-Tree [45]. However, for the better focus of the paper and also due to page length limit, we do not address orthogonal issues such as index level locking [46, 53] and index optimization [45].

7.1.2 Workloads & Default Configuration. We evaluate VeriTxn and the baselines using the following two widely known workloads.

YCSB [22, 33] generates synthetic workloads targeting large-scale Internet applications. Following existing works [72, 73], we run workloads based on a table with 10 million (10M) rows, with each row consuming 1KB. In total, the table hosts 10GB data if not stated otherwise. A parameter called skew_factor is used to control the distribution of the accessed data items. A high skew_factor value, for example, skew_factor=0.9, results in a high contention workload. By default, the `skew_factor` is set to 0.5. A default transaction comprises 16 operations, with 8 reads and 8 writes, resulting in an overall write ratio of 50%. For RO nodes, we execute read-only transactions containing 16 reads.

TPCC [25] generates OLTP workloads that model those of a warehouse order processing application. The workloads contain 9 relations per warehouse, with each warehouse being 350 MB in size. By default, we use 16 warehouses. Consistent with [20, 39], our TPCC implementation includes all the operational logics in the standard TPCC transactions. We use the standard TPCC by default, which consists of 45% of NewOrder, 43% of Payment, and 4% for each of the remaining three types of transactions. To compare with Litmus, we use a simple mix of 50% of NewOrder and 50% of Payment transactions following Litmus's paper [69].

We set the default verified cache size and data cache size to 4GB, and page size to 4KB. We set the default batch size for hash replication to 16KB. We deploy the storage layer based on Microsoft Azure Cloud Disk Storage [11]. Unless otherwise specified, by default, VeriTxn uses the B$^+$-tree index for both primary and secondary indexes to ensure a fair comparison.

7.2 Overall Performance of VeriTxn

In this set of experiments, we evaluate the overall performance and the cost of ensuring verifiable transactions by comparing VeriTxn with VeriTxn (w/o) and LedgerDB*. By default, we deploy the evaluated systems on 4 nodes (1 RW node + 3 RO nodes). Each node has 4 worker threads, 1 messenger thread, and 1 logger thread.

7.2.1 Experiments on the YCSB and TPCC Workloads, with Varying Thread Counts. We first measure the throughput with increasing thread counts. The results, shown in Figure 6a and Figure 6b, indicate that VeriTxn achieves up to 7.93× higher throughput than LedgerDB*, and incurs up to 68.67% throughput degradation compared with VeriTxn (w/o). It is widely acknowledged that there is a concurrency bottleneck on the root hash of the Merkle tree [6, 63]. Compared to LedgerDB*, which employs a Merkle tree with high verification costs, VeriTxn eliminates this bottleneck and enables transactions without conflict to be verified in parallel. We also observe that the performance degradation against the VeriTxn (w/o) increases when more threads are invoked. This is logical because both VeriTxn require additional resources to perform verification and thus affect the peak throughput when the resource is exhausted. As the conflicts between threads intensify (16 threads), thread coordination overhead such as context switching, dominates system performance, causing a throughput decline in both VeriTxn and VeriTxn (w/o), and consequently, the gap narrows. This statement is also mentioned in various works [41, 72].

7.2.2 Experiments on the YCSB Workload, with Varying the Number of Nodes. We now study the scalability of VeriTxn by increasing the number of nodes, and plot the throughput in Figure 6c.

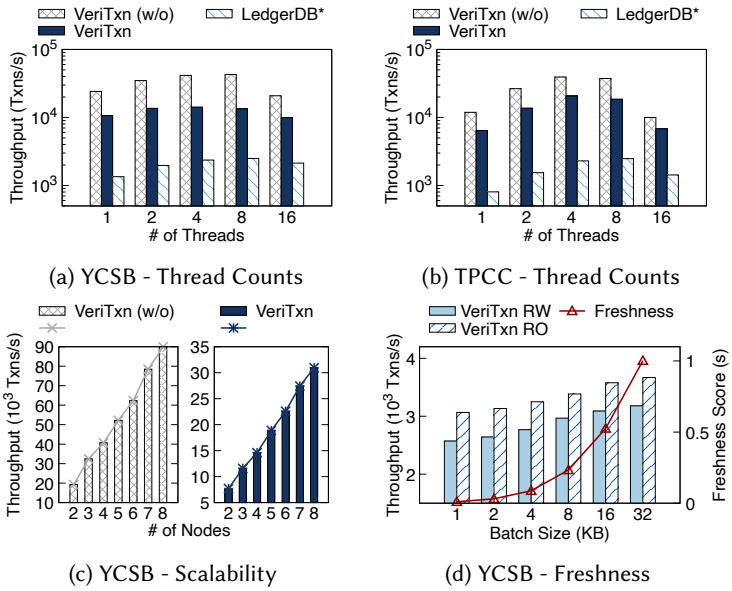


Fig. 6. Overall Performance of VeriTxn - The throughput is measured using YCSB and standard TPCC workloads.

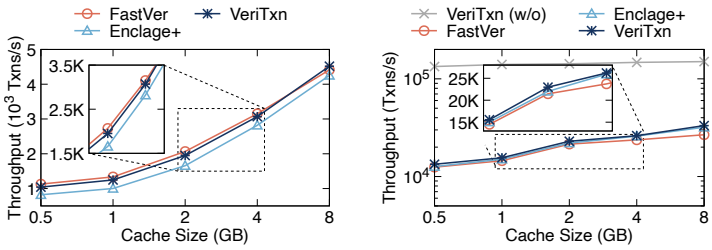
We can observe that VeriTxn achieves linear scalability, with the throughput of VeriTxn (w/o) and VeriTxn increasing by 3.67× and 3.02×, respectively, as the node count rises from 2 to 8. Since VeriTxn is designed based on the disaggregation architecture, adding additional RO nodes increases the number of read-only transactions that can be handled. Furthermore, the results demonstrate that the proposed verification mechanism does not cause a bottleneck to scalability.

7.2.3 Experiments on the YCSB Workload, with Varying the Replication Batch Size. We evaluate the data freshness on RO nodes by varying the frequency of verified hash replication. We measure the data freshness using the *freshness score*, as defined in [51]. The smaller the score, the fresher the system is. The results in Figure 6d illustrate a trade-off between performance and the freshness scores. A higher replication frequency ensures that read-only transactions obtain fresher data, as their read timestamps are more likely to be updated. On the contrary, a higher frequency also increases replication overhead, and blocks transactions until the required versions are replayed in the storage. Consequently, the throughput of both RW and RO nodes increases as the replication frequency decreases.

7.3 Analysis on the Verified Cache

We then evaluate the performance of the verified cache in VeriTxn by comparing it against the two related mechanisms used in Enclage and FastVer, respectively. In these experiments, we deploy a single RW node with 4 worker threads to focus solely on evaluating the cache and verification performance by analyzing the throughput across various cache sizes.

7.3.1 Experiments on the YCSB Workload, with Varying Cache Sizes, and 16 Data Item Access per Transaction. To explore the performance of different verification mechanisms from a transaction perspective, we first use the default setting of 16 data items accessed per transaction to conduct the experiments. As shown in Figure 7a, the throughput of VeriTxn is at most 8.27% lower than FastVer and up to 18.32% better than Enclage+. As cache size increases, the performance of all methods

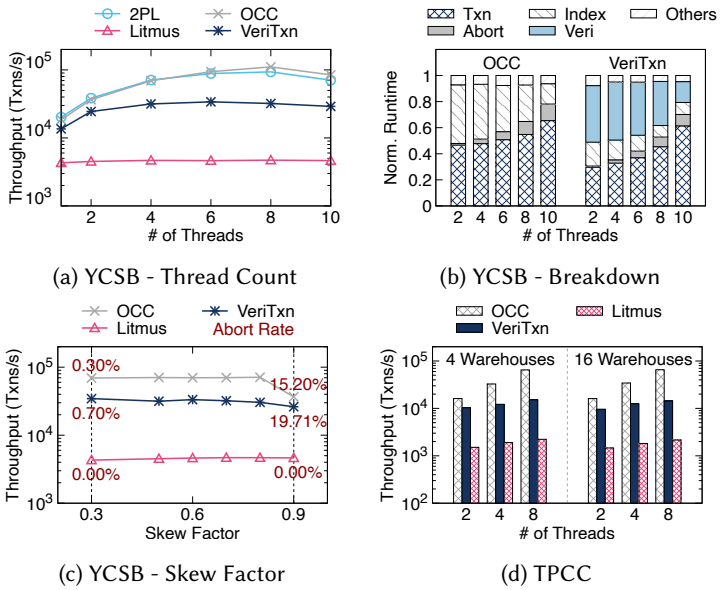


(a) Cache Size - 16 recs/txn (b) Cache Size - 1 rec/txn

Fig. 7. Verified Cache - Performance comparisons on YCSB with 10M data items as the verified cache size changes.

converges, due to the utilization of caching mechanisms by each approach. The data residing in the enclave's memory does not require integrity verification, thus reducing the verification overhead. Considering that a smaller cache size results in higher frequencies of cache replacement and integrity verification, we mainly compare the verification overhead of these methods under small cache size conditions. Enclave+ relies on a Merkle tree for verification when loading data into the cache, which suffers from root hash contention as discussed earlier. Compared to Enclage+, VeriTxn delivers higher throughput due to our efficient page-level verification, the overhead of which is bounded by the number of accessed pages per transaction. FastVer performs better than VeriTxn by incorporating deferred memory verification [7], which conducts verification in batches to reduce the verification overhead of each transaction. However, unlike VeriTxn, FastVer cannot support online verification and may not adapt well to the disaggregated architecture. FastVer requires an entire write set of each epoch to perform verification. As the write set is exclusively generated on the RW node, FastVer could be unsuitable for performing verification on RO nodes.

7.3.2 Experiments on the YCSB workload, with Varying Cache Sizes, and 1 Data Item Access per Transaction. Considering that Enclage and FastVer are originally designed as non-transactional storage systems, we then configure each transaction to access only one data item and rerun the previous experiments. The results reported in Figure 7b demonstrate that, VeriTxn outperforms Enclage+ and FastVer by up to 5.54% and 24.04%, respectively. Furthermore, when comparing with the performance differential observed with 16 data items accessed per transaction, the performance gap between VeriTxn and Enclage+ narrows, while the gap between VeriTxn and FastVer widens. The reason is that the overhead of the Merkle-tree-based verification in Enclage+ and the page-level verification in VeriTxn is affected by the number of data items accessed per transaction. In contrast, FastVer is not affected by the number of data items accessed per transaction due to the use of deferred verification. Under the current settings, each transaction only accesses 1 data item, leading to lighter verification overhead for VeriTxn and Enclage+. However, FastVer requires a complete table scan for each verification epoch, an overhead that can be non-negligible when each transaction is small. We also provide the single-node performance of VeriTxn (w/o) in Figure 7b for reference. Note that the throughput numbers of FastVer and Enclage reported in [6, 63] are higher due to the fundamental differences in the codebase. The original FastVer and Enclage implementations are based on purely key-value stores, such as Faster [19]. In comparison, our codebase, VeriTxn (w/o), is designed for evaluating transaction performance, and includes additional codes to support transactions. We have confirmed that the performance trends of FastVer and Enclage, and the throughput degradation caused by implementing their verification mechanisms, are consistent with those reported in [6, 63].



(a) YCSB - Thread Count (b) YCSB - Breakdown

(c) YCSB - Skew Factor (d) TPCC

Fig. 8. VeriTxn vs Litmus - The throughput and latency breakdown are measured on a single node with 4 worker threads. Both systems store data in memory.

7.4 VeriTxn vs Litmus

We then compare the performance of VeriTxn with Litmus. As Litmus is implemented as an in-memory single-node database, we deploy VeriTxn on one node, and store all the data in the verified cache. To ensure a fair comparison, we use the same hash index for VeriTxn as used in Litmus. We set the verification batch size for Litmus to 10^4, ensuring the performance of Litmus in our paper is consistent with that presented in [69]. We also use two baselines without verification and SGX to serve as performance upper bounds for Litmus and VeriTxn, respectively. The first baseline uses 2PL, similar to the one in [69]; and the second uses traditional Silo, denoted as OCC.

7.4.1 Experiments on the YCSB Workload, with Varying Thread Numbers. We measure the throughput with increasing thread counts from 1 to 8, and plot the throughput in Figure 8a. We can observe that VeriTxn achieves up to 6.32× performance gain over Litmus. VeriTxn's higher throughput is due to the proposed verification mechanism, which ensures efficient server-side verification. In contrast, Litmus is based on an encryption-based verification framework, incurring high overhead both on the client and server sides to perform the verification. As shown in Figure 8a, when the thread number is set to 1, VeriTxn can verify transactions at the cost penalty of 26.69% performance drop, while Litmus suffers 78.75% performance drop. It further indicates that the verification overhead of VeriTxn is lower than that of Litmus. All methods cannot scale well with more threads because of the increased contention among the transactions [72]. The time breakdown in Figure 8b indicates that with a higher thread number, transactions in VeriTxn spend a larger amount of time on concurrency control.

7.4.2 Experiments on the YCSB Workload, with Varying Contentions. We next investigate the impact of contentions on the performance of VeriTxn, by varying the `skew_factor`. The results in Figure 8c show that VeriTxn has up to 7.03× higher throughput than Litmus. When `skew_factor` does not reach 0.8, the verification dominates the cost, and thus VeriTxn performs better with 57.28% verification cost. Under high contention, transactions are more likely to abort due to concurrency

control, which is confirmed by the abort rates shown in Figure 8c. Litmus is not affected by the contentions, mainly because it processes transactions in a deterministic manner. However, deterministic transactions impose some other limitations, such as stored procedures requirement, reduced processing flexibility, etc [59]. Overall, VeriTxn still achieves 5.66× speedup than Litmus under extremely high contention (`skew_factor`=0.9).

7.4.3 Experiments with the TPCC Workload. We now evaluate the throughput with a different number of threads and warehouses. The results in Figure 8d show that VeriTxn achieves up to 5.88× higher throughput than Litmus. The performance of VeriTxn gains from its verification mechanism, which is less costly than the cryptographic-based approach used in Litmus.

7.5 Evaluation on Extended MySQL

In this section, we study the scalability performance of our proposal on a popular database system such as MySQL. We implement a MySQL variant, MySQL+VeriTxn, incorporating the verification mechanism of VeriTxn into MySQL v8.0.31, as discussed in Section 6.2. We host both MySQL and MySQL+VeriTxn in the enclave using the LibOS provided by Occlum v0.29.4 [61]. Unless otherwise specified, we set the buffer pool size of MySQL and MySQL+VeriTxn to 16GB, and use 64 client threads to obtain peak performance. By default, we set the anchor interval to 200, indicating that an anchor log for a page is generated after every 200 modifications on this page. We configure the isolation level of MySQL and MySQL+VeriTxn to serializable.

7.5.1 Experiments on the TPCC Workload. We first examine the overhead of our proposed verification mechanism by comparing the throughput of MySQL+VeriTxn with that of MySQL. In our experiments, we run the TPCC workload provided by Sysbench-TPCC v2.2 [44] on 10 tables with 50 data scales, occupying ~80GB of storage space. This setup enables us to better evaluate the verification overhead of our proposal under realistic conditions, where the data volume surpasses the cache capacity. We evaluate the throughput with increasing client thread counts from 16 to 128, and plot the throughput in Figure 9a. Comparing MySQL+VeriTxn with MySQL, we find that MySQL+VeriTxn incurs up to a 4.99% performance penalty due to the verification. We then study the throughput as we increase the buffer pool size. As shown in Figure 9a, the throughput of both systems escalates, while the performance gap between MySQL+VeriTxn and MySQL shrinks from 20.21% to 3.40% as the buffer pool size expands from 8GB to 16GB. This can be attributed to the fact that caching more data in the enclave reduces data verification overhead.

7.5.2 Experiments on the Sysbench-OLTP Workload. We then delve deeper into the verification overhead of MySQL+VeriTxn and study the effectiveness of the double-layer cache (abbreviated as DLC) by varying access contention and data size. We adopt the double-layer cache optimization into MySQL and MySQL+VeriTxn, referring to these variants as MySQL (DLC) and MySQL+VeriTxn (DLC), respectively. We set the data cache size to 16GB. Our experiments run the Sysbench-OLTP workload [44], in which every transaction executes 8 SELECT and 8 UPDATE queries. We set the skew_factor to 0.5 by default. Unless otherwise stated, we use 20 tables, each with 5M data items, consuming ~50GB of storage space. We first evaluate the throughput under various skew_factor, and plot the result in the left part of Figure 9b. As observed, the performance penalty of MySQL+VeriTxn compared to MySQL narrows from 11.76% to 8.43% as the skew_factor increases. This trend can be attributed to the fact that a higher skew_factor increases the likelihood of data being cached in the enclave, hence lowering the verification overhead. To study the effectiveness of the double-layer cache, we analyze the throughput of MySQL+VeriTxn (DLC), which shows an improvement of up to 24.85% over MySQL+VeriTxn, although this performance gain decreases as skew_factor increases. This pattern is expected as a higher skew_factor lessens the need for data loading, thereby reducing

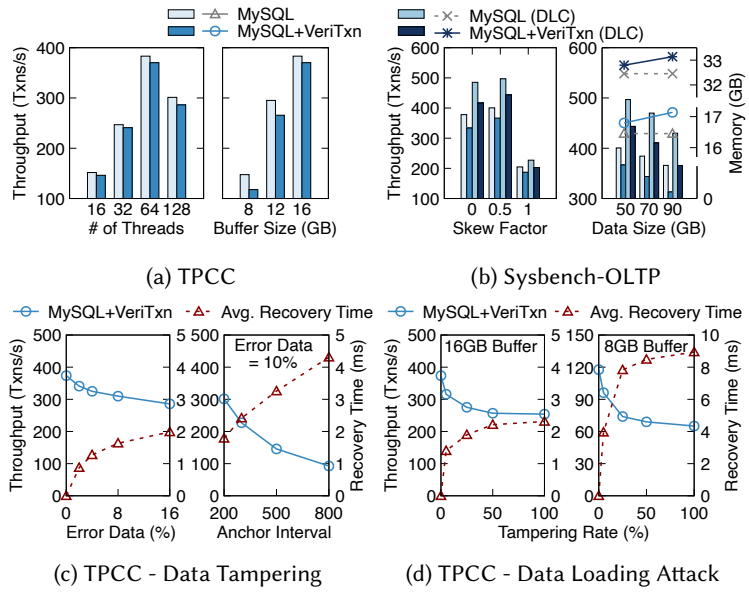


Fig. 9. VeriTxn on MySQL - Performance comparisons on the TPCC and Sysbench-OLTP workloads.

the overall benefit provided by the double-layer cache. We also observe a slight increase in the throughput of all these systems until the skew_factor reaches 0.5. Before transaction conflicts emerge as the primary bottleneck under higher contention, data loading overhead dominates performance [31].

We then investigate the throughput with different data sizes. As shown in the right part of Figure 9b, the performance degradation of MySQL+VeriTxn compared to MySQL (DLC) increases from 8.44% to 14.43% as the data size expands from 50GB to 90GB. Handling larger data sizes requires more disk loading, hence amplifying the impact of verification on throughput. The observed bounded verification overhead strengthens the claim that our proposed verification mechanism is not a bottleneck in various scenarios. In the right half of Figure 9b, we present the memory consumption of these systems, which includes the buffer pool, the verified hash (if applicable), and the data cache (if applicable). As depicted, while the memory usage of MySQL and MySQL (DLC) remains stable, MySQL+VeriTxn and MySQL+VeriTxn (DLC) incurs higher memory consumption, increasing by 3.08% and 3.24% respectively as the data size rises. This increase is due to the maintenance of verified hash. The memory usage of MySQL+VeriTxn (DLC) is higher than that of MySQL+VeriTxn, owing to the addition of the data cache.

7.5.3 Experiments on the TPCC Workload with Attack. In these experiments, we evaluate system performance and recovery time using the TPCC workload under two typical data integrity attack scenarios: data tampering (case 1a) and data loading attack (case 2a). We define the recovery time as the average time each transaction spends on both the process of tampering recovery and the blocking caused by this recovery. To record this recovery time, we embed time markers throughout the relevant code blocks.

We first conduct experiments in the face of data tampering, simulating scenarios where an attacker periodically tampers with specific portions of data with a certain frequency (every 1s in our experiments). We study the effect of varying the percentages of erroneous data, and plot the results in Figure 9c. As shown in the left part of Figure 9c, the throughput of VeriTxn decreases

by 23.49% as the percentage of tampered data rises. We attribute this performance decline to the increased chance of transactions encountering tampered pages, leading to an increase in recovery time. This observation indicates the availability of VeriTxn in the event of data tampering. To further understand the factors affecting recovery time, we also conduct experiments with varying the anchor interval. As presented in the right part of Figure 9c, the throughput of VeriTxn decreases by 69.3% with a higher anchor interval, because of the potentially larger volume of log replay. We then run experiments under the data loading attack scenario, where we adjust the tampering rate to control the proportion of tampered loading operations. Figure 9d shows that as the tampering rate increases, the throughput of VeriTxn drops by up to 32.05% when the buffer pool size is 16GB. When the buffer pool size is reduced to 8GB, the performance degradation rises to 40.5%, and the recovery time extends by 2.74×. We attribute this to the fact that a smaller buffer pool size increases the frequency of data loading, thereby magnifying the impact of blocking due to the recovery.

8 RELATED WORKS

In this section, we shall review cloud databases and verifiable databases, which are related to our work. **Cloud databases** are prevalent in recent years to provide cloud data service, i.e., database as a service (DBaaS). Recently, the new generation of cloud databases, known as cloud-native databases [4, 17, 68], emerge and seek for higher elasticity and flexibility by employing the disaggregation architecture. **Verifiable databases** are proposed to address the data integrity and security problem of outsourced data [12]. With the current trend of outsourcing data management to the cloud, the verifiability of databases becomes increasingly important. Generally, verifiable databases can be divided into two categories: *Software-based* verifiable databases are typically built with the implementation of various authenticated data structures, such as the Merkle hash tree [50], etc. *Hardware-based* verifiable databases rely on trusted hardware for design simplification and performance improvement. We have thoroughly evaluated the security and performance of several state-of-the-art verifiable databases in Section 5.2 and Section 7. VeriTxn adopts the widely-used compute-storage disaggregated architecture, and is constructed with an efficient verifiable transaction processing protocol for a real cloud environment, which is distinctly different from existing systems. Due to space constraints, a more detailed analysis of related work can be found in the extended version [76].

9 CONCLUSIONS

In this paper, we introduce VeriTxn, a novel cloud-native database that efficiently supports verifiable transactions. VeriTxn employs the trusted hardware, namely Intel SGX, to entirely handle transaction processing. By designing the verified cache in the trusted domain, VeriTxn executes verifiable transactions with bounded overhead. We propose various techniques, including double-layer cache management, and read-only transaction optimization, to exploit the benefit of the disaggregation architecture. We conducted extensive experimental evaluations using the YCSB and TPC-C workloads. The result demonstrated that VeriTxn achieves up to 7.93× higher throughput than state-of-the-art verifiable databases.

ACKNOWLEDGMENTS

We sincerely thank the reviewers and the shepherd for their valuable feedback. This research is supported by Singapore Ministry of Education Academic Research Fund Tier 3 under MOE's official grant number MOE2017-T3-1-007. Wei Lu is supported by National Natural Science Foundation of China under Grant 61972403.

REFERENCES

[1] Daniel Abadi, Anastasia Ailamaki, David G. Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter A. Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Y. Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh M. Patel, Andrew Pavlo, Raluca A. Popa, Raghu Ramakrishnan, Christopher Ré, Michael Stonebraker, and Dan Suciu. 2022. The Seattle report on database research. *Commun. ACM* 65, 8 (2022), 72–79.

[2] Peter Alvaro and Kyle Kingsbury. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280.

[3] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL Database Always Encrypted. In *SIGMOD*. ACM, 1511–1525.

[4] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *SIGMOD*. ACM, 1743–1756.

[5] Panagiotis Antonopoulos, Raghav Kaushik, Hanuma Kodavalla, Sergio Rosales Aceves, Reilly Wong, Jason Anderson, and Jakub Szymaszek. 2021. SQL Ledger: Cryptographically Verifiable Data in Azure SQL Database. In *SIGMOD*. ACM, 2437–2449.

[6] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath T. V. Setty, Nikhil Swamy, Alexander van Renen, and Min Xu. 2021. FastVer: Making Data Integrity a Commodity. In *SIGMOD*. ACM, 89–101.

[7] 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 *SIGMOD*. ACM, 251–266.

[8] Amazon AWS. 2023. *Amazon S3.* https://aws.amazon.com/s3/aws/

[9] Amazon AWS. 2023. *Cloud for State and Local Government.* https://aws.amazon.com/stateandlocal/

[10] Microsoft Azure. 2023. *Azure Blob Storage.* https://azure.microsoft.com/services/storage/

[11] Microsoft Azure. 2023. *Azure Cloud Disk.* https://azure.microsoft.com/en-us/products/storage/disks/

[12] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. 2011. Verifiable Delegation of Computation over Large Datasets. In *CRYPTO (Lecture Notes in Computer Science, Vol. 6841)*. Springer, 111–131.

[13] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*. ACM Press, 1–10.

[14] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *ATC*. USENIX, 49–60.

[16] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (2018), 1849–1862.

[17] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD*. ACM, 2477–2489.

[18] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*. USENIX, 173–186.

[19] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD*. ACM, 275–290.

[20] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2022. Plor: General Transactions with Predictable, Low Tail Latency. In *SIGMOD*. ACM, 19–33.

[21] Tencent Cloud. 2023. *Webank on Tencent Cloud.* https://www.tencentcloud.com/dynamic/news-details/100115/

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.

[23] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed

Database. In *OSDI*. USENIX, 251–264.
[24] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* (2016), 86.
[25] The Transaction Processing Council. 2023. *TPC Benchmark C*. http://www.tpc.org/tpcc/
[26] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *OSDI*. USENIX, 727–743.
[27] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. ACM, 215–226.
[28] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *VLDB*. Morgan Kaufmann, 330–341.
[29] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage. In *SOSP*. ACM, 655–671.
[30] NUS DBSystem. 2023. *LedgerDatabase Codebase*. https://github.com/nusdbsystem/LedgerDatabase/
[31] Justin A. DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (2013), 1942–1953.
[32] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *SIGMOD*. ACM, 1463–1478.
[33] Akon Dey, Alan D. Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *ICDE Workshops*. IEEE, 223–230.
[34] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. 2022. Benchmarking the Second Generation of Intel SGX Hardware. In *DaMoN*. ACM, 5:1–5:8.
[35] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (2019), 169–183.
[36] Yu Xia et al. 2023. *LitmusDB Codebase*. https://github.com/yuxiamit/LitmusDB/
[37] Zhanhao Zhao et al. 2023. *VeriTxn Codebase*. https://github.com/zhanhaozhao/VeriTxn/
[38] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.
[39] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A. Bernstein. 2022. Cornus: Atomic Commit for a Cloud DBMS with Storage Disaggregation. *Proc. VLDB Endow.* 16, 2 (2022), 379–392.
[40] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (2017), 553–564.
[41] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2022. Opportunities for optimism in contended main-memory multicore transactions. *VLDB J.* 31, 6 (2022), 1239–1261.
[42] Rohit Jain and Sunil Prabhakar. 2013. Trustworthy data from untrusted databases. In *ICDE*. IEEE, 529–540.
[43] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *ATC*. USENIX, 323–336.
[44] Alexy Kopytov. 2023. *Sysbench*. https://github.com/akopytov/sysbench/
[45] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. IEEE, 302–313.
[46] David B. Lomet. 1993. Key Range Locking Strategies for Improved Concurrency. In *VLDB*. Morgan Kaufmann, 655–664.
[47] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *OSDI*. USENIX, 333–349.
[48] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *Proc. VLDB Endow.* 12, 11 (2019), 1316–1329.
[49] MariaDB. 2023. *DBS migrates to MariaDB*. https://mariadb.com/resources/customer-stories/dbs/
[50] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO (Lecture Notes in Computer Science, Vol. 293)*. Springer, 369–378.
[51] Elena Milkai, Yannis Chronis, Kevin P. Gaffney, Zhihan Guo, Jignesh M. Patel, and Xiangyao Yu. 2022. How Good is My HTAP System?. In *SIGMOD*. ACM, 1810–1824.
[52] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *IEEE Symposium on Security and Privacy*. IEEE, 279–296.
[53] C. Mohan. 1990. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *VLDB*. Morgan Kaufmann, 392–405.
[54] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans.*

Database Syst. 17, 1 (1992), 94–162.
[55] MySQL. 2023. *MySQL Homepage.* https://www.mysql.com/
[56] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *EuroSys*. ACM, 238–253.
[57] PostgreSQL. 2023. *PostgreSQL Homepage.* https://www.postgresql.org/
[58] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *IEEE Symposium on Security and Privacy*. IEEE, 264–278.
[59] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2014. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *Proc. VLDB Endow.* 7, 10 (2014), 821–832.
[60] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE Symposium on Security and Privacy*. IEEE, 38–54.
[61] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *ASPLOS*. ACM, 955–970.
[62] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *MICRO*. ACM, 413–425.
[63] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building Enclave-Native Storage Engines for Practical Encrypted Databases. *Proc. VLDB Endow.* 14, 6 (2021), 1019–1032.
[64] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *OSDI*. USENIX, 63–80.
[65] Robert H. Thomas. 1979. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. Database Syst.* 4, 2 (1979), 180–209.
[66] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*. ACM, 1–12.
[67] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.
[68] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. ACM, 1041–1052.
[69] Yu Xia, Xiangyao Yu, Matthew Butrovich, Andrew Pavlo, and Srinivas Devadas. 2022. Litmus: Towards a Practical Database Management System with Verifiable ACID Properties and Transaction Correctness. In *SIGMOD*. ACM, 1478–1492.
[70] Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. 2020. LedgerDB: A Centralized Ledger Database for Universal Audit and Verification. *Proc. VLDB Endow.* 13, 12 (2020), 3138–3151.
[71] Yifei Yang, Matt Youill, Matthew E. Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proc. VLDB Endow.* 14, 11 (2021), 2101–2113.
[72] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (2014), 209–220.
[73] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *Proc. VLDB Endow.* 11, 10 (2018), 1289–1302.
[74] Xiangyao Yu, Matt Youill, Matthew E. Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *ICDE*. IEEE, 1802–1805.
[75] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proc. VLDB Endow.* 14, 10 (2021), 1900–1912.
[76] Zhanhao Zhao, Hexiang Pan, Gang Chen, Xiaoyong Du, Wei Lu, and Beng Chin Ooi. 2023. *VeriTxn: Verifiable Transactions for Cloud-Native Databases with Storage Disaggregation (Extended Version).* https://storage.googleapis.com/veritxn/veritxn-extended-version-sigmod24.pdf
[77] 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, 283–298.
[78] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. VeriDB: An SGX-based Verifiable Database. In *SIGMOD*. ACM, 2182–2194.

Received April 2023; revised July 2023; accepted September 2023