VeDB: A Software and Hardware Enabled Trusted Relational Database

XINYING YANG, ByteDance, China
RUIDE ZHANG, ByteDance, USA
CONG YUE, National University of Singapore, Singapore
YANG LIU, ByteDance, China
BENG CHIN OOI, National University of Singapore, Singapore
QUN GAO, ByteDance, China
YUAN ZHANG, ByteDance, China
HAO YANG, ByteDance, China

Blockchain-like ledger databases emerge in recent years as a more efficient alternative to permissioned blockchains. Conventional ledger databases mostly rely on authenticated structures such as the Merkle tree and transparency logs for supporting auditability, and hence they suffer from the performance problem. As opposed to conventional ledger DBMSes, we design VeDB – a high-performance verifiable software (Ve-S) and hardware (Ve-H) enabled DBMS with rigorous auditability for better user options and broad applications. In Ve-S, we devise a novel verifiable Shrubs array (*VSA*) with two-layer ordinals (serial numbers) which outperforms conventional Merkle tree-based models due to lower CPU and I/O cost. It enables rigorous auditability through its efficient credible timestamp range authentication method, and fine-grained data verification at the client side, which are lacking in state-of-the-art relational ledger databases. In Ve-H, we devise a non-intrusive trusted affiliation by TEE leveraging digest signing, monotonic counters, and trusted timestamps in VeDB, which supports both data notarization and lineage applications. The experimental results show that VeDB-*VSA* outperforms Merkle tree-based authenticated data structures (ADS) up to 70× and 3.7× for insertion and verification; and VeDB Ve-H data lineage verification is 8.5× faster than Ve-S.

CCS Concepts: • **Information systems → Database design and models**; *Database management system engines*; *Data management systems*; *Data provenance*; *Temporal data*; *Relational database model*; • **Security and privacy → Tamper-proof and tamper-resistant designs**; *Database and storage security*; • **Theory of computation →** *Data structures design and analysis.*

Additional Key Words and Phrases: databases, ledger databases, blockchain-like databases, database security, relational databases, authenticated data structures, trusted execution environment, blockchains

ACM Reference Format:
Xinying Yang, Ruide Zhang, Cong Yue, Yang Liu, Beng Chin Ooi, Qun Gao, Yuan Zhang, and Hao Yang. 2023. VeDB: A Software and Hardware Enabled Trusted Relational Database. *Proc. ACM Manag. Data* 1, 2, Article 194 (June 2023), 27 pages. https://doi.org/10.1145/3589774

Authors' addresses: Xinying Yang, ByteDance, China, xinying.yang@bytedance.com; Ruide Zhang, ByteDance, USA, ruide.zhang@bytedance.com; Cong Yue, National University of Singapore, Singapore, yuecong@comp.nus.edu.sg; Yang Liu, ByteDance, China, liuyang.007@bytedance.com; Beng Chin Ooi, National University of Singapore, Singapore, ooibc@comp.nus.edu.sg; Qun Gao, ByteDance, China, qun.gao@bytedance.com; Yuan Zhang, ByteDance, China, zhangyuan.42@bytedance.com; Hao Yang, ByteDance, China, yanghao.2019@bytedance.com.

2836-6573/2023/6-ART194
https://doi.org/10.1145/3589774

1 INTRODUCTION

Data management systems with tamper resistance and non-repudiation functionalities are emerging due to widespread popularity of permissionless blockchains such as Bitcoin [43] and Ethereum [15]. Transactions are hashed into digests that construct a Merkle tree [39] within a block and entangled with the previous block hash to form a verifiable chain structure. Permissioned blockchains such as Hyperledger Fabric [3], Corda [29] and Quorum [42] deliver relatively higher system performance compared to permissionless blockchains, but high workload use cases such as online transaction processing are still not well supported. Consequently, verifiable ledger database systems with a centralized architecture, have emerged as an efficient alternative [27] for applications that require immutability and auditability.

In recent years, we have witnessed the introduction of verifiable systems in the industry. Ledger database systems such as Quantum Ledger Database (QLDB) [8], LedgerDB [68], have been respectively implemented by Amazon and Alibaba. Oracle Blockchain Table [47] and Microsoft SQL Ledger [5] embed blockchain features in database management systems. Ledger databases support tamper proofs to prevent database records falsification by exploiting verifiable hash trees and link data structures. They achieve relatively high performance compared with permissioned blockchains by eliminating the consensus overhead of decentralized systems. They utilize auditable one-way peg as in Factom [55] or two-way peg protocol to overcome trusted dependency on a centralized ledger service provider.

Nonetheless, current centralized ledger databases still face two major challenges. One challenge is to design a universal platform with high performance for all the main types of verifications, i.e., data notarization, data provenance, and data transaction. Another challenge is to provide an efficient native authenticated data structures (ADS) [56, 69] for requests of ubiquitous verifications. We now discuss these four major requirements with respect to what have not been achieved in conventional ledger systems as follows.

General purpose verifiable DBMS. In general, ledger applications can be broadly grouped into three types, namely notarization, provenance, and transaction. Data notarization requires that once a data entry is written onto a ledger, any future data update must be verifiable. Data provenance (also called data lineage) involves certain entries connected by a specified key, where data integrity for all the entries' contents and order must be provable. Data transaction refers to a database transaction results in state transferring; all the states have to be verifiable. While most ledger DBMS support verifiable notarization and transaction applications, they are vulnerable in data lineage (discussed in Section 3.4).

High performance native ADS. Most ledger databases leverage hash-linked or ADS as tamper proofs. Conventional ADS mostly utilize Merkle tree-based structures which involve significant unnecessary intermediate hash computations and random I/O to store data. However, ADS are often stored in KV storage [18, 28, 48] which is not the case for general RDBMS storage engines. Hence the design choices lie in either storing to relational tables and indices which are not efficient, or reusing their original storage model but facing transaction and consistency difficulties such as ACID and recovery.

Verification with rigorous auditability. Fine-grained client-side verification is not supported in conventional ledger RDBMSes due to the encoding schema difference between the host language and database engine. Thus they verify data integrity at the server side by view mapping [5] and row audit trail [35]. Moreover, to withstand illicit ledger cover-up caused by the authorities, they post digests to independent trusted devices [5], public stores or blockchains [35] periodically. Nevertheless, such a one-way pegging paradigm is still vulnerable to the 'infinite amplification attack', which is analyzed and compared with the two-way pegging protocol [67].

Hardware-assisted ledger databases. Hardware-assisted secure database systems [4, 12, 49, 50] have been increasing in recent years. Trusted execution environment (TEE) is utilized in these databases for privacy-preserving security, but is rarely employed for auditable functionality. Blockchain systems also leverage TEE as oracles [13] to provide decentralized off-chain proofs. As a whole, TEE-assisted integrity features are not well studied in conventional ledger databases, especially efficient data lineage.

In summary, none of the existing ledger databases well serve all trusted applications requiring rigorous auditability, high performance, and fine-grained verification. In this paper, we introduce VeDB, a trusted RDBMS that not only provides high-performance trusted features but also supports ubiquitous tamper-resistant real-world applications with flexibility. We offer two alternative solutions for applications with different requirements. In particular, software-oriented verifiable feature (Ve-S), utilizing ADS to accomplish verifiable database, supports efficient data notarization and easy deployment, while hardware-enabled verifiable feature (Ve-H), realized based on TEE, ensures no vulnerable window and offers more efficient data lineage.

We devise a novel array-based ADS with two-layer ordinals (serial numbers) called verifiable Shrubs array (*VSA*). Compared with existing Merkle tree-based ADS, *VSA* has two advantages. First, it realizes append-only writes of digests, which eliminates duplicated calculations of internal nodes in the Merkle tree-based data structures as data evolve. Second, authenticated data can be simply written in and verified from a file dispensing with any additional key value or other storage used in conventional ADS. In Ve-H, we present our initiative reassembling of TEE endorsed monotonic counter, trusted timestamp, and secret-based signature, which supports tamper-evidence for all the aforementioned ledger applications as an efficient alternative. We summarize our contributions in this paper as follows:

- We present VeDB, and to the best of our knowledge, it is the first software and hardware-enabled trusted RDBMS that supports data notarization, lineage, and transactions.
- We devise a two-layer ordinal array-based authenticated structure *VSA*, with efficiently append-only authenticated data insertion, and fast ordinal proof verification.
- We present various efficient designs in Ve-S: *VSA*-based MPT for data lineage, rigorous auditable timestamp range, and fine-grained column verification at the client side.
- We design Ve-H by reassembling TEE monotonic counters, trusted timestamps, and secret-based enclave signatures to realize a practical tamper-evidence DBMS.
- We conduct experiments to evaluate VeDB insertion and verification performance. The experimental results show that Ve-S outperforms the Merkle tree-based ADS more than 70× on insertion and 3.7× on verification, and Ve-H verification is 8.5× faster than Ve-S on data lineage application.

The rest of this paper is organized as follows. Section 2 outlines VeDB architecture with data model and functionalities as a trusted database. Section 3 presents our detailed design of Ve-S, including *VSA* design, fine-grained verification, etc. Section 4 describes TEE monotonic counter and trusted timestamp-enabled Ve-H in our system. We then provide our experimental evaluation in Section 5 and related work in Section 6, before we conclude in Section 7.

2 OVERVIEW

As a hardware and software-enabled ledger DBMS, VeDB supports configurable queries and user-friendly verification requests. We discuss the threat model, data model, and main query functionality of VeDB in this section.

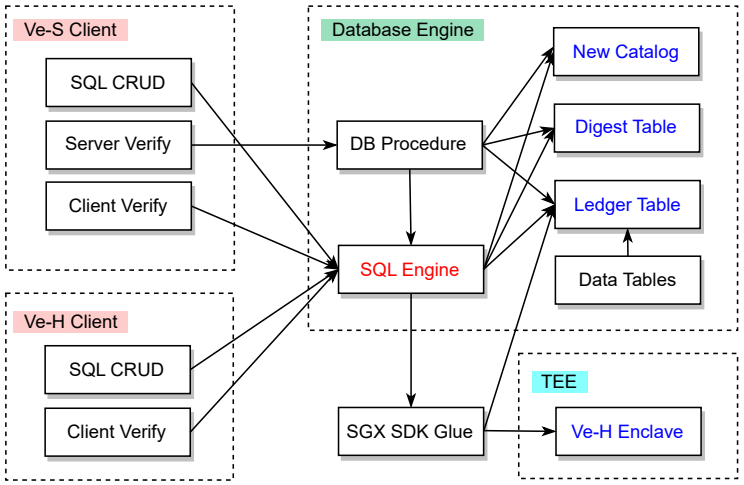


Fig. 1. System architecture of trusted features in VeDB. Ve-S utilizes software-oriented authenticated data structures to realize verifiability, while Ve-H leverages TEE to accomplish hardware-enabled trusted features.

2.1 Threat Model

VeDB is designed to prevent powerful server-side adversaries, who can gain privileged control over the operating system, database and network of the hosting machine. A strong server-side adversary is considered to have gained privileged control of the kernel which we call *server attack*. The adversaries can tamper with data through database APIs, by attaching a debugger to manipulate memory data, or modifying the storage page data directly. Adversaries can also hide intermediate records or tail records in a data provenance trail, which we call *lineage attack*. Moreover, to make a large-scale cover-up and cheat an external auditor, server and users can collude to forge data and timestamps in *colluding attack*. However, A strong adversary cannot falsify timestamps when a trusted timestamp authority (TSA) [1] is endorsed on the ledger because it is a legally accepted timestamp endorsement. Also, an adversary cannot observe or modify data within a TEE enclave. Side channel attacks [57] are orthogonal to our work and could be mitigated by [46].

2.2 Design Goal

We aim to develop an efficient and full ledger RDBMS to not only support verifiable database functions, i.e., tamper-resistant and non-repudiation, but also cover all auditable ledger features including data mutation [68]. Additionally, we also want to offer common verifiable ledger libraries for future efficient migration to other DBMSes. Hence trusted functionality in VeDB is designed to be decoupled from a DBMS engine. Another design goal worth mentioning is we want to decouple our functional implementation from a special-purpose storage system. That is, we want to be able to plug onto a host DBMS storage kernel whose distributed architecture, transaction models, availability protocols, and data reliability can be effectively leveraged.

2.3 Architecture

VeDB is a trusted relational database on ByteDance public could platform called Volcano Engine (Volcengine) [17], which includes cloud infrastructure, video and content distribution, data processing and artificial intelligence, with a total of around 100 services. The system architecture of

VeDB trusted features, i.e., Ve-S and Ve-H , is shown in Figure 1. Both Ve-S and Ve-H support verification API besides standard SQL and share the same ledger table once a data table is defined with trusted features via DDL. Data tables include current and historical tables which reference journal sequence number generated from ledger tables (detailed in Section 3.2). All data I/U/D operations executed by the SQL engine are routed to *VSA* computation (in Ve-S) or enclave procedure (in Ve-H) after row digest is calculated. Ve-S then stores authenticated metadata to digest and catalog tables, while Ve-H doesn't. Ve-S has both server-side and client-side verification for ADS proving, while Ve-H only needs client-side verification to validate the TEE signature.

2.4 Data Model

VeDB Ve-S categorizes ledger-related stores into three characteristics, i.e., ledger tables, catalog tables, and digest tables. Ledger table stores DBMS temporal table and auditable ledger operation trail, both in a row store. Digest table, together with its implicit index, is used to store digest in a verifiable Shrubs array (*VSA*, detailed in Section 3.1). Catalog tables store ledger metadata.

Ledger tables. To isolate the entire ledger operation recording from the temporal table, we design a separate ledger table to store ledger data that not only logs data modification but also operations themselves, such as mutation and time endorsement. The ledger table entails a monotonic number as the primary key called record serial number (*rsn*), which is referenced by the same named implicit column defined as the foreign key in the temporal table. Besides entering records for every temporal table data modification (we call general records), the ledger table also records data mutation and time endorsing trails (called extended records) to provide an auditable ledger (detailed in Section 3.2).

Digest tables. Both general and extended records are assigned a digest using a predefined hash function type (e.g., SHA256 [14]), and stored in the ledger table. These leaf node digests, stored in L0 of Figure 3b, are computed in the *VSA* ordinal path to construct the final digest array, which is stored in table *t_vsa* within the same schema of the ledger table. An auxiliary block range index [64] is implicitly created to accelerate sequential data page locating based on key *vsa_serial*, which is the column for *VSA* ordinal number.

Catalog tables. *ve_cert* stores user and DBMS certificates with their authorized timestamps. This is used to withstand a client-side repudiation (by verifying its signature on submitted transactions) or a server-side contradiction (by validating its signature on receipt). *ve_ledger* records the authenticated *VSA* metadata of ledger tables such as the latest *rsn*, *d_proof* (linear digest as proof) of *VSA*, etc. *ve_tl* is a time ledger containing time anchoring metadata to solve *colluding attack* (detailed in Section 3.5).

An additional implicit column called *tee_sign* is added in the ledger table of Ve-H to store the signed proof by enclave (detailed in Section 4). Digest table *t_vsa* and trusted catalog tables, i.e., *ve_cert*, *ve_ledger* are not created for Ve-H since ADS-based methods are replaced by TEE in hardware-enabled design.

2.5 Data Insertion

A full-cycle data insertion in Ve-S has two phases, i.e., client-side phase and server-side phase as depicted in Figure 2. In the client-side phase, a user serializes the data by predefined formats and computes a digest, then signs the data using a private key whose paired RSA [52] public key is stored in *ve_cert*. Regarding the server-side phase, each newly inserted row will be assigned a *rsn*. DBMS computes a row digest (*digest* field in ledger table) based on the hex value of the row in the data manager (DM) format, and calculates *d_proof*, which is then updated into *ve_ledger* together with its related *rsn*. Meanwhile, *t_vsa* and its BRIN (Block Range Index) are also updated from *VSA*. Finally, after updating all the ledger table fields such as *digest*, *trantype* (transaction type),

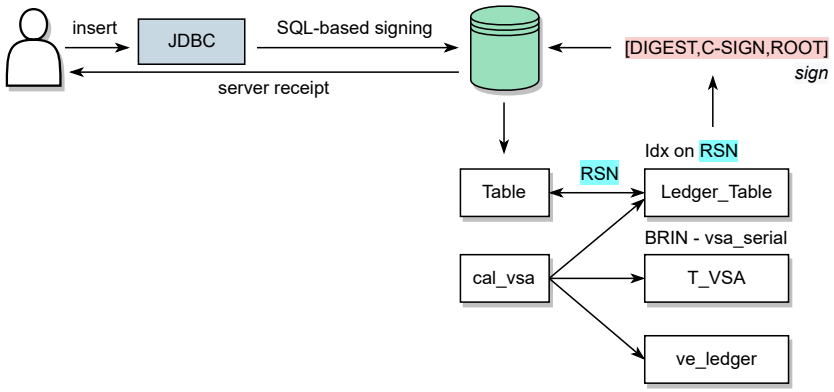


Fig. 2. Data insertion in VeDB (Ve-S). An ordinal is assigned and inserted into ledger table, together with other metadata, and referenced by data (temporal) table as a foreign key. Digest table is filled with the computed ADS data, and a server-side signed receipt would be replied to the client.

c_sign (client signature), etc, DBMS signs on the [*digest*, *c_sign*, *d_proof*] receipt using its private key, and replies the client caller.

As discussed in Section 2.4, Ve-H does not create the new digest and catalog tables in Ve-S. After computing the row digest, a new row insertion will send the digest to the enclave by triggering the enclave call, and TEE subsequently signs a proof set composed of digest, TEE endorses monotonic counter and trusted timestamp, and finally updates *tee_sign* column of ledger table with the TEE signed data as proof (detailed in Section 4).

2.6 Data Verification

Unlike most ledger DBMSes, VeDB provides not only server-side verification, but also client-side and dual-side verification. Server-side verification validates a tamper-evidence data structure based on a 'trust-but verify' premise, which means a user doesn't consider the DBMS server as compromised by an attacker who would tamper database, but indeed needs a guarantee of the data integrity. VeDB supports systematic procedures to verify data integrity on full table audit, row-level verification, and column-level validation. Regarding client-side verification, a user may be concerned with the honesty of data server and suspects tampering attacks from the server, and thus VeDB provides interfaces for users to download all the *VSA* file data to verify any digest existence via the supported client SDK. Dual-side verification is designed for what we call semi-trusted purposes, in which a user does not delegate an overall verification task to a database server that is not fully trusted, but seeks a certain verification path (e.g., simple payment verification [37]) from the server for self-validation. Ve-S dual side verification API retrieves a verifiable path of *VSA*. Users can validate data integrity by the paired *d_proof* and the fetched path.

2.7 Queries and Procedures

We implement non-verification ledger functionalities by adding syntax onto standard SQL, and support new DBMS procedures and client SDK APIs for verification. Note that general SQL for data insertion, deletion, and updates are all retained in VeDB. A ledger table is specified when creating a table by newly added syntax as shown in Table 1. `LOCAL` is the default type for Ve-S while `TEE` for Ve-H. `SHA2_256` defines the hash function. User certificates are defined and altered by adding a suffix to existing SQL `CREATE/ALTER`. Data mutation, i.e., ledger deletion and data erasing are all invoked by standard SQL `UPDATE` and `DELETE`. Ledger deletion uses table deletion by a specified

Table 1. Ledger DBMS queries (DDL, DML) and verification procedures in VeDB.

Data Operations	SQL and Procedures
User certificate	CREATE/ALTER USER name WITH CERT 'pubkey';
Table creation	CREATE TABLE T(C1 INT, C2 VARCHAR) IMMUTABLE YES WITH TYPE LOCAL USING "SHA2_256";
Ledger deletion	DELETE FROM T WHERE RSN <rsn_del;
Data erasing	UPDATE T SET col_hide = null WHERE RSN = rsn_hide; UPDATE T_ledger SET TRANTYPE = 'H' WHERE RSN = rsn_hide;
Server verify	VERIFY_TABLE(&schema_name, &table_name, &num_verified); VERIFY_ROW(&schema_name, &table_name, &rsn, &bool_verified); VERIFY_COTENT(&schema_name, &table_name, &rsn, &col, &value_verified);
Get ledger data	GET_RAW_DATA(&schema_name, &table_name, &jsn, &row_bytes, &col); GET_PROOF(&schema_name, &table_name, &rsn1, &rsn2, &root_rsn2); GET_VSA(&schema_name, &table_name, &vsa_file);

retention *rsn* predicate, and data erasing is supported by nulling the value of the column to be hidden, and flagging its TRANTYPE to *'hidden'* on the ledger table afterward.

Ledger verification related procedures in VeDB are packaged within a DBMS maintained schema called DBMS_LEDGER_TABLE. The server-side verification APIs are designed to support full table audit, row-level verification, and column-level content validation. Data fetching APIs are developed to furnish user self-verification by client SDK. Fine-grained content validation can be conducted when the data is in database row format. The proof retrieving API offers the user the *VSA* verification path for self-validation. Users can also obtain the *VSA* file for overall self-verification when server-side verification is not trusted.

2.8 Use Cases

Recall that we differentiate ledger DBMS applications into three categories in Section 1, i.e., data notarization, lineage, and transaction. From a technical perspective, a verifiable data transaction is a fine-grained data notarization that can handle state updating. We shall discuss the two main types, i.e., data notarization and lineage, with TikTok [16] use cases here.

Data Notarization. TikTok users publish and share their videos on the platform. All these videos have authorship copyright. When a video is uploaded, a hash is computed based on its content to produce a unique digital DNA that is provable [66]. The blockchain-like features enhance tamper-evidence in conventional databases to satisfy judicial evidence, which has already been accepted in some internet courts [22, 63].

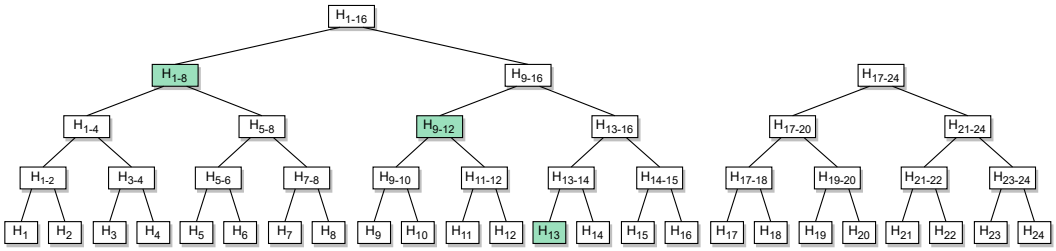
Data Lineage. A video/music's royalty can be assigned and reassigned to different holders. Therefore, an entire trace of all royalty changes has to be proved without 1) any data tampering, 2) any hidden historical data, and 3) any insertion before a prior timestamp. VeDB supports a provable data lineage to such use cases that are not supported by conventional relational databases.

3 VE-S DESIGN DETAILS

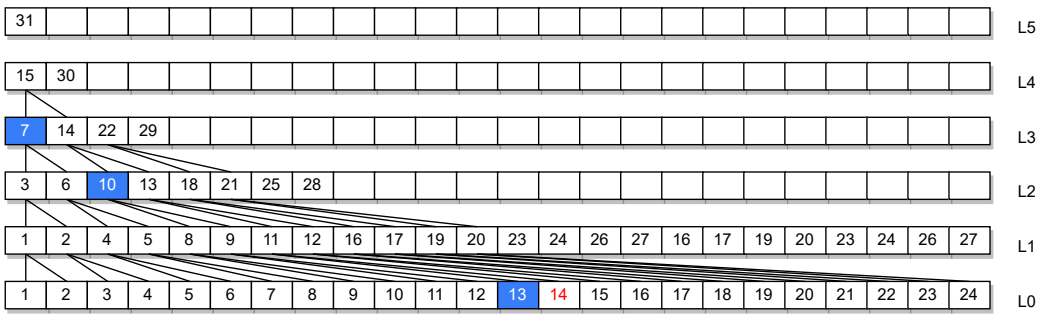
In this section, we present VeDB in detail, including *VSA* design with the implementation based on native DBMS storage, fine-grained client-side verification, rigorous ledger DBMS timestamp range in action, and DBMS native data lineage, etc.

3.1 Verifiable Shrubs Array

Merkle tree-based authenticated structures have been mostly used in ledger databases, with implementation in accumulator model [68], branching vector commitment [34], or block-chained



(a) Shrubs tree



(b) Verifiable Shrubs array ordinals

Fig. 3. Data structure of verifiable Shrubs array. The root-set when inserting the 13th leaf node composes intermediate nodes iNode7 (H_{1-8} in Shrubs tree), iNode10 (H_{9-12} in Shrubs tree), and leaf node lNode13.

structures [5]. Their insertion and verification cost can be prohibitively expensive for intermediate non-leaf node computation when the tree depth is high. To reduce the hash calculation overhead, Shrubs tree[2] is introduced with $O(1)$ insertion complexity due to amortized updates, which is defined not by the root, but by the path set to the rightmost non-empty leaf node, and then uses the root-set composed by the full sub-tree roots as tamper-proof. As shown in Figure 3a, a Shrubs tree returns a $[H_{1-8}, H_{9-12}, H_{13}]$ root-set as authenticated proof for leaf node 13 insertion, to avoid intermediate bottom-up hash computations in conventional Merkle tree. However, tree node insertion and searching based on KV storage systems still incur significant overhead caused by random I/O compared to append-only file systems. We, therefore, devise a verifiable Shrubs array (called *VSA*) to flatten conventional authenticated tree model structures into logical arrays, and treat verifiable digests as append-only stream data stored in the file model based on their Shrubs generation ordinals, i.e., the digest computing sequence from the leaf node (we call lNode) to the higher meta layers (we call iNode). Ordinals in *VSA* are divided into transaction ordinals (L0) and meta ordinals in the upper layers as depicted in Figure 3. Transaction ordinal increments based on the transaction sequence, similar to LSN (Log Sequence Number), while meta ordinal is incrementally assigned on the Shrubs computing path with a bottom-up order. Both ordinals count their own without intersecting each other.

3.1.1 VSA Insertion by Proof. A complete *VSA* insertion by proof (IP-1) contains two steps: locate the proof set for the underlying leaf node (we call proof-set determination), and the meta node filling triggered by the inserting cell (we call meta ordinal assignment).

Algorithm 1: *VSA* Data Insertion and Receipt Generation

Data: *rsn* to be inserted m_r, *VSA* latest ordinal m_o
Result: inserted meta node $\mathbb{N}$, Receipt $\mathbb{R}$, Receipt digest π

$len \leftarrow$ binary length of m_r; $i, j, k, \lambda \leftarrow 0$;
if m_r *is even* **then** ▷ proof-set search
  foreach *bit of* m_r *from high order to the next lowest* **do**
    if *TRUE* **then**
      $\lambda \leftarrow 2^{len-1-(i++)} + \lambda - 1$;
      $\mathbb{R}_{j++} \leftarrow \lambda$;
    end
  end
  $\mathbb{R}_j \leftarrow m_r$;
else
  proof-set search takes as input $m_r - 1$;
  repeat ▷ assign meta ordinal
    search m_r binary from the low order bit;
    $k \leftarrow k + 1$;
  until *TRUE*;
  $D_{m_o} \leftarrow D_{m_r}$;
  repeat
    $\mathbb{N} \Leftarrow D_{m_o++} \leftarrow cal\_digest(D_{\mathbb{R}_{j-1}}, D_{m_o})$;
    $k \leftarrow k - 1$; $j \leftarrow j - 1$;
  until *k==0*;
  $\mathbb{R}_{++j} \leftarrow m_o$;
end
$\pi \leftarrow digest\_of(\mathbb{R}_{0 \rightarrow j})$;

Proof-set Determination. A proof-set determination is separately discussed when its ordinal is even or odd. Given an odd node, the relevant proof-set concatenates its left even sibling with the sibling's proof-set. Thus, the main proof-set determination algorithm lies in the procedure for even nodes as depicted in Algorithm 1.

A destination binary encoded even number is traversed from high order bit for true bit hit. A first hit indicates a first needed meta node found for the proof path, whose ordinal is located by $2^M - 1$, where M is the digital length from the hit bit to the low order bit. The system then recursively searches for a true bit on the loop, and calculates the ordinal of the needed node by adding the previous hit bit accumulatively. For instance, as shown in Figure 3, the ordinal binary value of the leaf node `lNode13` is `1101`. Thus, the first hit of proof-set searching happens at the first high-order bit, with decimal ordinal $2^3 - 1$. The next hit occurs at the second loop of bit searching, whose decimal ordinal is accumulated as $2^3 - 1 + 2^2 - 1$. No more true value was found before the low order bit in this case. By this method, the final proof-set for `lNode13` is therefore located at `<iNode7,iNode10>`. Regarding `lNode14`, due to odd ordinal, the path is its left sibling, i.e., `lNode13`, plus `lNode13`'s proof path. Thus we get the validation path `<iNode7,iNode10,lNode13>`.

Meta Ordinal Assignment. A typical *VSA* insertion procedure consists of three major steps: meta ordinal assignment and digest computation, linear digest calculation, and *VSA* table updates, as illustrated in Algorithm 1. Firstly, meta ordinals are monotonically assigned based on Shrubs generation path. Secondly, linear digest *d_proof* is then computed on binary data by concatenating

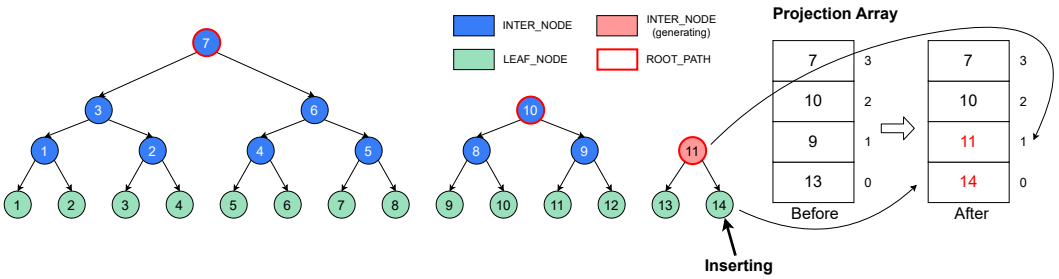


Fig. 4. VSA insertion by projection (IP2). The projection array is [13,9,10,7] after the 13th leaf node is inserted. During the 14th leaf node insertion, array would be updated to [14,11,10,7].

all Shrubs root set digests linearly from left to right. Finally, *VSA* tables, i.e., *t_vsa* and *ve_ledger*, are filled and updated with the newly generated ordinals and digest *d_proof*.

The procedure for meta digest computation consists of two major steps: verification path (i.e., proof-set) rehearsal and bottom-up traversal. Verification rehearsal entails the pre-execution of *VSA* verification taking the underlying transaction ordinal as input. Let us refer to Figure 3, suppose we insert `lNode14`, the correlated rehearsal path is `<iNode7,10,lNode13>`. Bottom-up traversal calculates meta digests of each layer with the pairwise siblings. The depth of bottom-up traversal is determined by the bits passed from the low order bit to the first located true value of the leaf binary data. For example, the binary value of `lNode14` is `1110`, since the first true bit is found on the first bit next to the low order bit. The traversal depth is only 1, which means just one round digest calculation is needed in this *VSA* insertion. The digest is computed by `lNode14` with `lNode13`, and results in the digest value of `iNode11`.

Linear digest is an efficient design to serialize *VSA* proof set into a simple digest, which saves network bandwidth and memory cost for intermediate Shrubs proof set. We offer a deferred *d_proof* computation where not each transaction root is calculated. Instead, a root is computed for a batch accumulated based on predefined constraints, either by limited transaction number or time window. Since prior proof can be avoidable when a later proof is provided in a neglected time slot, this method provides a tradeoff between performance and proof granularity.

3.1.2 VSA Insertion by Projection. Besides the set searching cost, the proof-set determination phase in IP-1 may introduce additional I/O cost for proof-set fetching if not cached. Hence we devise an initiative *VSA* insertion alternative called IP-2 (insertion by projection), which is extremely efficient leveraging the register cache.

Here projection means vertically mapping the rightmost element of each layer in Figure 3b into an array defined as $\mathbb{P}$. As described in Algorithm 2, projection array $\mathbb{P}$ stores all the latest ordinal in each logic layer. Once a new insertion with odd *rsn* is requested, the bottom-up hash calculations are first determined similar to the meta ordinal assignment phase in Algorithm 1. After that, $\mathbb{P}$ array elements are refreshed by the most current element within each logical layer consequently.

To better understand the algorithm, we take an example of $\mathbb{P}$ values as depicted in Figure 4. Let the current inserting transaction be *rsn-14*, where the underlying $\mathbb{P}$ array ordinal is [13, 9, 10, 7]. After successful insertion, the array ordinal is updated to [14, 11, 10, 7]. Note the first element (represents lNode) can be just updated for even leaf ordinal to avoid unnecessary assignment in another implementation. When `lNode15` is inserted, $\mathbb{P}$ ordinals are [15, 11, 10, 7] (just updates the first element when even ordinal). When `lNode16` arrives, the array ordinals are refreshed to [16, 12, 13, 14, 15]. IP-2 performs superior fast when using register cache to store the $\mathbb{P}$ array. The practical

Algorithm 2: *VSA* Insertion by Projection

Data: inserting *rsn* m_r, latest ordinal m_o, projection array $\mathbb{P}$
Result: inserted meta node $\mathbb{N}$, $\mathbb{P}$

$D_{\mathbb{P}_0} \leftarrow D_{m_0}; i, j \leftarrow 0;$
repeat
  search m_r (when odd) binary from the low order bit;
  $i \leftarrow i + 1;$
until *TRUE*;
$D_{m_o} \leftarrow D_{m_r}; D_{\mathbb{P}_j} \leftarrow D_{m_r-1};$
repeat
  $\mathbb{N} \Leftarrow D_{m_o++} \leftarrow cal\_digest(D_{\mathbb{P}_j}, D_{m_o});$
  $i \leftarrow i - 1; j \leftarrow j + 1;$
  $\mathbb{P}_j \leftarrow m_o; D_{\mathbb{P}_j} \leftarrow D_{m_o};$
until *i==0*;

storage cost can be limited within several KB, since the node structure is generally less than 100B, and 50 elements can support peta-level transactions.

3.1.3 VSA Verification. A *VSA* verification validates a specified *rsn* by outputting a validating proven path that takes as input a given latter *rsn* and root pair, i.e., as `GET_PROOF` described in Table 1, r_1 would be verified on $(r_2, root_2)$ pair. Given a r_2, the related proof-set is computed by Algorithm 1 for concatenated digest *d_proof* computation, which is then verified by $root_2$ if equivalent. Therefore the verification for a digest of r_1 lies in how to locate its affiliating sub-root in r_2's proof-set, then obtain the bottom-up verification path to the sub-root.

We present the path selection in Algorithm 3. After r_1 proof-set and r_2 root-set (root-set search would retrieve final root set for odd leaf nodes) are calculated, a replace node, i.e., the sub-root in r_2 proof-set which r_1 resides in, would split r_2 proof-set and be replaced by the selection path. Then $\mathbb{R}_2$ is filled consecutively between layers and compensated recursively until reaches the sub-root to be replaced based on the algorithm depicted. Moreover, the overall set would be verified in two steps separately. The replace node is firstly verified by $root_2$ after concatenating root set linearly. After that, r_1 is verified by the selected path mutually from the leaf node to the replaced node. Note each intermediate node will be compared with its peer in the same layer before hash computation to decide whether it is a left sibling or a right.

To better understand the algorithm, we present two examples here. Consider the data verification case of $m_r = 8$ and $m_o = 16$.

- Locate their root-set and proof-set by Algorithm 1.
  - m_{16} root-set $\Rightarrow$ <iNode15>.
  - m_8 proof-set $\Rightarrow$ <lNode7/8,iNode3/4>.
- $m_8 \Rightarrow$ <lNode8/7,iNode4/3> layer heights.
  - Heights $\longrightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 1 \dashrightarrow$ iNode7.
- Intermediate node expansion.
  - Skip layer fill up since m_8 proof-set is consecutive.
  - Compensated root $\Rightarrow$ iNode15, height $\longrightarrow$ 5.
  - Expand to iNode15, get right sibling iNode14 from iNode7, then append into $\mathbb{R}_2$, and finish the extension loop since reached the fist meta node of m_{16}, i.e., iNode15.
- Set replacement and finalization.
  - <<lNode8/7,iNode4/3>,iNode14>.

Algorithm 3: *VSA* Data Verification Path Selection

Data: *rsn* to be verified r_1, *rsn* of verification root-base r_2
Result: set of verification path $\mathbb{R}_{v \to r}$

$\mathbb{R}_r\{lNode_{m_{lr} \to n_{lr}}, iNode_{m_{ir} \to n_{ir}}\} \leftarrow r_2$ root-set search;
$\mathbb{R}_v\{lNode_{m_{lv} \to n_{lv}}, iNode_{m_{iv} \to n_{iv}}\} \leftarrow r_1$ proof-set search;
find the first (split) iNode in $\mathbb{R}_r \Rightarrow iNode_s \notin \mathbb{R}_v$;
$\mathbb{R}_1 \leftarrow \mathbb{R}_r\{lNode_{m_{lr} \to n_{lr}}, iNode_{m_{ir} \to s-1}\}$;
$\mathbb{R}_2 \leftarrow \mathbb{R}_v\{iNode_{s \to n_{iv}}, lNode_{m_{lv} \to n_{lv}}\}$;
$\mathbb{R}_3 \leftarrow \mathbb{R}_r\{iNode_{s+1 \to n_{ir}}\}$;
$\mathbb{H}_n \leftarrow \mathbb{R}_2$ logical tree height permutation ;
foreach $\mathbb{H}_{i \mapsto [n-2,0]}$ **do** ▷ node compensation
    $j \leftarrow \mathbb{H}_i + 1$;
    repeat
        if $\mathbb{H}_{i-1} \neq j$ **then**
            find $\mathbb{R}_{2_i}$ right sibling m_{ir} in the first loop;
            $m_{jl} \leftarrow (i \neq (n-2))?(m_{ir} + 1) :$ parent of $\mathbb{R}_{2_{i,i+1}}$;
            find m_{jl} right sibling $m_{jr} \leftarrow m_{jl} + 2^{\mathbb{H}_i} - 1$;
            $\mathbb{R}_2 \leftarrow \mathbb{R}_2 \cup \mathbf{m}_{jr}$;
            $j \leftarrow j + 1$;
        end
    until $\mathbb{H}_{i-1} \equiv j$;
end
$m_k \leftarrow (\mathbb{R}_{2_{iNode_s}} \equiv \emptyset)?$ parent of $\mathbb{R}_{2_{lNode}} : \mathbb{R}_{2_{iNode_s}}$;
repeat ▷ set replacement
    $m_k \leftarrow m_k + 2^{height(m_k)-1} - 1$;
    $\mathbb{R}_2 \leftarrow \mathbb{R}_2 \cup \mathbf{m}_k$;
    $m_k \leftarrow m_k + 1$;
until $m_k \equiv \mathbb{R}_{1_{iNode_s}}$;
$\mathbb{R}_{v \to r} \leftarrow \mathbb{R}_1 \cup \mathbb{R}_2 \cup \mathbb{R}_3$;

Consider another example of $m_r = 14$ and $m_o = 24$, which needs an intermediate node expansion during verification as depicted in Figure 5.

- Locate their root-set and proof-set by Algorithm 1.
  - m_{24} root-set $\Rightarrow$ <iNode15/22>.
  - m_{14} proof-set $\Rightarrow$ <lNode13/14,iNode7/10>.
- $m_{14} \Rightarrow$ <lNode14/13,iNode10/7> layer heights.
  - Heights $\longrightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 1 \dashrightarrow$ iNode15.
- Intermediate node expansion.
  - Compensate the missed layer 2 $\rightarrow$ iNode12, which expends the intermediate set $\Rightarrow$ <lNode14/13,iNode12/10/7>.
  - Compensated root $\Rightarrow$ iNode15, height $\longrightarrow$ 5.
  - Expand to iNode15, and finish the extension loop since reached the first meta node of m_{24}, i.e., iNode15.
- Set replacement and finalization.
  - <<lNode14/13,iNode12/10/7>,iNode22>.

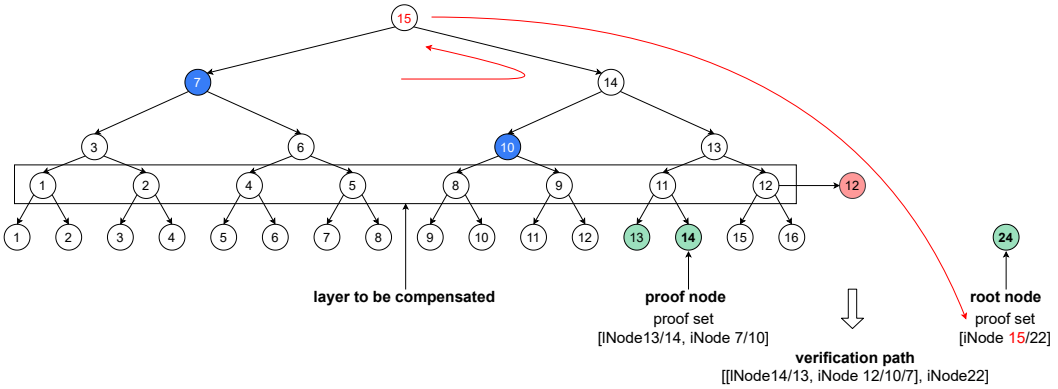


Fig. 5. Data verification when proof-node is 14 and root-node is 24. After computing their proof-sets, each intermediate layer would be filled if needed, then traverse until matching an iNode in the proof-set of lNode24.

Regarding the overall verification, `iNode15` is first verified from $root_{24}$ by $H_{concat(H_{15},H_{22})}$. Next, `lNode14` is verified by mutually hash calculation serially by set <`lNode13`, `iNode12/10/7`> (right or left sibling considered by ordinal comparison in each layer) if equivalent to the digest of `iNode15`.

3.1.4 VSA Storage. When a table `T` is defined as a ledger table, VeDB creates an implicit table called `T_VSA` within the same schema where `T` resides. `T_VSA` contains two columns named *vsa_serial* to record *VSA* non-leaf ordinals and *vsa_digest* to store related digests. A BRIN on key *vsa_serial* is also implicitly created on `T_VSA`, to accelerate the append-only ordinal matching by lossy searching. BRIN performs *VSA* ordinal searching as fast as a sequential valued filesystem. The latest *vsa_serial* and its pairwise *d_proof* are also refreshed in *ve_ledger* for fast retrieval. As a mostly used catalog table to store ledger table metadata with a small size, *ve_ledger* is designed as a memory resident, to get avoidance of LRU adaptation.

3.2 Ledger Table

Before introducing the detailed design of the ledger table, we review data mutation in ledger DBMSes first. Data immutability in conventional blockchains is a well-known prerequisite, which means none of the data mutations are supported once a piece of data is inserted into a blockchain. However, as introduced in Section 1, obsolete data leads to significant storage overhead while violated data should be deleted to meet the regulation requirement. Hence, ledger databases support data deletion before a specified retention window in their ledger DBMS to overcome storage overhead [5, 47, 68], and provide data erase functionality [51, 68] to be compliant with a regulation rule such as 'rights to be forgotten' [45] in GDPR [59]. VeDB implements data mutation by general standard SQL presented in Table 1 with constraint predicates using RBAC [26] operations presented in Section 2.7.

The ledger table is designed to store ledger metadata and is conducted separately from the temporal data table, which is briefly introduced in Section 2.4. A ledger table stores all audit trails including data mutation and trusted timestamp anchors besides its paired append-only temporal table data logs, the count size of the ledger table is, therefore, a superset of the temporal table. Ledger table includes columns defined as *rsn* (ledger ordinal), *digest* (record related hash value), *user* (operator), *trantype* (journal type), *c_sign* (client signature), and *e_rsn* (extensive *rsn* anchor) as depicted in Figure 6.

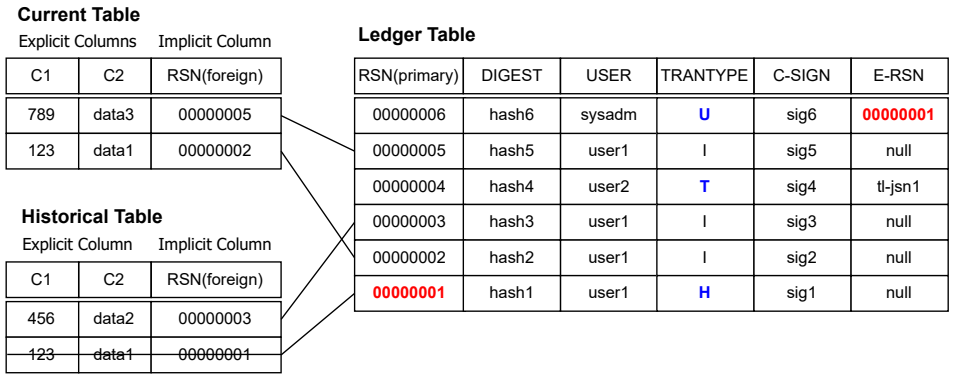


Fig. 6. Data and ledger tables in VeDB (Ve-S). Data tables and ledger table are connected by column RSN.

Table 2. Transaction types in the ledger table.

Action	Trantype	Digest Input	e-rsn
data insertion	I	row hex data	null
data updating	I	row hex data	null
ledger deletion (data purge)	D	concat('D', e-rsn)	ledger deleting point
data erasing (hiding)	U	concat('U', e-rsn)	erasing occurrence point
data erased (hided)	H	concat('H', e-rsn)	erased data point
trusted time notary	T	concat('T', e-rsn)	anchor point from TSA

We summarize enumerations of *trantype* with detailed value selection for *e_rsn* in Table 2. The digest computation is discussed by a different selection of *trantype*. In particular, digest computes raw data for a normal insertion or update, while a non-insertion operation coalesces *trantype* with *e_rsn*. Generally, the ledger table is append-only for temporal data tracing and auxiliary auditing. The only accepted updating is triggered by a data hiding transaction as described in Table 1, which forms an atomic operation between the hiding journal itself and the journal it referred to, i.e., the journal to be hidden. A successful data-hiding transaction then updates *trantype* column value to `'H'` on the hidden journal.

3.3 Fine-grained Verification

Most of the conventional ledger databases delegate verification to the server side, and reply to the client side with verification result or a proof path that can be verified on the client side to testify its integrity [5, 8, 47]. LedgerDB supports client-side verification for KV raw binary data only, not aiming for fine-grained structured data. We offer fine-grained column-level data verification at the client side, which is a useful and competitive feature in VeDB compared to conventional ledger DBMSes.

The main challenge of a fine-grained client-side verification lies in how to build an efficient parsing paradigm for the designated column with value, meaning how you interpret the hex data to be verified compared to its raw data format when digest calculated by server (e.g., a server side encoding for small int value 1 maybe 0x80000001, while the client side encodes it 0x00000001), and how you match the column locality in the raw data at the client side, not aware of the table schema.

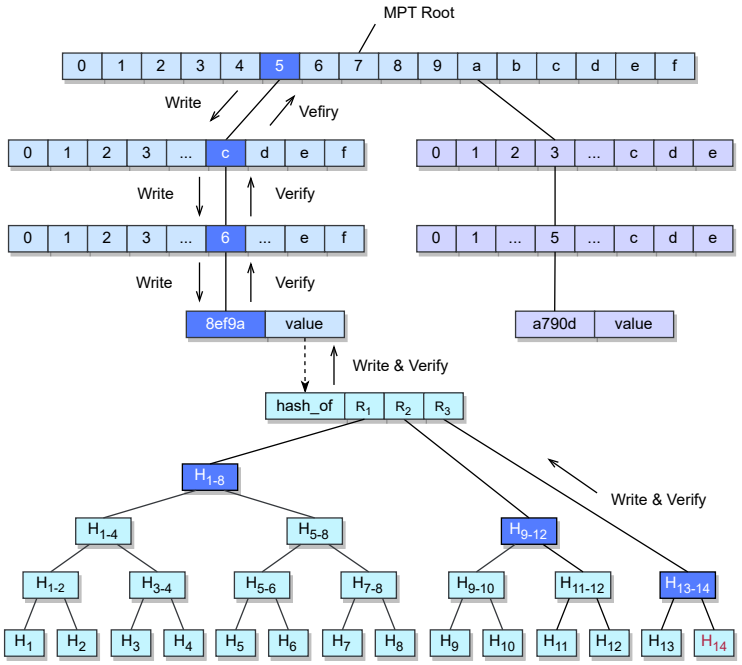


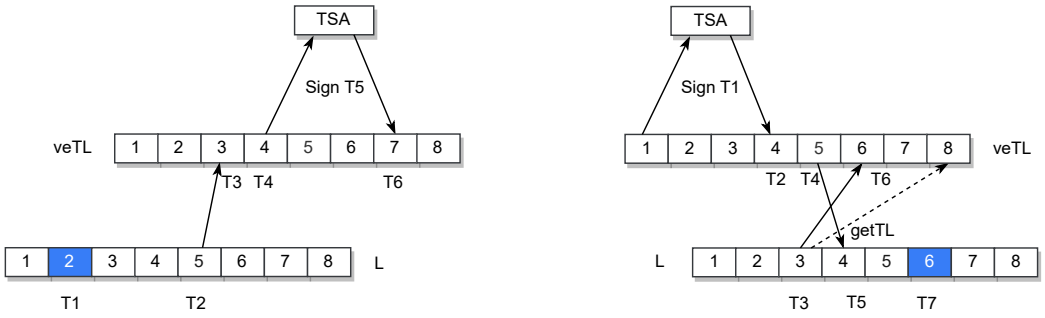
Fig. 7. Data lineage structure in Ve-S. A two-layer ADS combined by MPT and VSA realizes verifiable states.

We propose a data type assembled serialization method on server-side raw data to resolve these problems. When a user triggers a client-side verification to certain *rsn*, the database server will respond with a data type array composed of all table columns serially, together with the raw data that is the input of digest computation. Thus, the hex value of a specified column could be retrieved based on matching data type combined with actual data length from the first column until to find the designated column. To better explain the algorithm, an example is illustrated here: consider a table defined with c1 nullable INT (data type 497), c2 VARCHAR(20) (data type 448), c3 CHAR(8) (data type 452). A row data '*1, Hello, Ledger*' is encoded as 0x00800000010005c885939396d385848785994040 by EBCDIC CCSID 37 [30]. When we verify column c3 from the client side, the data type array [497, 448, 452] is also fetched. For the first data type, length value 5 is directly determined; for the second, since it is a varchar data type, so the length data is read from the raw data (0x0005) first then determine the actual buffer size 7 (length itself added). Thus, c3 data is located at offset 12 with length 8, i.e., 0xd385848785994040.

3.4 Data Lineage

Data lineage tracks all records related to a certain label (key). A complete trusted data lineage requires a full tamper-evidence trail to guarantee none of the entities are modified, and no one is intensively deleted or added in the queue that withstands *lineage attack* described in Section 2.1.

A typical permissioned blockchain [9] and ledger database solution is searching the lineage key in the database and proving the existence of all retrieved entries. This method cannot withstand intensive hiding of any record in the trail (*hide any* attack), e.g., considering we have 5 entries for a specified key, a server-side attacker can reply with entries 1,3,5 or 2,3, all of which can pass the verification. Another advanced solution [21, 31] is mutually entangling label entries one by one, but it is also vulnerable to tail entries hiding (*hide tail* attack). The attacker can respond 1,2,3 in our



(a) Posterior endorsement

(b) Anterior endorsement

Fig. 8. Trusted timestamp range in Ve-S. A TSA-endorsed trusted timestamp range is located via mutual entanglement between common ledger and time ledger, time ledger and TSA service.

example by hiding the last two entries. Technically, it can be verified by replaying a full audit, but the cost is unacceptable.

We implement an efficient Merkle Patricia Trie (MPT) [7] and *VSA* combined data structure to support verifiable data lineage in Ve-S as depicted in Figure 7. Lineage keys are stored in the MPT. All the journals related to a certain key form a *VSA* stores their digests, and the *VSA* root is stored as a value of the MPT searched key. Once a new journal to a certain lineage key is inserted, the pairwise *VSA* is filled with a new root computed, which is then stored as the value of the specified key searched from MPT. Finally, the MPT root is updated by bottom-up computation. Consider inserting the 14th journal of lineage key `5c68ef9a` as illustrated in Figure 7. We fill digests of *VSA* leaf node 14 and meta node 11, and calculate *VSA* root based on concatenated `<iNode 7,10,11>`, then calculate the latest MPT root based on the new *VSA* root. If we verify this lineage key, we first rebuild the *VSA* with the *rsn*-14 journal's digests and validate the root is the same as stored on MPT, then verify the path from the validated value to the MPT root.

3.5 Auditability

A ledger DBMS is externally auditable if all its ledger data integrity can be rigorously authenticated by an independent third party, while a ledger DBMS is internally auditable if it cannot be rigorously proved to be trusted by an external user. Permissioned blockchains and most conventional ledger DBMSes (e.g., QLDB, SQL Ledger, Oracle blockchain table, etc) only support internal auditability, meaning their architectures cannot resist ledger and systematic timestamp forging attacks. LedgerDB introduced a TSA-based two-way peg protocol that limits the attacking time window into intervals [67]. However, a theoretical and practical credible time-range implementation is not provided in existing ledger databases.

We propose a trusted time-range workflow which offers legally compliant (e.g., Internet court [63]) timestamp, based on a practical timestamp ledger as an intermediate TSA tradeoff called *veTL*. *veTL* implements mutual endorsement with both TSA and common ledger to provide trusted timestamps as a public could service on Volcengine. The trusted time-range workflow is depicted in Figure 8 and the detailed procedure is discussed below.

Posterior endorsement. The aim is to find the nearest posterior TSA endorsement. As depicted in Figure 8a, transaction *L-rsn-2* on common ledger *L* happens at T1. Afterwards, *L-rsn-5* is anchored onto *veTL* as *TL-rsn-3*. Next, *TL-rsn-4* seeks for TSA endorsement whose receiving timestamp is T5 and anchored back to *veTL* as *TL-rsn-7* at T6. In this demonstration, we say *L-rsn-2* happens before

the TSA signing timestamp T5. Our goal is to find the nearest credible endorsement. Given a *rsn x* from common ledger *L*, search the nearest greater *rsn y* on *veTL* anchored from *L*. Next, locate the nearest latter TSA anchor *rsn* and read its content.

Anterior endorsement. To find the nearest anterior TSA endorsement, we leverage the anchor on *L* from *veTL*. *TL-rsn-4* is the TSA endorsement journal at T1 on *veTL* as depicted in Figure 8b. Next, *TL-rsn-5* anchors back onto *L* as *L-rsn-4*. After that, transaction *L-rsn-6* is inserted. We say *L-rsn-6* happens after the TSA endorsing timestamp T1. The anterior endorsement algorithm can be described as, given a common ledger *rsn x*, search its anterior-most anchor from *veTL* (*TL-jsn y*) on the common ledger, then locate the earlier TSA journal on *veTL* and read its data. Remind it is important for *veTL* anchoring onto common ledger to get a rigorously anterior credible timestamp. If we only use one-way pegging from common to time ledger, there isn't a credible way to find an anterior trusted timestamp. As illustrated in Figure 8b, it would appear *L-rsn-3* and *TL-rsn-8* to be the logic, because *TL-rsn-8* is anterior to *TL-rsn-4* and *L-rsn-3* is prior to *L-rsn-8*. However, if taken a deeper analysis, the latency from *L-rsn-3* to *TL-rsn-8* can compromise the whole logic.

4 TEE-ASSISTED TRUSTED DBMS

In this section, we introduce our motivation to devise Ve-H in VeDB with detailed application analysis and describe Ve-H trusted feature design.

4.1 Motivation

We design Ve-H to make use of TEE to ensure tamper-resistance and efficiently support data lineage.

In the prior section, we discussed our authenticated DBMS framework in a software-oriented architecture. However, as introduced in Section 3.5, time window still exists for malicious tampering in verifiable software conductions. A trusted hardware-enabled systems yet do not have this problem. A TEE enclave can execute a pre-authorized code and sign the receipt in a small Trusted Computing Base (TCB), which is unlikely to be tampered with and easy for the client to validate.

As noted in Section 1, most ledger databases are vulnerable to data lineage. LedgerDB proposed clue-based ADS [67, 68] to support rigorously verifiable data lineage. We also provide our MPT and *VSA*-based ADS for data lineage. However, all these algorithms suffer from significant random I/O overhead caused by MPT, which limits the overall system performance. A non-tailored MPT implementation in Ethereum [7] just performs around 3000 tps. Thus we realize TEE-enabled lineage system design to increase data lineage application performance.

4.2 Ve-H Design

Similar to Ve-S, Ve-H protects the integrity of all historical records. It achieves the goal by fulfilling the three crucial requirements in its design: 1) Content tampering must be prohibited. 2) Insertion into a prior location must be discovered. 3) Deletion of a historical record must be detected. Following the design requirements, we devise a TEE hardware-enabled methodology with implementations, as an efficient alternative for users' trusted options, which overcomes the malicious time window and inefficient data lineage problems in software-based algorithms introduced in Section 4.1.

We leverage TEE enclave digest signing, monotonic counter, and trusted timestamp to meet all the discussed goals, and thoroughly avoid mutual entanglement used in conventional software algorithms. The trusted insertion functionality goes through the following steps: 1) DBMS calculates the digest of a row to be inserted, then 2) sends the digest to the enclave. 3) A trusted timestamp authenticated code is triggered when TEE receives the call. 4) TEE then assigns a new monotonic counter for the specified ledger table, and 5) signs the set composition of digest, monotonic counter, and the trusted timestamp. 6) Lastly, enclave returns DBMS with the signed data to store on *tee_sign*. Note that we control implicit column creation by differentiating `TYPE` in the ledger table DDL

introduced in Table 1. The default value `LOCAL` stands for using software Ve-S, while `TEE` represents implementing hardware Ve-H features.

When we verify a column value or a row, the digest of the row is recalculated and compared with the TEE signed value first, if matched, the signature is further validated to conclude the verification. The overall verification complexity is $O(1)$ which is faster than software methods with an efficient signature cryptographic algorithm conducted. An entire ledger table audit process is straightforward by verifying each row with assigned ordinal monotonically until reaches the latest ordinal proved by enclave.

One may argue that a trusted timestamp could resolve the ordering problem since they are already utilized in our system. Nevertheless, a deletion of a prior record cannot be detected by this method. Furthermore, an attacker may generate a risky record at a certain timestamp, then decide whether to cut in line later based on up-to-date conditions. Hence, both 2) and 3) of the design goals cannot be satisfied by simply using trusted timestamps.

We allocate a fine-grained monotonic counter for each key in the data lineage application. A unique monotonic counter generator is assigned to a specific label, i.e., key of data provenance, and follows the discussed data workflow for data notarization and full audit. Monotonic counter makes *hide any* attack impractical. Since the latest counter can be proved by TEE, the *hide tail* attack described in Section 3.4 is also resolved in an effective way, more efficient compared to Ve-S solution.

4.3 Ve-H Implementation

In Ve-H, we require monotonic counters and trusted timestamps inside TEE. Thus, we choose to implement Ve-H with Intel Software Guard Extensions (SGX). On one hand, SGX allows an application to create an attestable TEE enclave. The confidentiality and integrity of an enclave are protected by hardware. On the other hand, SGX supports monotonic counters and trusted timestamps, which are protected by hardware as well. Besides, we adopt Intel SGX SDK 2.1.3, which provides all functions we require in its trusted runtime APIs [20]. The programming model of the Intel SGX SDK [32] allows developers to define the interface between the untrusted application side and the trusted enclave side through Enclave Definition Language (EDL). In addition, Intel SGX SDK provides glue code generation for the processor mode switching and function calling between the untrusted application side and the trusted enclave side. The invocation into the enclave is referred to as ECALL and the invocation outside of the enclave is referred to as OCALL. Besides, Intel SGX SDK provides common utilities for enclave through Trusted Runtime System (tRTS) and optional helper libraries which can be included in trusted runtime (such as platform services).

In our implementation, we define EDL for Ve-H and include the necessary trusted runtime libraries for cryptographic operations, monotonic counter, and trusted timestamp. We implement ECALL functions to allow DBMS to send the digest to the enclave. Inside the enclave, the digest would be concatenated with fetched trusted timestamp and monotonic counter as required to create a bundle. The bundle would be signed by a predefined enclave identity. Note that, we allow clients to remote attest on the enclave to reach a consensus on the authority of the enclave identity, which would sign on the composition of digest, monotonic counter, and the trusted timestamp. In the end, we implement OCALL to call back DBMS to store the signed bundle.

4.4 Ve-H Ongoing Work

Although Intel SGX provides monotonic counters and trusted timestamps inside TEE, the performance overhead added is nontrivial. The Intel SGX adds multiple ECALL and OCALL on each Ve-H insertion operation for sake of the trusted platform services. For each Ecall (warm cache), it brings an 8,640 median latency measured in CPU cycles and for each Ecall (cold cache), it brings a

14,170 median latency in CPU cycles. For each Ocall (warm cache) it brings an 8,314 median latency in CPU cycles and for each Ocall (cold cache), it brings a 14,160 median latency in CPU cycles [62].

To improve the latency brought by leveraging Intel Converged Security and Management Engine (CSME). We would bring in the idea of Physical Secure Timer implemented in ARM architecture [60]. We bind a physical secure timer to a specific enclave id and allow the remote parties to remotely attest to an enclave and its bonded timer. We would implement such a secure timer extension on Risc-V enclave [61]. For platforms without a natively supported secure timer, we will extend the Hardware Root of Trust proposed by Open Compute Project (OCP) [23].

5 EVALUATION

This section presents experimental results of verifiable feature performance and end-to-end performance with and without trusted features in VeDB.

5.1 Hardware Configuration

We conduct experiments in different hardware environments listed below, to get a comprehensive evaluation of *VSA* efficiency, Ve-S and Ve-H comparison, and VeDB end-to-end performance.

prod-env. All cloud experiments are conducted on g2i.16xlarge instance, which runs Debian 9 Linux 5.4 and is equipped with 64 Intel(R) Xeon(R) Platinum 2.4GHz CPU cores, 256GB RAM, and 5TB SSD storage from a public storage service on ByteDance Volcengine Cloud. All nodes are connected via 25Gb Ethernet.

sgx-env. All Ve-S and Ve-H application experiments are conducted on an Intel SGX machine running Ubuntu 18.04.5 LTS system equipped with 4 CPU cores, 16GB RAM, and 1TB SSD storage.

5.2 Workloads

We evaluate *VSA* algorithm performance with comparison to Merkle accumulator (M-acc) and batch accumulated Merkle tree (bAMT) [68] in Section 5.3. Thus we divide the insertion part into three steps: 1) transaction hash calculation, 2) authenticated data structure composition, and 3) receipt generation. We compute SHA256 digests offline, meaning step 1) is skipped during the insertion performance testing for *VSA*, M-acc, and bAMT, to get a fair algorithm-level comparison. We extract the bAMT implementation from an open-source re-implemented LedgerDB repository [44]. The data structure is implemented in C++ with RocksDB 5.8 as the storage layer. M-acc is conducted by setting the batch size to 1 in bAMT.

We evaluate VeDB end-to-end performance with and without trusted features in Section 5.5. VeDB instance is deployed on *prod-env*, which is our production environmental configuration. We use TikTok copyright notarization system (Section 2.8) table with schema containing music and video ID, media and original types, word and cover texts, camera and voice information, etc, whose relational row size is around 2KB. The workload is conducted by using the most recently one billion uploaded video/music records on TikTok after data masking. A baseline experiment without trusted features for insertion and selection is first tested, then the workload is reloaded to test insertion and verification performance for trusted features by turning on the `IMMUTABLE` parameter during table creation.

5.3 VSA Experiments

We evaluate *VSA* authenticated data insertion and verification performance in comprehensive dimensions with detailed comparisons with the Merkle tree-based data structures introduced in Section 5.2, i.e., M-acc and bAMT.

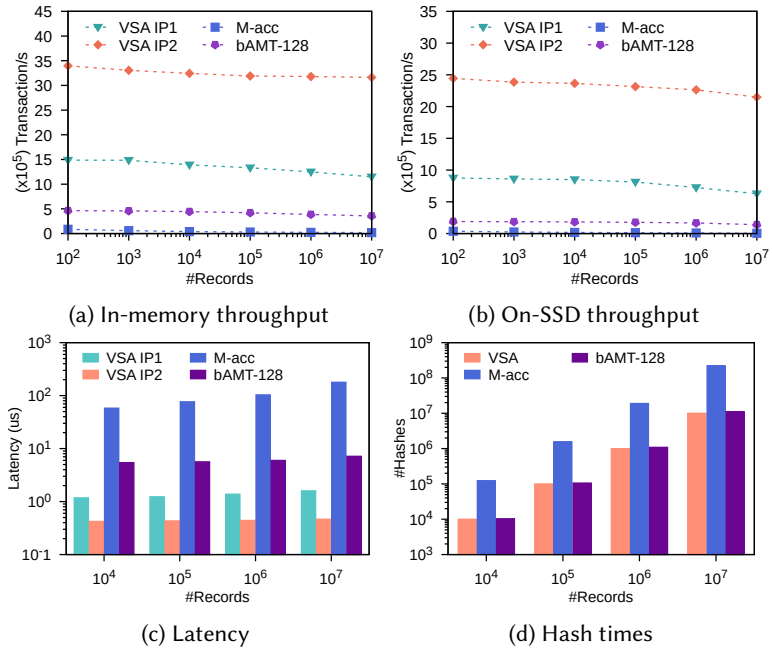


Fig. 9. VSA insertion performance evaluation and comparison with M-acc, bAMT. VSA in-memory insertion outperforms M-acc and bAMT-128 up to 76× and 7.7× on average, respectively. In disk accessing insertion, VSA performs 135× and 13.7× faster than that in M-acc and bAMT-128, respectively.

5.3.1 Data Insertion. We first evaluate *VSA* insertion throughput in direct memory and disk access modes, and compare its performance with Merkle tree-based ADS, i.e., M-acc and bAMT-128 (batch size 128). We vary basic ledger transaction amounts from 100 to 10 million and test their insertion throughputs on *prod-env*.

As shown in Figure 9a, *VSA* IP-1 in-memory version sustains around 1.3M tps while *VSA* IP-2 performs around 3.2M tps. We observe that both IP-1 and IP-2 throughputs decreasing rate from 10,000 to 10 million transaction amounts are 17.2% and 2.5%, respectively. We can see that M-acc and bAMT perform 86K and 463K tps when the journal size is 100 respectively, and decrease to 18K and 36K when the journal size grows to 10 million. The experiments show that *VSA* IP-2 outperforms M-acc around 76× and bAMT-128 up to 7.7× respectively, due to significantly lower CPU cost for the ADS construction.

VSA disk accessing insertion throughput evaluation is described in Figure 9b. IP-1 sustains more than 800K tps, and IP-2 reaches 2.4M tps at peak. We observe that IP-2 outperforms IP-1 by up to 3×. We notice that as an array-based efficient I/O design, *VSA* insertion outperforms M-acc by 135× which is 1.8× faster than the in-memory comparison, due to M-acc random I/O storage overhead. *VSA* IP-2 is 13.7× faster than that in bAMT-128 on average.

Figure 9c shows the latency for the target authenticated data structures conducted on *prod-env* when we vary the latest *rsn*, by testing the consecutive 100 insertions, then average the latency. We can see IP-1 executes 3.2× lower than IP-2 due to more CPU and I/O cost involved when computing and retrieving the latest proof set for IP-1. The overall latency of M-acc is 247× higher than *VSA* IP-2, mainly because of the significant random I/O overhead of the KV store in M-acc, compared to the append-only file flushing in *VSA*. The insertion latency in bAMT is 14.4× higher than that in *VSA* on average.

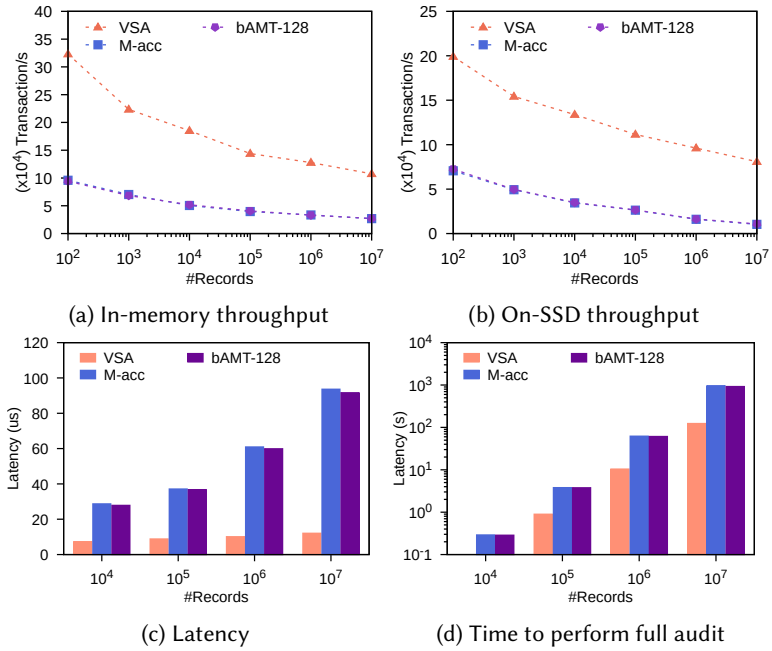


Fig. 10. VSA verification performance evaluation and comparison with M-acc, bAMT. The peak throughput in VSA in-memory verification is 3.4× and 4× higher than that in M-acc and bAMT-128, respectively. VSA verifies 3.8× and 3.6× faster than that in M-acc and bAMT-128 on average, in disk accessing.

A breakdown evaluation for CPU-intensive SHA256 calculation is shown in Figure 9d. We vary overall ledger transaction amounts and summarize the total SHA256 computations for all mentioned algorithms. We observe that when transaction amount is 10,000, M-acc computes 12.4× higher SHA256 compared to *VSA*, and 22.3× higher respectively. *VSA* saves 11.2% SHA256 calculations than bAMT when transactions accumulate to 10 million.

5.3.2 Data Verification. Our data verification evaluation also covers direct memory and disk access modes. We vary the verification basis $(r_2, root_2)$ pair from 100 to 10 million, and randomly pick up 100 r_1 to be verified, then average their throughputs. The full audit experiments are conducted by varying ledger volume and measure the entire ledger auditing from genesis to the latest *rsn*.

Figure 10a shows the in-memory (*prod-env*) verification throughputs of all discussed ADS. This experiment aims to test CPU intensive cost of all mentioned algorithms, avoiding I/O operations. We can observe that *VSA* verification reaches 323K tps when r_2 is 100, and gets 108K tps when r_2 arrives at 10 million, which outperforms M-acc to 3.4× and 4×, bAMT to 3.4× and 3.9×.

Figure 10b shows the disk accessing verification throughputs of the different algorithms conducted on *prod-env*. *VSA* verification reaches 199K tps when r_2 is 100, then gradually decreases when r_2 grows, and finally reduces to 81K tps when r_2 reaches 10 million. We can see that *VSA* verification throughputs are 3.8× and 3.6× higher than M-acc and bAMT on average.

The average verification latency experimental results are shown in Figure 10c, whose empirical setup is the same as the insertion latency experiment discussed in Section 5.3.1. We observe that *VSA* verification latency is around 1/4 of M-acc and bAMT when r_2 is 100, and around 12us latency when r_2 grows to 10 million, with 1/8 and 2/15 of that in M-acc and bAMT respectively. This is

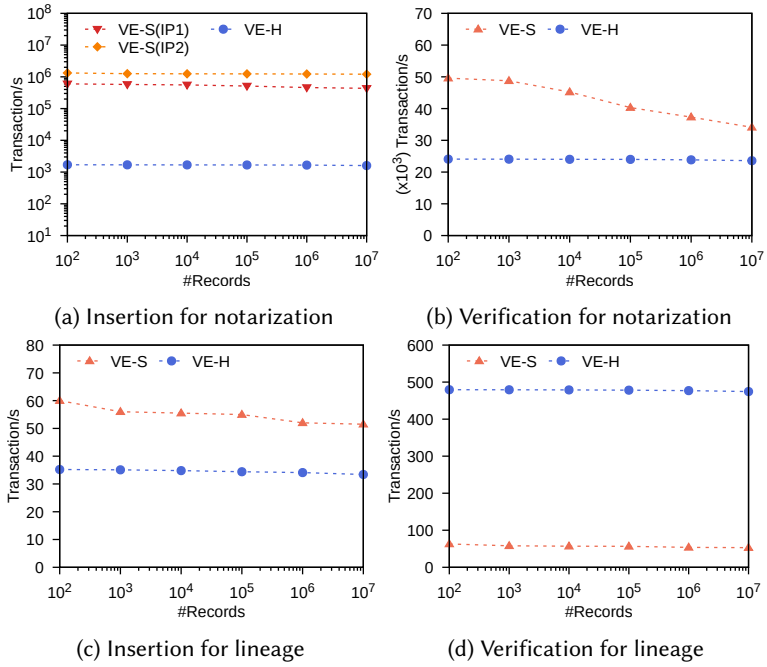


Fig. 11. System performance comparison between Ve-S and Ve-H in data notarization and lineage applications. Ve-H verifies 8.5× faster than Ve-S in data lineage application, while Ve-S is faster in the other experiments.

because *VSA* verification advantages in saving CPU and random I/O are significant when the ledger grows, compared to M-acc and bAMT.

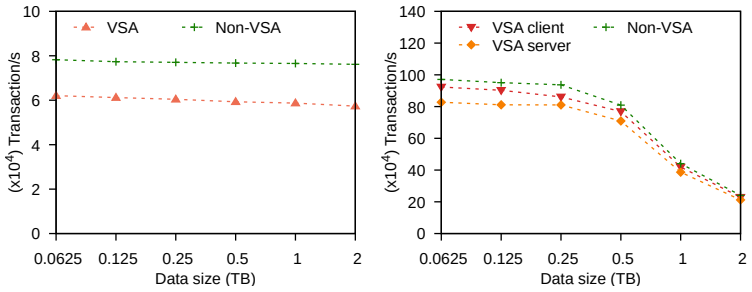
Figure 10d shows the full audit experimental results of the authenticated algorithms. For a table with 10,000 transactions, *VSA* uses 0.07s to audit the entire ledger, while M-acc costs 0.29s and bAMT spends 0.28s. For 10 million transactions, *VSA* uses 124s for the full audit, while M-acc and bAMT spend 7.8× and 7.5× more time respectively.

5.4 Ve-S and Ve-H Applications

We evaluate VeDB application, i.e., data notarization and data lineage, performance of Ve-S and Ve-H run on *sgx-env* separately, and compare their efficiency. We use the *VSA* testing scenario introduced in Section 5.2 to evaluate data notarization, and the same workload by adding additional lineage keys with 0 to 100 random entries assigned to each key for data lineage testing. Ve-H is conducted using `sgx_increment_monotonic_counter` and `sgx_get_trusted_time` API for monotonic counters and trusted timestamps. The proof set is signed by `sgx_rsa3072_sign` within the enclave and verified at the client side by `sgx_rsa3072_verify`.

Figure 11a and 11b show the data notarization system throughput of both systems. We can see both data insertion and verification of Ve-H sustains at a stable level when the ledger grows. It is because Ve-H proof generation and verification are not related to transaction amounts. We observe all Ve-H and Ve-S perform dozens of Ktps in verification throughput. However, the insertion performance of Ve-H is significantly lower compared to that in Ve-S, due to the SGX monotonic counter and trusted timestamp performance limitations.

Figure 11c and 11d show data lineage throughput in Ve-S and Ve-H. Both Ve-S and Ve-H lineage insertion throughputs are less than 100 tps for the MPT bottleneck in Ve-S and the enclave monotonic



(a) Performance of insertion (b) Performance of verification

Fig. 12. VeDB end-to-end system performance evaluation with and without enabling trusted features, when data table size grows from 64GB to 2TB. VSA reaches 78% insertion throughput of that in the baseline.

counter limitation in Ve-H respectively. We observe that Ve-H outperforms Ve-S around 8.5× in lineage verification, proving as an efficient alternative to the software solution.

The selection of Ve-S and Ve-H in a ledger use case could be a tradeoff between trusted basis, hardware support and application types. If the use case focuses on data notarization and a TEE-based hardware is not easy to be supported, Ve-S would be a good choice. If a data lineage focused use case can be easily deployed on a TEE-based environment, Ve-H would be a better choice.

5.5 VeDB End-to-end Experiments

We evaluate end-to-end VeDB system performance with baseline (trusted features not added) comparisons on broad operations, e.g., insertion, selection, verification, using the TikTok copyright data notarization workload described in Section 5.2. We measure the average insertion throughputs when data table size grows from 64GB to 2TB, for baseline and *VSA* tables enabled, i.e., table defined with `IMMUTABLE YES`. All experiments are run with 256 concurrent clients submitted to *prod-env* server with 64 CPU cores.

We observe that the original insertion sustains around 76K tps, while *VSA* performs above 58K tps as shown in Figure 12a. The insertion performance impact caused by enabling the trusted feature is not significant in VeDB, with about 22.3% of performance degradation. Note the current Ve-S inserting implementation is not optimized with parallelism, hence there are several subroutines that can be optimized by parallel computing such as hash calculation, *VSA* construction, and *t_vsa* insertion.

Our end-to-end data verification performance experiments are conducted both on the server side and the client side, with a comparison to row selection using sargable predicate (marked as Non-VSA) performance as shown in Figure 12b. Server-side verification is conducted via the DBMS `VERIFY_ROW` function. Client-side verification is implemented via `Verify` API in client SDK, with a prerequisite that *VSA* data is already downloaded offline by `GET_VSA`.

We observe that the client-side verification sustains around 930K tps when the table size is less than 256GB, and performs 240K tps when the transaction amount reaches one billion (table size 2TB). The server-side verification performance is with a similar trend but less than the client-side verification. This is because *VSA* file is more efficient than the database table page, and the client-side verification thread can be parallelized with data selection, to overcome the server-side bottleneck. The client-side verification achieves 94.4% of the throughput of selection, and server-side verification reaches 86.4% of the selection baseline. Hence, *VSA*-based verification in DBMS is just slightly less than related selection performance, due to both its CPU and I/O efficiency compared to other DBMS executions.

6 RELATED WORK

Blockchain systems [15, 43] have captured enormous attention in computer sciences. As a typical feature, verifiable tamper-evidence in blockchains is often conducted by hash-tree algorithms [25, 41]. However, the low system throughput limited applications on permissionless blockchains. Permissioned blockchains yield a higher performance than their permissionless counterparts, but their decentralization deployment in real applications have not been strong. As an empirical permissioned blockchain, Hyperledger Fabric [3] changes its consensus protocol from PBFT [19] in its earlier version to the current Raft protocol, which practices centralized consistent protocol regardless of malicious assumption. Another common problem in conventional blockchains is their general immutability, which is impractical in real-world applications to be compliant with data regulations like GDPR [59].

Centralized ledger databases [24, 38, 53] are implemented in recent years, whose vision is to overcome limited performance in conventional blockchains but with equivalent or better auditability. QLDB [8] invokes a 'trust but verify' threat model, but it lacks external auditability. LedgerDB [68] discloses a 'Client-Sever-TSA' signing and pegging workflow to offer rigorous audibility, and also supports native data mutation and lineage. Nonetheless, its advanced MPT-based data lineage structure is a significant performance overhead in DBMS. Oracle blockchain table adopts the mutual signing workflow [35] and data deletion before a retention window [47]. Microsoft SQL ledger [5] also utilizes ADS to ensure tamper-resistance in the relational database, and implements data truncation as a type of mutation.

Outsource databases (ODB) often implement authenticated data structures (ADS) [34, 36, 40] to guarantee data integrity. However, ADS-enabled systems can only support limited verifiable query patterns [10, 11, 65]. TEE-assisted database systems [33, 54] emerge in recent years. However, most of these systems are built for data encryption purposes. Cipherbase [6] integrates user-defined hardware to perform operations on encrypted data. EnclaveDB [50] uses TEE to implement in-memory database confidentiality and integrity. StealthDB [58] is an encrypted database system from Intel SGX. TrustedDB [12] uses TEE to perform operations, but the trusted zone involves large portions of the DBMS engine.

7 CONCLUSIONS

In this paper, we introduce VeDB as a software (Ve-S) and hardware (Ve-H) enabled trusted database. We devise a new append-only array-based authenticated data structure called verifiable Shrubs array (*VSA*), distribute the two-layer ordinals into an array structure, and decompose sparse sub-root set, as opposed to the conventional tree-based models. We also present fine-grained client-side verification and credible timestamp range in Ve-S. Lastly, we introduce Ve-H, composed of TEE-assisted monotonic counters and trusted timestamps, to offer practical data lineage application with better performance compared to Ve-S.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. We also express our gratitude to Xionghan Wei, Qi Yang and Wenmao Zhang for their help on TikTok copyright application development and product works.

The work of Beng Chin Ooi and Cong Yue is supported by the National Research Foundation, Singapore under its Emerging Areas Research Projects (EARP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Carlisle Adams, Pat Cain, Denis Pinkas, and Robert Zuccherato. 2001. *Internet X. 509 public key infrastructure time-stamp protocol (TSP).* Technical Report.

[2] Gluchowski Alex, Gurkan Kobi, Olszewski Marek, Tromer Eran, and Vlasov Alexander. 2019. *Shrubs - A New Gas Efficient Privacy Protocol.* https://archive.devcon.org/archive/watch/5/shrubs-a-new-gas-efficient-privacy-protocol/?tab=YouTube

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.

[4] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. 2020. Azure SQL database always encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1511–1525.

[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 *Proceedings of the 2021 International Conference on Management of Data*. 2437–2449.

[6] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure database-as-a-service with cipherbase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1033–1036.

[7] The Ethereum Association. 2022. *PATRICIA MERKLE TREES.* https://ethereum.org/developers/docs/data-structures-and-encoding/patricia-merkle-trie/

[8] AWS. 2018. *Amazon quantum ledger database (qldb).* https://aws.amazon.com/qldb

[9] AWS. 2022. *Amazon Managed Blockchain.* https://aws.amazon.com/blockchain

[10] Michael Backes, Dario Fiore, and Raphael M Reischuk. 2013. Verifiable delegation of computation on outsourced data. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 863–874.

[11] Sumeet Bajaj and Radu Sion. 2013. CorrectDB: SQL engine with practical query authentication. *Proceedings of the VLDB Endowment* 6, 7 (2013), 529–540.

[12] Sumeet Bajaj and Radu Sion. 2013. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering* 26, 3 (2013), 752–765.

[13] Abdeljalil Beniiche. 2020. A study of blockchain oracles. *arXiv preprint arXiv:2004.07140* (2020).

[14] D Bider and M Baushke. 2012. *SHA-2 data integrity verification for the secure shell (SSH) transport layer protocol.* Technical Report.

[15] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014), 2–1.

[16] ByteDance. 2022. *Make Your Day - TikTok.* https://www.tiktok.com/

[17] ByteDance. 2023. *Volcengine.* https://github.com/volcengine

[18] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking {RocksDB}{Key-Value} Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[19] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[20] Shanwei Cen and Bo Zhang. 2017. Trusted time and monotonic counters with intel software guard extensions platform services. *Online at: https://software. intel. com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services. pdf* (2017).

[21] Alibaba Cloud. 2022. *Alibaba Blockchain as a Service.* https://www.alibabacloud.com/product/baas

[22] Beijing Internet Court. 2023. *Beijing Internet Court.* https://english.bjinternetcourt.gov.cn

[23] Yigal Edery and Rajeev Sharma. 2020. OCP Security Announces version 1.0 specs for Root of Trust. https://www.opencompute.org/blog/ocp-security-announces-version-10-specs-for-root-of-trust

[24] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB: A shared database on blockchains. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1597–1609.

[25] Zachary Amsden et al. 2019. The Libra Blockchain. *The Libra Association.* https://mitsloan.mit.edu/shared/ods/documents?PublicationDocumentID=5859

[26] David Ferraiolo, Janet Cugini, D Richard Kuhn, et al. 1995. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th annual computer security application conference*. 241–48.

[27] Gartner. 2020. *Gartner Top 10 Trends in Data and Analytics for 2020.* https://www.gartner.com/smarterwithgartner/gartner-top-10-trends-in-data-and-analytics-for-2020

[28] Sanjay Ghemawat and Jeff Dean. 2014. Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values.

[29] Mike Hearn and Richard Gendal Brown. 2016. Corda: A distributed ledger. *Corda Technical White Paper* 2016 (2016). https://www.corda.net/content/corda-technical-whitepaper.pdf

[30] IBM. 2021. *CODEPAGE option syntax.* https://www.ibm.com/docs/en/cobol-zos/6.3?topic=options-codepage
[31] IBM. 2022. *IBM Blockchain.* https://www.ibm.com/blockchain
[32] R Intel. 2020. Software guard extensions sdk developer reference for linux* os. (2020).
[33] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019.* 1–15.
[34] John Kuszmaul. 2019. Verkle trees. *Verkle Trees* 1 (2019).
[35] SQL Maria. 2021. *Why Oracle Implement Blockchain in the Database.* https://sqlmaria.com/2021/03/03/why-oracle-implement-blockchain-in-the-database
[36] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. 2004. A general model for authenticated data structures. *Algorithmica* 39, 1 (2004), 21–41.
[37] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. 2019. {BITE}: Bitcoin lightweight client privacy using trusted execution. In *28th USENIX Security Symposium (USENIX Security 19).* 783–800.
[38] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. 2016. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB* (2016).
[39] Ralph C Merkle. 2019. Protocols for public key cryptosystems. In *Secure communications and asymmetric cryptosystems.* Routledge, 73–104.
[40] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated data structures, generically. *ACM SIGPLAN Notices* 49, 1 (2014), 411–423.
[41] Arun Prasad Mohan, Angelin Gladston, et al. 2020. Merkle tree and Blockchain-based cloud data auditing. *International Journal of Cloud Applications and Computing (IJCAC)* 10, 3 (2020), 54–66.
[42] JP Morgan. 2016. Quorum whitepaper. *New York: JP Morgan Chase* (2016).
[43] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008). https://bitcoin.org/bitcoin.pdf
[44] National University of Singapore. 2022. *Ledger Database.* https://github.com/nusdbsystem/LedgerDatabase
[45] The Horizon 2020 Framework Programme of the European Union. 2020. *Everything you need to know about the Right to be forgotten.* https://gdpr.eu/right-to-be-forgotten
[46] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting {SGX} Enclaves from Practical {Side-Channel} Attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18).* 227–240.
[47] Oracle. 2022. *Blockchain Tables in Oracle Database 21c.* https://oracle-base.com/articles/21c/blockchain-tables-21c
[48] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
[49] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the twenty-third ACM symposium on operating systems principles.* 85–100.
[50] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP).* IEEE, 264–278.
[51] Provendb. 2020. *Provendb: A blockchain enabled database service.* https://provendb.com/litepaper
[52] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.
[53] Nathan Senthil, Govindarajan Chander, Saraf Adarsh, et al. 2019. Blockchain meets database: design and implementation of a blockchain relational database [J]. In *Proceedings of the VLDB Endowment*, Vol. 12. 1539–1552.
[54] Rohit Sinha and Mihai Christodorescu. 2018. Veritasdb: High throughput key-value store with integrity. *Cryptology ePrint Archive* (2018).
[55] Paul Snow, Brian Deery, Jack Lu, David Johnston, Peter Kirby, Andrew Yashchuk Sprague, and Dustin Byington. 2014. Business processes secured by immutable audit trails on the blockchain. *Brave New Coin* (2014).
[56] Roberto Tamassia. 2003. Authenticated data structures. In *European symposium on algorithms.* Springer, 2–5.
[57] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking data on Intel CPUs via cache evictions. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 339–354.
[58] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 370–388.
[59] Paul Voigt and Axel Von dem Bussche. 2017. The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing* 10, 3152676 (2017), 10–5555.
[60] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. 2022. RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone. In *2022 IEEE Symposium on Security and Privacy (SP).* IEEE Computer

Society, 1573–1573.
[61] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v.. In *NDSS*.
[62] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 81–93.
[63] Wikepedia. 2022. *Hangzhou Internet Court*. https://en.wikipedia.org/wiki/Hangzhou_Internet_Court
[64] Wikimedia. 2021. *Block Range Index*. https://en.wikipedia.org/wiki/Block_Range_Index
[65] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data*. 141–158.
[66] Xinying Yang. 2020. Blockchain-based music originality analysis method and apparatus. US Patent 10,628,485.
[67] Xinying Yang, Sheng Wang, Feifei Li, Yuan Zhang, Wenyuan Yan, Fangyu Gai, Benquan Yu, Likai Feng, Qun Gao, and Yize Li. 2022. Ubiquitous Verification in Centralized Ledger Database. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1808–1821.
[68] 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. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3138–3151.
[69] Meihui Zhang, Zhongle Xie, Cong Yue, and Ziyue Zhong. 2020. Spitz: A Verifiable Database System. *Proceedings of the VLDB Endowment* 13, 12 (2020).

Received November 2022; revised February 2023; accepted March 2023