V^{2}FS: A Verifiable Virtual Filesystem for Multi-Chain Query Authentication

Haixin Wang[1], Cheng Xu[1], Xiaojie Chen[2], Ce Zhang[1],
Haibo Hu[3], Shikun Tian[2], Ying Yan[2], Jianliang Xu[1]

[1] *Hong Kong Baptist University* [2] *Ant Group* [3] *Hong Kong Polytechnic University*
{hxwang, chengxu, cezhang, xujl}@comp.hkbu.edu.hk,
{tianxun.cxj, shikun.tiansk, fuying.yy}@antgroup.com haibo.hu@polyu.edu.hk

***Abstract*—The rise of decentralized finance (DeFi), Web 3.0, and other blockchain-based applications has led to an increased demand for on-chain data analysis across multiple blockchains. Conducting advanced queries, such as data aggregation and correlation analysis, is essential for gaining valuable insights in this context. However, multi-chain queries pose several challenges for the querying system, including compatibility with existing blockchains, supporting diverse query types, and ensuring the integrity of query results. To tackle these challenges, we propose a novel paradigm called *verifiable virtual filesystem* (V^{2}FS). V^{2}FS extends the POSIX I/O interface, shifting the focus from verifying computation to verifying data. This innovative approach empowers query clients to leverage an off-the-shelf database engine to evaluate queries using verifiable data retrieved from an indexing service provider (ISP). Our solution ensures strong integrity guarantees and can be smoothly integrated with existing database engines to support various query types. To achieve blockchain compatibility, we utilize the DCert framework to certify blocks from different blockchains, making our system applicable to various blockchain systems. Furthermore, we propose cache-based algorithms and a bloom filter-integrated algorithm to optimize query performance and minimize network communication costs. Security analysis and empirical study validate the effectiveness and efficiency of the proposed system.**

I. Introduction

The emergence of decentralized finance (DeFi), Web 3.0, and other blockchain-based applications in recent years has sparked a significant need for on-chain data analysis. As various blockchains continue to advance at a rapid pace, the demand for querying data across multiple chains has grown. This entails the capability to conduct advanced queries like data aggregation and correlation analysis across multiple chains. Such multi-chain queries, as exemplified by the following scenarios, have the potential to yield valuable insights for both clients and businesses.

Example 1. *In the NFT realm, digital artworks can be transferred through various NFT marketplaces and blockchains. A collector can issue a multi-chain query to verify the ownership and provenance of an NFT [1]. Figure 1(a) presents a SQL query to inquire about the ownership history of the NFT with the ID '0x1af7' across two blockchains.*

Example 2. *A DeFi application can use data from various blockchains to assess the risk associated with a particular asset portfolio, optimize trading strategies, or closely monitor liquidity pools across multiple networks [2]. Figure 1(b) shows a query where a DeFi user searches the daily total value locked from two blockchains to make investment decisions.*

```
SELECT
 tx_id, owner_addr, ts
FROM(
 SELECT tx_id, owner_addr, ts
 FROM polygon
 WHERE nft_id='0x1af7'
 UNION ALL
 SELECT tx_id, owner_addr, ts
 FROM solana
 WHERE nft_id='0x1af7'
 )
ORDER BY ts
```

(a) Query 1

```
SELECT SUM(value) FROM(
 SELECT value, blk_ts
 FROM eth_transactions
 WHERE blk_ts>='2023-01-01'
  AND blk_ts<='2023-12-31'
 UNION ALL
 SELECT value, blk_ts
 FROM bsc_transactions
 WHERE blk_ts>='2023-01-01'
  AND blk_ts<='2023-12-31'
 )
GROUP BY DAY(blk_ts)
```

(b) Query 2

Fig. 1: Example Queries

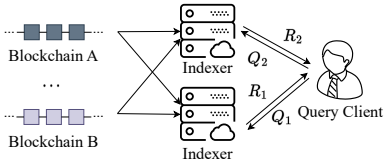

Fig. 2: System Model of TG

However, conducting multi-chain queries presents several challenges for the querying system. First, the system must be compatible with existing blockchains, without requiring any modifications to their underlying structures or protocols. This ensures smooth integration with various blockchain networks. Second, the system should support a wide range of query types to meet the diverse and evolving data analysis needs of clients. Ideally, the system should be flexible enough to integrate with a variety of existing database engines to provide comprehensive query capabilities. Third, as an integral part of the decentralized blockchain ecosystem, it should enable clients to verify the integrity of query results derived from the source chains.

One notable solution proposed to address these challenges is The Graph (TG) [15]. It is a decentralized protocol that provides clients a standardized interface for querying data from diverse blockchain networks. As illustrated in Figure 2, the protocol operates through a network of nodes called *indexers*. The indexers are incentivized to index data from multiple source chains and respond to client queries based on the indexed data. This design allows for high compatibility with existing blockchain systems. Moreover, the protocol employs the GraphQL query language to support client applications. To ensure the trustworthiness of query results provided by

TABLE I: Comparison with Existing Query Authentication Systems

Category	Representative Systems	Query Type	Blockchain Compatibility	Source Chains	Database Compatibility	Security Assumption	Instant Verification
Outsourced Database	IntegriDB [3]	Semi-SQL	N/A	N/A	✗	Cryptography	✓
	FalconDB [4]	Semi-SQL	N/A	N/A	✗	Incentive Model+Cryptography	✗
	vSQL [5]	SQL	N/A	N/A	✗	Cryptography	✓
	VeriDB [6]	SQL	N/A	N/A	✗	Auditing+TEE	✗
	SQL Ledger [7]	SQL	N/A	N/A	✗	Auditing+Trusted Storage	✗
	LedgerDB [8], GlassDB [9]	Read	N/A	N/A	✗	Auditing	✗
Blockchain Database	vChain [10], vChain+ [11]	Boolean Range	✗	Single	✗	Cryptography	✓
	GEM^2 [12]	Range	✗	Single	✗	Cryptography	✓
	[13]	Keyword	✗	Single	✗	Cryptography	✓
	LVQ [14]	Membership	✗	Single	✗	Cryptography	✓
	TG [15]	GraphQL	✓	Multiple	✗	Arbitration	✗
	Ours	Various Types	✓	Multiple	✓	TEE	✓

indexers, TG incorporates a dispute mechanism that enables clients to challenge query responses from indexers. Accepted disputes result in penalties imposed on the responsible indexers for any inaccurate or incorrect information. This mechanism incentivizes indexers to uphold the integrity of the query answers. However, the current dispute resolution process in TG has certain weaknesses. First, it is impractical for the protocol to verify every single query, so only a sample of queries is checked, which results in limited integrity guarantee for the majority of unverified query results. Second, the query dispute period can last up to seven epochs, with each epoch spanning approximately 24 hours. Furthermore, the dispute resolution process involves multiple parties, leading to additional delays and inefficiencies.

To enable strong integrity assurance, we may consider using verifiable computation (VC) techniques to authenticate the computation process involved in query evaluation. Several works have been proposed to design dedicated authenticated data structures for specific query types within a single blockchain system [10]–[14], [16], [17]. Alternatively, to support verification for arbitrary queries, general VC schemes can be employed [5], [18], [19]. The basic idea is to convert computing tasks into Boolean or arithmetic circuits that can be verified using generated cryptographic proofs.

However, general VC-based approaches suffer from high time complexity, which makes them impractical for real-world query services. For instance, processing Query #2 of the TPC-H benchmark with a scale factor of 1 using vSQL [5] takes over 30 minutes. Moreover, general VC schemes impose significant constraints, such as limiting the instruction set, prohibiting dynamic loops or dynamic memory allocation, and imposing an upper bound on the circuit size. Consequently, implementing general VC-based solutions for diverse query engines entails considerable engineering challenges and is often infeasible in real-world applications.

To fully address the need for blockchain compatibility, database compatibility, and strong integrity assurance, in this paper, we propose a novel paradigm called *verifiable virtual filesystem* (V^{2}FS). The key idea behind V^{2}FS is to shift the focus from *verifying computation* to *verifying data*. In this paradigm, to ensure query integrity, the client leverages an off-the-shelf database engine to evaluate queries using *verifiable* data obtained from an indexing service provider (ISP). Acting as a middleware between the client's query evaluation layer and the ISP's storage layer, V^{2}FS fetches data on an as-needed basis and verifies its integrity using a Merkle-based authenticated data structure. On one hand, the proof generation and verification process can be completed in logarithmic time, enabling efficient and strong integrity guarantee. On the other hand, V^{2}FS is a plug-able module that can be easily integrated with diverse database engines to support a wide range of query types. Meanwhile, to achieve blockchain compatibility, our system utilizes the DCert framework [20] to certify blocks from different blockchains, making it applicable to various blockchain systems. In addition, we propose several optimizations to enhance system performance, including two cache-based algorithms that reduce network communication overhead and a bloom filter-based enhancement that further reduces network costs.

To summarize, the contributions of this paper are as follows:

- Designing a system that utilizes verifiable data for multi-chain query authentication. To the best of our knowledge, our system is the first to achieve blockchain compatibility, database compatibility, and strong integrity guarantee simultaneously (see Table I for a comparison with existing systems).
- Proposing V^{2}FS, a novel paradigm that facilitates verifiable query processing. V^{2}FS can be integrated with various database query engines to support a wide range of query types.
- Proposing two cache-based algorithms and a bloom filter integrated algorithm to optimize query performance and reduce network communication costs.
- Conducting an extensive experimental evaluation to validate the effectiveness and efficiency of our system.

The rest of the paper is organized as follows. Section II gives some preliminaries and reviews existing systems for query verification. Section III offers the system overview. Section IV presents the detailed system design, followed by the cache-based algorithms and bloom filter integrated algorithm in Section V. Security analysis and experimental evaluation are discussed in Section VI and Section VII, respectively. Finally, we conclude the paper in Section VIII.

II. Preliminaries and Related Work

In this section, we introduce some necessary preliminaries for the proposed system. We also review some systems that support verifiable query processing and highlight the novelty of our system.

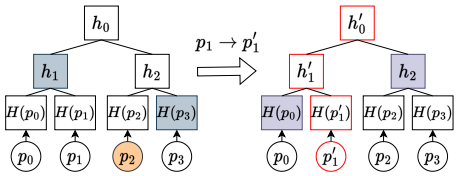

Fig. 3: Merkle Tree

A. Preliminaries

Cryptographic Hash Function: A cryptographic hash function $H(\cdot)$ is an algorithm that maps an arbitrary-length message m to a fixed-length hash digest $H(m)$. It is collision-resistant, meaning that the likelihood of a polynomial-time adversary finding two distinct messages $m_1 \neq m_2$ with the same hash digest $H(m_1) = H(m_2)$ is negligible.

Merkle Tree [21]: A Merkle tree is a tree structure used for efficient data authentication. Figure 3 shows an example of a Merkle tree containing four data objects, which is constructed in a bottom-up manner. Each leaf node stores the hash digest of an indexed data object. Each non-leaf node stores a hash digest computed from its child nodes (e.g., $h_0 = H(h_1||h_2)$, where "||" denotes the string concatenation operation). The root hash digest (i.e., h_0) is published and used to authenticate the indexed data objects. For instance, to authenticate the data object with value p_2, a Merkle proof $\{h_1, H(p_3)\}$ (depicted in shaded nodes in Figure 3) is returned. By reconstructing the root hash using the Merkle proof and p_2, one can verify the integrity of the object value p_2. If the reconstructed root hash matches the public root hash, it indicates p_2 exists in the Merkle tree and has not been tampered with. When a data object is updated, the digests of all its ancestors should be updated. To compute the updated digests of internal nodes and the root, the sibling hashes of the ancestors are used. For example, when p_1 is updated to p'_1, h'_1 and h'_0 can be computed using the sibling digests $H(p_0)$ and h_2, respectively.

Intel SGX [22]: Intel Software Guard Extensions (SGX) is an implementation of a Trusted Execution Environment (TEE) designed to protect the integrity and privacy of remote application execution, even in the presence of an untrusted host system. Within SGX, sensitive code and data are placed within an isolated memory region known as an *enclave*. This enclave is securely reserved from RAM and remains inaccessible to the external environment, including privileged system code and the operating system. This ensures that computations performed inside the enclave are executed correctly. However, transitioning in and out of the enclave can introduce performance overhead, primarily due to the usage of outside calls, or OCalls, which enable the program to access the data outside the enclave. To mitigate this performance impact, it is advisable to reduce the frequency of OCalls and minimize the potential for performance degradation.

DCert [20]: DCert is a decentralized certification framework that is compatible with any existing blockchain systems. It enables lightweight clients to validate the current state of the blockchain in constant time without the need of synchronizing the entire blockchain history. The framework employs trusted hardware (e.g., Intel SGX) to recursively certify a block by validating the integrity of the current block header, state transitions from the preceding block to the current block, and the preceding block's certificate. To check the integrity of the blockchain state, lightweight clients only need to verify and store the latest block header hdr^i and its certificate $\mathcal{C}^i_{\texttt{blk}}$. Specifically, DCert consists of the following algorithms:

- $\texttt{DCert.certify}(blk^{i-1}, \mathcal{C}^{i-1}_{\texttt{blk}}, blk^i, sk_{\texttt{DCert}}) \rightarrow \mathcal{C}^i_{\texttt{blk}}$: On input the previous block blk^{i-1}, its certificate $\mathcal{C}^{i-1}_{\texttt{blk}}$, the new block blk^i, and the private key $sk_{\texttt{DCert}}$ of the certificate issuer, it outputs blk^i's certificate $\mathcal{C}^i_{\texttt{blk}}$.
- $\texttt{DCert.valid}(\mathcal{C}^i_{\texttt{blk}}, hdr^i, pk_{\texttt{DCert}}) \rightarrow \{0, 1\}$: On input a certificate $\mathcal{C}^i_{\texttt{blk}}$, the new block header hdr^i, and the public key $pk_{\texttt{DCert}}$ of the certificate issuer, it returns 1 if and only if $\mathcal{C}^i_{\texttt{blk}}$ is valid with respect to hdr^i and $pk_{\texttt{DCert}}$.

DCert can be extended to support verifiable query processing over blockchain data by certifying additional authenticated indexes. However, its support is limited to handling one query type at a time, necessitating the design and implementation of a corresponding verifiable query processing algorithm for each new query type. Additionally, DCert lacks the capability to integrate multiple blockchains as data sources simultaneously.

Bloom Filter [23]: A bloom filter (BF) is a space-efficient probabilistic data structure used for testing set membership. It can efficiently determine if an item is not a member of a set without storing the set's actual elements. A BF utilizes an m-bit vector and k distinct hash functions to represent a large set of items efficiently and compactly. Each item is mapped to k positions in the vector using the hash functions, and the corresponding bit values are set to 1. To check the membership of a given item x in the set, x is input into the hash functions to generate k positions. If any of the positions have a value of 0, it indicates that x is not a member of the set. Otherwise, the membership of x in the set is undetermined due to the possibility of false positives.

B. Related Work

Table I shows a comparison of various systems that support verifiable query processing, including our proposed system. Broadly, these systems fall into two categories: *outsourced databases* in the cloud environment and *blockchain databases*. We now give a brief review of each of these systems.

Outsourced Databases. Several works have been proposed to support verifiable query processing in cloud-based outsourced databases [3]–[9]. IntegriDB [3] employs cryptographic set accumulators to enable verifiable queries. FalconDB [4] uses the same cryptographic tool but introduces an incentive model to improve performance. However, both of them only support a subset of SQL queries. To support general SQL queries, vSQL [5] leverages general verifiable computation cryptographic primitives. Nevertheless, due to the high complexity of the cryptographic primitives involved, vSQL suffers from extremely long, impractical proving time as mentioned in the introduction. Apart from relying on cryptographic primitives, other techniques like TEE and periodic auditing of database operations have also been proposed. VeriDB [6] provides efficient SQL query verification by leveraging TEE to execute

queries and auditing the related I/O access periodically. SQL Ledger [7] proposes using trusted storage to store the digests of all historical data for auditing purposes. LedgerDB [8] and GlassDB [9] also utilize auditing techniques but are limited to simple read-value queries. Although these techniques can, in theory, be applied in a blockchain system to support data queries, they all rely on dedicated custom query engines, with most of them targeting a single query type.

Blockchain Databases. Several studies have explored verifiable query processing over blockchain databases. vChain [10] and vChain+ [11] use cryptographic set accumulators to facilitate verifiable boolean range queries. GEM^2-tree [12], [13] address the scenario of a hybrid-storage blockchain, aiming to reduce on-chain storage costs in Ethereum. LVQ [14] focuses on Bitcoin transaction analysis and uses a bloom filter-integrated authenticated index to verify transaction membership. Despite ensuring integrity, these works are restricted regarding query types due to their specialized index design. Moreover, many of these solutions demand dedicated custom blockchain structures, making them incompatible with existing blockchain networks. Additionally, these works usually consider a single blockchain data source. As mentioned in Section I, TG [15] is a new decentralized protocol for indexing blockchain data. It utilizes indexer nodes to integrate data from multiple blockchains and offers relatively flexible query services. A dispute resolution mechanism is used to ensure the trustworthiness of query results. However, this mechanism does not provide integrity assurance for all queries and can be subject to lengthy dispute resolution delays during query verification.

In summary, existing systems lack the capability to simultaneously support all desired features, including blockchain compatibility, multi-source integration, database compatibility, strong integrity assurance, and instant verification. In contrast, our proposed system is the first to encompass all of these features simultaneously.

III. System Overview

This section provides an overview of our designed system. We start by introducing the design goals in Section III-A. Then, we present an overview of how our system achieves these design goals, followed by a discussion on the threat model in Section III-B.

A. Design Goals

We aim to achieve the following design goals for our system:

- **Blockchain compatibility**: The system should seamlessly integrate with existing blockchain networks, facilitating queries that span across multiple blockchain sources.
- **Database compatibility**: The system should be flexible enough to accommodate diverse database engines to support a wide range of query types.
- **Strong integrity guarantee**: Clients should have the ability to efficiently verify the integrity of query results, ensuring the trustworthiness of information originating from the decentralized blockchain environment.

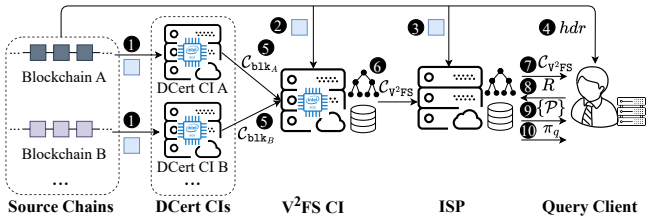

Fig. 4: System Model

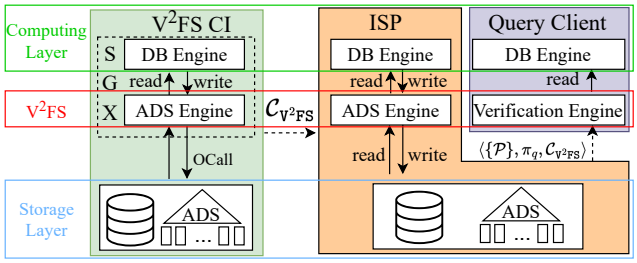

Fig. 5: System Architecture

B. Design Overview

To meet the design goals, we devise a new system as depicted in Figure 4, which consists of five types of parties:

- **Source Chains**: These are existing blockchains that serve as data sources.
- **DCert Certificate Issuers (DCert CIs)**: They are used to certify the latest states of source chains using the DCert framework [20].
- **V^{2}FS Certificate Issuers (V^{2}FS CIs)**: They are used to certify the integrity of the proposed *verifiable virtual filesystem* (V^{2}FS).
- **Indexing Service Providers (ISPs)**: Similar to the indexers in TG [15], they are responsible for indexing the data from source chains and supporting verifiable queries.
- **Query Client**: It is a lightweight node with limited storage space. The query client keeps track of the latest block headers of source chains and processes queries using the data and certificate obtained from the ISP.

We achieve our design goals through the following approaches. To ensure blockchain compatibility, we adopt the DCert framework [20], which allows for efficient blockchain certification without modifying the underlying blockchain systems. Multiple DCert CIs can be associated with each blockchain. In the event that one of the DCert CIs becomes unavailable or crashes, the remaining DCert CIs are still operational to provide blockchain certification services. Similarly, multiple V^{2}FS CIs can be deployed to ensure the resilience and robustness of our system in certifying the integrity of V^{2}FS. For ease of composition, in the following, we assume that a single DCert CI is employed for each blockchain and a single V^{2}FS CI is employed in the system.

Whenever a new block is created in a source chain, it is synchronized to the corresponding DCert CI, V^{2}FS CI, and ISP (❶, ❷, and ❸ depicted in Figure 4), while the block header is broadcasted to the query client (❹). Upon receiving a new block, the DCert CI constructs a DCert certificate ($\mathcal{C}_{\texttt{blk}}$) and transmits it to the V^{2}FS CI (❺). The DCert certificate enables the V^{2}FS CI to efficiently validate the current state of

the corresponding blockchain.

To achieve maximum database compatibility hence supporting a wide range of query types, we propose a novel solution called *verifiable virtual filesystem* (V^{2}FS). It extends the POSIX I/O interface to separate data storage from query processing, making it compatible with various database engines. The key idea of V^{2}FS is to empower the query client to leverage an *off-the-shelf* database engine to process queries using data obtained from the ISP. As illustrated in Figure 5, the query client is responsible for the computing layer, which evaluates queries using a database engine. Meanwhile, the storage layer, which maintains underlying data synchronized from the source chains, is managed by the ISP. The storage is organized and stored as regular *files*, with the query client fetching necessary *pages* of these *files* from the ISP through V^{2}FS as needed.

To establish strong integrity guarantee, we integrate a Merkle-based Authenticated Data Structure (ADS) into the storage layer of V^{2}FS. We introduce an SGX-enabled V^{2}FS CI to maintain the ADS in the form of a V^{2}FS certificate ($\mathcal{C}_{\text{V}^2\text{FS}}$). This certificate is used to validate the ADS root against the current global states across all source chains. As shown in Figure 5, the V^{2}FS CI's SGX enclave consists of a database engine and an ADS engine, while the storage layer is located outside the enclave to minimize enclave memory usage. Since the outside-enclave storage is inherently untrusted, V^{2}FS enables the enclave program to verify the data retrieved from the outside-enclave storage. When a new block is discovered in a source chain, the V^{2}FS CI updates the ADS based on the blockchain data and securely constructs a new certificate $\mathcal{C}_{\text{V}^2\text{FS}}$ inside the SGX enclave (❻). This certificate is then sent to the ISP, which will be used for verification during subsequent query processing. Similarly, the ISP utilizes an identical database engine and ADS engine to maintain its own storage upon receiving a new block (❸).[1] During query processing, the query client firstly requests the V^{2}FS certificate $\mathcal{C}_{\text{V}^2\text{FS}}$ from the ISP and verifies $\mathcal{C}_{\text{V}^2\text{FS}}$ w.r.t. the latest block headers $\{hdr\}$ observed from the blockchain networks (❼). Then, it locally evaluates the query using the data retrieved from the ISP. Specifically, whenever the database engine performs *read* operations, V^{2}FS sends a corresponding read request R to the ISP (❽). In response, the ISP provides the requested pages $\{\mathcal{P}\}$ of the filesystem (❾). At the end of query processing, the ISP sends a *verification object* (VO) consisting of a Merkle proof π_q to the query client (❿). Finally, the query client can validate the integrity of received pages using π_q and $\mathcal{C}_{\text{V}^2\text{FS}}$ to ensure the integrity of the entire query processing.

Threat Model. Without loss of generality, we assume that the integrity and availability of the underlying blockchains are guaranteed. This implies that all parties can receive up-to-date blockchain data from the networks. Furthermore, we assume the security of the underlying cryptographic primitives, such as collision-resistant hash functions and the integrity of Intel SGX [24]–[26]. On the other hand, the ISP is considered untrusted, meaning it has the potential to behave arbitrarily, e.g., returning tampered or incomplete data. Similarly, the V^{2}FS CI is also untrusted, as it may return arbitrary data to the SGX enclave or attempt to forge certificates. Finally, we assume that the query client, being read-only end users, is honest and has no intention to act against themselves. The integrity of the query results is established w.r.t. the source data in the blockchains under the following two security criteria: (i) *soundness*: all of the query results are correct and none of them are tampered with; (ii) *completeness*: no valid result is missing.

[1] If the database engine is non-deterministic, the ISP can directly synchronize the storage layer updates from the V^{2}FS CI to ensure data consistency.

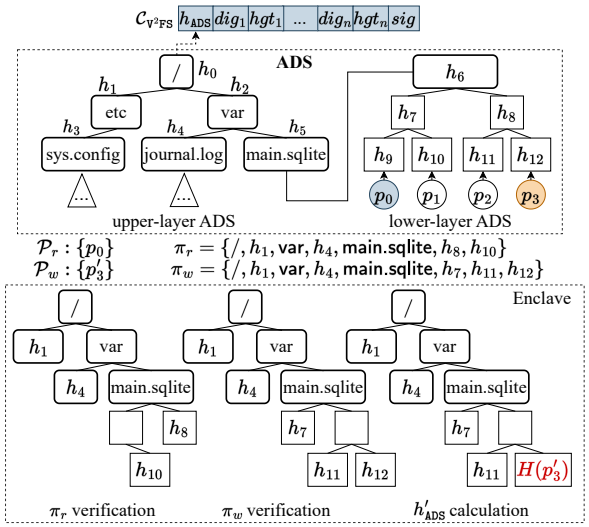

Fig. 6: Example of the V^{2}FS ADS

IV. System Design

In this section, we delve into the detailed system design, including the architecture of V^{2}FS, its maintenance mechanism, and its utilization for supporting verifiable query processing.

A. Verifiable Virtual Filesystem (V^{2}FS)

In our system, V^{2}FS is designed as a middleware to enable the utilization of off-the-shelf database engines for verifiable queries. To achieve this, V^{2}FS extends the widely-adopted POSIX I/O interface and effectively separates the data storage from query processing. This design enables smooth integration with existing database engines, thus enhancing database compatibility. Moreover, it adopts a common filesystem as its primary I/O interface, which stores raw data from source chains along with metadata such as schema and indexing information. All of these data are organized as regular files and incorporate a Merkle-based Authenticated Data Structure (ADS). In the following, we introduce the ADS of V^{2}FS, an SGX-generated certificate to attest to the integrity of V^{2}FS, and an interface for interacting with the V^{2}FS storage layer.

V^{2}FS ADS. In the traditional POSIX I/O, files are commonly organized as *pages*, which are fixed-length contiguous blocks of data. When accessing files, the database engine uses the file path and offset to locate the required pages. Therefore, we build a two-layer tree structure as the ADS to authenticate the entire filesystem. The ADS consists of a lower-layer Merkle tree built on the pages of each file and an upper-layer Merkle trie built on the file path.

Figure 6 shows an example of our designed ADS. For ease of

illustration, we assume that the storage layer contains three files and the file /var/main.sqlite has four pages. The lower-layer ADS is a complete binary tree. Each leaf node contains a digest computed using the corresponding page, while each non-leaf node contains a digest computed using its two children. For example, $h_9 = H(p_0)$ and $h_7 = H(h_9||h_{10})$. The root digests of lower-layer trees serve as the leaves of the upper-layer ADS. Each node in the upper-layer ADS contains a segment of the file path, whose digest is computed using the file path segment and its children. For example, $h_2 = H(\mathsf{var}||H(h_4||h_5))$. The root digest of the upper-layer ADS (e.g., h_0) is used to authenticate all the files in the storage layer.

To support client queries and blockchain updates simultaneously, our designed ADS utilizes multiversion concurrency control. Specifically, each update introduces new pages with a corresponding ADS tree path and a new Merkle root. To maintain query consistency, the previous version of the updated pages is retained and can be accessed through the previous Merkle root, ensuring snapshot isolation. Once the query is finished, the old pages are removed to free up storage space.

V^{2}FS Certificate. A V^{2}FS certificate $\mathcal{C}_{\mathrm{V^2FS}}$ is used to certify the integrity of the filesystem w.r.t. the global states of all source chains. Assume there are n blockchains in the system. As shown in Figure 6, $\mathcal{C}_{\mathrm{V^2FS}}$ consists of the following elements: $\langle h_{\mathrm{ADS}}, [(dig_1, hgt_1), \ldots, (dig_n, hgt_n)], sig\rangle$. Here h_{ADS} represents the root digest of the ADS; $[(dig_1, hgt_1), \ldots, (dig_n, hgt_n)]$ are the pairs of digest and height of the latest block header in each blockchain. Specifically, dig_i represents the blockchain state, while hgt_i is used to verify whether the current block conforms to the consensus protocol. Lastly, sig is a signature of the content of the certificate, signed by the SGX secret key sk_{sgx} kept inside the enclave. The client can validate $\mathcal{C}_{\mathrm{V^2FS}}$ using the SGX public key pk_{sgx} and certify the V^{2}FS integrity against the latest blockchain headers observed in the network.

V^{2}FS Interface. The V^{2}FS interface is used to interact with the V^{2}FS storage layer during blockchain state updates and query processing. To ensure compatibility with the standard POSIX I/O, it provides the following callbacks:

- `int open(const char *path)`: Given a file path, it returns a file descriptor associated with the specified file.
- `off_t seek(int fd, off_t offset)`: On input a file descriptor and an offset, it adjusts the current cursor of the file to the offset.
- `ssize_t read(int fd, void *buf, size_t count)`: Similar to the standard `read` operation, it reads data from a file into a provided buffer. In addition, the `read` operation records some auxiliary ADS information, which is used for integrity verification at a later stage.
- `ssize_t write(int fd, const void *buf, size_t count)`: Unlike the standard `write` operation that directly writes the pointed data into the specified file, our `write` operation writes data into an internal buffer, which will be processed in batch at a later stage to update the filesystem and corresponding ADS.
- `int close(int fd)`: It closes the file and releases the file descriptor.

How to use the V^{2}FS interface for V^{2}FS maintenance and query processing will be elaborated in Section IV-B and Section IV-C, respectively.

Algorithm 1: V^{2}FS Maintenance – Initialize (V^{2}FS CI)

1 **Function** `initialize()`
  `/* Enter the enclave */`
2 Fetch $\mathcal{C}_{\mathrm{V^2FS}}$, blk'_j, $\mathcal{C}'_{\mathrm{blk}_j}$,$pk_{\mathrm{DCert}_j}$;
3 $pk_{\mathrm{sgx}} \leftarrow$ `load_pk()`; `verify_sig`$(\mathcal{C}_{\mathrm{V^2FS}}, pk_{\mathrm{sgx}})$;
4 $\langle h_{\mathrm{ADS}}, [(dig_1, hgt_1), \ldots, (dig_n, hgt_n)], sig\rangle \leftarrow \mathcal{C}_{\mathrm{V^2FS}}$;
5 `DCert.valid`$(\mathcal{C}'_{\mathrm{blk}_j}, blk'_j.hdr, pk_{\mathrm{DCert}_j})$;
6 **assert** $hgt_j + 1 = blk'_j.hgt$;
7 **assert** $dig_j = blk'_j.prev\_dig$;
8 Initialize read and write page collections $\mathcal{P}_r$, $\mathcal{P}_w$;
  `/* Exit the enclave */`

B. V^{2}FS Maintenance

To facilitate verifiable queries over multi-chain data, we utilize an an off-the-shelf database engine with V^{2}FS to manage and index data received from source chains. During this process, V^{2}FS not only updates corresponding files in the filesystem but also performs ADS bookkeeping for subsequent query verification. In a decentralized environment where the V^{2}FS CI is untrusted, we employ an SGX-powered enclave to handle database updates. The database engine runs inside the protected enclave, carrying out computations based on the new block while utilizing V^{2}FS for data access through *read* and *write* operations. These operations are translated into corresponding OCalls to interact with the outside-enclave storage layer. To mitigate the performance impact caused by frequent OCalls, we introduce two page collections, namely $\mathcal{P}_r$ and $\mathcal{P}_w$, within the enclave to minimize cross-enclave operations. Once finishing the database computation, the enclave program asks the external storage layer to generate Merkle proofs for the accessed pages during computation. Additionally, the storage layer provides the corresponding Merkle paths for updating the ADS. If these Merkle proofs can be successfully verified against the previous ADS root signed by the previous $\mathcal{C}_{\mathrm{V^2FS}}$, the enclave program proceeds to compute a new ADS root based on the contents of $\mathcal{P}_w$ and the associated Merkle paths. Subsequently, a new $\mathcal{C}'_{\mathrm{V^2FS}}$ is generated for the updated database and ADS. Thanks to the blockchain's inherent transaction serialization, the database engine is relieved from managing concurrency issues.

Upon receiving a new block, the V^{2}FS CI follows three phases to maintain the V^{2}FS storage layer, its ADS, and construct a new V^{2}FS certificate $\mathcal{C}'_{\mathrm{V^2FS}}$. First, an `initialize` phase prepares all necessary data structures and executes setup procedures. Then, in the `compute` phase, a corresponding callback outlined in the V^{2}FS interface is invoked whenever the database engine accesses data in V^{2}FS. These callbacks provide standard POSIX I/O operations with additional functionalities related to integrity assurance. Finally, a `finalize` phase is invoked to perform certificate generation.

Algorithms 1 to 3 present the detailed implementation of V^{2}FS maintenance at the V^{2}FS CI. In the `initialize` phase (Algorithm 1), the enclave program first gathers necessary

Algorithm 2: V^{2}FS Maintenance – Interface (SGX)

```
1  Function open(path)
2      fd ← ocall_open(path); return fd;
3  Function seek(fd, offset)
4      fd.offset ← offset; return offset;
5  Function read(fd, buf, count)
6      readCnt ← 0;
7      while readCnt < count do
8          pid ← Calculate the page id w.r.t. fd.offset;
9          if ⟨fd.path, pid⟩ ∈ P_w or P_r then
10             page ← Retrieve the page from P_w or P_r;
11         else
12             page ← ocall_get_page(h_ADS, fd.path, pid);
13             P_r.insert(⟨fd.path, pid⟩, page);
14         Copy data from page to buf;
15         Increment readCnt, buf, and fd.offset;
16     return readCnt;
17 Function write(fd, buf, count)
18     writeCnt ← 0;
19     while writeCnt < count do
20         pid ← Calculate the page id w.r.t. fd.offset;
21         if fd.offset or buf does not align to a page then
22             if ⟨fd.path, pid⟩ ∈ P_w or P_r then
23                 page ← Retrieve the page from P_w or P_r;
24             else
25                 page ← ocall_get_page(h_ADS, fd.path, pid);
26                 P_r.insert(⟨fd.path, pid⟩, page);
27         else
28             page ← an empty page;
29         Copy data from buf to page;
30         Increment writeCnt, buf, and fd.offset;
31         P_w.insert(⟨fd.path, pid⟩, page);
32     return writeCnt;
33 Function close(fd)
34     return ocall_close(fd);
```

information to establish the current context (Line 2). This includes: (i) the previous V^{2}FS certificate $\mathcal{C}_{\text{V}^2\text{FS}}$, (ii) the new block blk'_j from the j-th blockchain, (iii) the new block's DCert certificate $\mathcal{C}'_{\text{blk}_j}$ generated by the corresponding DCert CI, and (iv) the public key of the DCert CI pk_{DCert_j}. Next, these data obtained from untrusted sources need to undergo validation to ensure their integrity, which includes the following checks: (i) the previous V^{2}FS certificate is valid against the SGX public key (Line 3); (ii) the new block is indeed signed by its corresponding DCert CI (Line 5); and (iii) the new block satisfies the chain condition in relation to its previous block embedded in the previous $\mathcal{C}_{\text{V}^2\text{FS}}$ (Lines 6 to 7). These checks are crucial for establishing the presence of a valid state transition history for the V^{2}FS ADS since the genesis block of the source chain. Finally, the two page collections $\mathcal{P}_r$ and $\mathcal{P}_w$ are initialized (Line 8).

In the `compute` phase, the database engine within the enclave processes updates from the new block to maintain the database's bookkeeping. This phase iteratively invokes the following standard POSIX I/O callbacks to access the storage layer and perform necessary operations for V^{2}FS ADS maintenance (Algorithm 2).

The `open` operation invokes an OCall to open the target file outside the enclave and returns its file descriptor. The `seek` operation updates the offset of the corresponding file. During

Algorithm 3: V^{2}FS Maintenance – Finalize (V^{2}FS CI)

```
1  Function finalize()
2      π_r ← Generate Merkle proof for pages in P_r;
3      π_w ← Generate Merkle proof for path w.r.t. pages in P_w;
       /* Enter the enclave                                   */
4      verify_merkle(π_r, P_r, h_ADS);
5      verify_merkle(π_w, P_w, h_ADS);
6      h'_ADS ← Calculate new root hash using π_w and P_w;
7      sk_sgx ← load_sk();
8      sig' ← sign(sk_sgx, H(h'_ADS||H(···||dig'_j||hgt'_j||···)));
9      C'_V2FS ← ⟨h'_ADS, [..., (dig'_j, hgt'_j), ...], sig'⟩;
       /* Exit the enclave                                    */
10     Flush P_w to storage and update ADS accordingly;
11     Broadcast C'_V2FS to ISP;
```

the `read` operation, the algorithm retrieves all the relevant pages and reads the acquired data into a designated buffer. If the data spans multiple pages, the algorithm iterates until all the data has been read. In each iteration, the page id is first calculated based on the offset of the specific file (Line 8). If the page is present in $\mathcal{P}_w$ or $\mathcal{P}_r$, it implies that the page either (i) is an updated or newly created page, or (ii) has been previously retrieved and remains unchanged. In such cases, it is directly fetched from $\mathcal{P}_w$ or $\mathcal{P}_r$ for subsequent data reading (Lines 9 to 10). Otherwise, an OCall is invoked to retrieve the page from the external storage layer, and the received page is inserted into $\mathcal{P}_r$ for future use (Lines 12 to 13). Finally, the acquired data in the page is copied to the pointed buffer with the corresponding offsets being incremented (Lines 14 to 15). In the `write` operation, the data to be written is organized as pages and inserted into $\mathcal{P}_w$. In each iteration, if the current offset of the file or the destination buffer does not align with a page boundary, the corresponding page needs to be retrieved first, using a procedure similar to the `read` operation (Lines 21 to 26); otherwise, since the data to be written covers the entire page, an empty page is used to avoid unnecessary page retrieval (Line 28). Subsequently, the data to be written is copied to the page, and the corresponding offsets are incremented (Lines 29 to 30). Finally, the page is inserted into $\mathcal{P}_w$ (Line 31), to be further processed in the `finalize` phase in batch. The `close` operation invokes an OCall to close the target file.

The `finalize` phase verifies the integrity of the accessed pages and constructs a new V^{2}FS certificate $\mathcal{C}'_{\text{V}^2\text{FS}}$ based on $\mathcal{P}_w$ (Algorithm 3). Specifically, the enclave program invokes an OCall to request two Merkle proofs: (i) π_r to authenticate the read pages in $\mathcal{P}_r$ (Line 2) and (ii) π_w to include the neighboring nodes in the Merkle path associated with the pages in $\mathcal{P}_w$ within the ADS (Line 3). The two proofs are then verified against h_{ADS} in the previous $\mathcal{C}_{\text{V}^2\text{FS}}$ (Lines 4 to 5) inside the enclave. Next, the new ADS root h'_{ADS} is computed by applying the updates in $\mathcal{P}_w$ and re-constructing the digest using π_w's tree nodes in a bottom-up fashion (Line 6). With the updated h'_{ADS}, a new certificate $\mathcal{C}'_{\text{V}^2\text{FS}}$ is signed using the SGX secret key of the V^{2}FS CI (Lines 7 to 9). Finally, the pages in $\mathcal{P}_w$ are flushed to the external storage with the ADS updated accordingly and $\mathcal{C}'_{\text{V}^2\text{FS}}$ is broadcasted to the ISP (Lines 10 to 11).

On the ISP's side, the database engine follows a similar

Algorithm 4: Query Processing (Query Client)

```
1  Function initialize()
2      Fetch {hdr_i};
3      C_V2FS ← Request certificate from the ISP;
4      pk_sgx ← load_pk(); verify_sig(C_V2FS, pk_sgx);
5      ⟨h_ADS, [(dig_1, hgt_1), ..., (dig_n, hgt_n)], sig⟩ ← C_V2FS;
6      for i in [1..n] do
7          assert dig_i = H(hdr_i);
8          assert ⟨hdr_i, hgt_i⟩ conforms to the consensus protocol;
9      digsToVerify ← [];
10 Function read(fd, buf, count)
11     readCnt ← 0;
12     while readCnt < count do
13         pid ← Calculate the page id w.r.t. fd.offset;
14         page ← access_page(h_ADS, fd.path, pid);
15         Copy data from page to buf;
16         Increment readCnt, buf, and fd.offset;
17         digsToVerify.append(H(page));
18     return readCnt;
19 Function finalize()
20     π_q ← Request VO from the ISP;
21     verify_merkle(π_q, digsToVerify, h_ADS);
```

procedure to process the new block to maintain its database's bookkeeping and update the V^{2}FS ADS accordingly. Since we do not rely on the trustworthiness of the ISP, no SGX is needed and no Merkle proof is involved.

Example. *In the example shown in Figure 6, suppose that a new block involves (i) reading p_0; (ii) writing p_3; and (iii) reading the updated p_3 (denoted as p'_3) in the file /var/main.sqlite. For simplicity, we assume that p'_3 aligns to a page boundary. In the* `initialize` *phase, the V^{2}FS CI verifies the previous V^{2}FS certificate and the new block w.r.t. its DCert certificate. Additionally, it initializes $\mathcal{P}_r$ and $\mathcal{P}_w$ as empty. During the* `compute` *phase, the enclave program first invokes an* `OCall` *to read p_0 from the external storage, followed by inserting p_0 to $\mathcal{P}_r$. Next, in the write operation, the data is written to an empty page, which is then inserted to $\mathcal{P}_w$. When reading the updated p_3, it is directly fetched from $\mathcal{P}_w$ since p'_3 exists in $\mathcal{P}_w$. After the* `compute` *phase is finished, $\mathcal{P}_r = \{p_0\}$ and $\mathcal{P}_w = \{p'_3\}$. In the* `finalize` *phase, two Merkle proofs $\pi_r = \{/, h_1, \textsf{var}, h_4, \textsf{main.sqlite}, h_8, h_{10}\}$ and $\pi_w = \{/, h_1, \textsf{var}, h_4, \textsf{main.sqlite}, h_7, h_{11}, h_{12}\}$ are generated and passed to the enclave. To verify the integrity of the accessed pages, the enclave program uses π_r and π_w to reconstruct the ADS root and compares it with h_{ADS}. Upon successful verification, the enclave program computes the new ADS root $h'_{ADS} = H(/||H(h_1||H(\textsf{var}||H(h_4||H(\textsf{main.sqlite}||H(h_7||H(h_{11}||H(p'_3))))))))$ and generates a new V^{2}FS certificate $\mathcal{C}'_{V^2FS}$ using h'_{ADS}. Following this, the pages in $\mathcal{P}_w$ are flushed to the external storage and the ADS is updated accordingly.*

C. Query Processing

As depicted in Figure 5, the query client employs the same database engine as the V^{2}FS CI and ISP for query processing. Since the storage layer is located at the ISP, the database engine retrieves pages from the ISP on demand through V^{2}FS via network communication. Moreover, the ISP provides the necessary Merkle proofs π_q as a *verification object* (VO) and the corresponding $\mathcal{C}_{\text{V}^2\text{FS}}$ for integrity validation. To reduce the communication cost, rather than generating a VO for each page access, the ISP consolidates all Merkle proofs and transmits a single VO at the end of query processing. With the VO, the query client can verify the database engine's use of data from the latest blocks in the source chains through $\mathcal{C}_{\text{V}^2\text{FS}}$, and ensure the correctness of all received pages using the Merkle proofs π_q and h_{ADS} in $\mathcal{C}_{\text{V}^2\text{FS}}$. Since the query client engages in read-only operations based on a consistent snapshot identified by the root hash in the block header, the need for transaction management is eliminated. Consequently, our system does not require additional concurrency techniques on the client side, such as those mentioned in [27], [28].

Similar to the V^{2}FS maintenance, query processing using V^{2}FS follows three phases, as illustrated in Algorithm 4. In the `initialize` phase, the query client requests and verifies the V^{2}FS certificate from the ISP. Specifically, it first fetches the latest block headers $\{hdr_i\}$ for all source chains (Line 2). Next, it requests $\mathcal{C}_{\text{V}^2\text{FS}}$ from the ISP and validates it against the SGX public key (Lines 3 to 4). To certify the integrity of the ISP's V^{2}FS w.r.t. the current consensus, the query client compares the block digest in $\mathcal{C}_{\text{V}^2\text{FS}}$ with the corresponding block header hdr_i observed in the network and checks if hdr_i complies with the blockchain consensus protocol (Lines 7 to 8). Finally, a `digsToVerify` collection is created to store the digests of all retrieved pages. During the query processing, POSIX I/O callbacks are invoked to access data from the ISP. Operations like `open`, `seek`, and `close` function in the standard manner. The `read` operation, on the other hand, requests corresponding pages from the ISP based on the accessed file path, offset, and the ADS root. Multiple iterations are performed if the requested data spans multiple pages (Lines 13 to 16). After that, the digests of these pages are stored to `digsToVerify` for later verification (Line 17). In the `finalize` phase, the query client verifies the integrity of all retrieved pages. To do so, it requests the VO from the ISP and verifies the digests stored in `digsToVerify` using the Merkle proof π_q and the ADS root h_{ADS} extracted from $\mathcal{C}_{\text{V}^2\text{FS}}$ (Lines 20 to 21).

V. Query Optimizations

Since network communication is the bottleneck during query processing, this section proposes two cache-based strategies and a bloom filter approach to enhance system performance.

A. Query Processing with Cache

Query with Intra-query Cache. During query processing, it has been observed that certain pages are accessed repeatedly within a single query. For example, in the case of query #5 from the TPC-H benchmark, 68% of the retrieved pages are accessed more than once. To address this issue, we propose the use of an intra-query cache. This cache stores recently visited pages in memory. When a page is needed, the query client can first check if it is already present in the intra-query cache. If it is found, the need to request the page from the ISP can be eliminated, leading to improved efficiency.

Query with Inter-query Cache. Since pages may be frequently accessed across multiple queries, an inter-query cache can be implemented to store commonly accessed pages. However, a key issue arises when these cached pages may become stale due to blockchain updates occurring between query executions. To tackle this problem, we propose a novel inter-query cache structure that supports efficient freshness checks. The main idea is to enable the query client to exchange information with the ISP such that the freshness of multiple pages can be confirmed by a single request. The designed inter-query cache, depicted in Figure 7, comprises multiple complete Merkle subtrees that include both the cached pages and their ancestor nodes in the ADS. Each node in the cache is associated with a freshness flag, denoting either *fresh* or *unknown* status. At the beginning of each query, all nodes in the cache are marked as *unknown*. Algorithm 5 illustrates the page accessing with the inter-query cache. During query processing, there are three possible scenarios when V^{2}FS accesses a specific page. If the requested page is not in the cache, V^{2}FS requests the page from the ISP and adds it into the cache as *fresh* (Lines 3 to 5); if the requested page exists in the cache and is marked as *fresh*, V^{2}FS simply returns the cached page to the database engine (Line 7); and finally if the requested page is present in the cache but marked as *unknown*, a query needs to be made to the ISP to determine whether the page has been altered.

To validate the freshness of a cached page, the query client sends the complete Merkle path associated with the requested page to the ISP (Lines 8 to 9). The ISP then traverses this path in a top-down manner. If a digest in the path matches its counterpart in the current ADS, it indicates that the requested page, along with other pages covered by the matching node, has not been updated since the last query. In this case, the ISP returns the location and digest of the first matching node, confirming the freshness of the entire subtree. Additionally, the ISP generates a Merkle proof for the matching node, which will be consolidated and sent to the query client in the `finalize` phase to reduce the VO size (Lines 22 to 23). Conversely, if none of the digests in the path match their corresponding nodes in the ADS, it signifies that the requested page has been updated. Consequently, the updated page is returned to the query client (Lines 24 to 26).

On the query client's side, when the ISP responds with a matching node, its digest is added to `digsToVerify`, to be verified in the `finalize` phase to ensure the integrity of the ISP's response. At the same time, the cached page is passed to the database engine for query processing (Lines 11 to 13). In case that the ISP responds with a new or updated page, V^{2}FS adds or refreshes the page in the cache, and returns it to the database engine (Lines 15 to 17). It is important to note that whenever a page is inserted into the cache or an existing node is verified as *fresh*, all its descendant nodes in the cache can be marked as *fresh* as well. Meanwhile, all ancestors and siblings of the same node are removed from the cache, resulting in the split of child nodes into separate trees. When contiguous *fresh* pages in the cache can form a complete subtree, such a tree is marked as *fresh* and its root is added to `digsToVerify` to replace all of its descendants. When the cache reaches its capacity, a common LRU strategy is employed to evict cached pages along with all their ancestors.

Algorithm 5: Access Page with Inter-query Cache

```
   /* Performed by the query client                          */
1  Function access_page(h_ADS, fPath, pid)
2    if ⟨fPath, pid⟩ ∉ cache then
3      page ← Request page from the ISP;
4      cache.insert(⟨fPath, pid⟩, page);
5      return page;
6    page ← cache.get(⟨fPath, pid⟩);
7    if page is fresh then return page ;
8    digsPath ← Digests in the path of the requested page;
9    response ← Ask the ISP to validate digsPath w.r.t. h_ADS;
10   if response is a digest then
11     cache.set_fresh(node w.r.t. response.dig);
12     digsToVerify.append(response.dig);
13     return page;
14   else
15     page ← response.page;
16     cache.insert(⟨fPath, pid⟩, page);
17     return page;
   /* Performed by the ISP                                   */
18 Function validate(h_ADS, fPath, pid, digsPath)
19   while digsPath.is_not_empty() do
20     dig ← digsPath.pop();
21     if dig matches the corresponding node in the ADS then
22       π ← gen_proof(dig); Merge π to π_q;
23       return dig;
24   page ← get_page(h_ADS, fPath, pid);
25   π ← gen_proof(H(page)); Merge π to π_q;
26   return page;
```

Example. *Consider the example in Figure 7. Assume the current cache contains six pages (p_0 to p_5) and the upcoming query requests pages p_0, p_4, and p_5. Before the next query is processed, all cached pages are initially marked as unknown.*

- ❶ *When requesting p_0, the query client sends the path $\{h_0, h_1, h_5\}$ to the ISP for validation. Suppose that the ISP responds that h_1 matches the corresponding node in the ADS. The query client then updates all the nodes covered by h_1 as fresh and retrieves p_0 from the cache to the database engine. Meanwhile, the query client adds h_1 to `digsToVerify` for verification at a later stage as well as removes h_0 and h_2 from the cache.*
- ❷ *Next, to access p_4, the query client sends the path $\{h_3, h_9\}$ to the ISP for validation. Suppose that p_4 had been updated to p'_4. The ISP returns the updated p'_4 to the query client. The query client then replaces p_4 by p'_4 and removes h_3. Meanwhile, $h'_9 = H(p'_4)$ is added to `digsToVerify`.*
- ❸ *Then, to access p_5, the query client sends h_{10} to the ISP. Suppose that p_5 is unchanged, the query client can build a new subtree consisting of $\{h'_9, h_{10}, h'_3\}$. Additionally, h'_3 is added to `digsToVerify` to replace h'_9.*
- ❹ *At the end, the ISP returns a Merkle proof π_q, which contains $\{h'_2, h_4\}$, where h_4 is h_3's sibling digest, to attest the nodes in `digsToVerify`. On the client-side, h_{ADS} can be reconstructed and verified using $\{h_1, h'_3\}$ and π_q.*

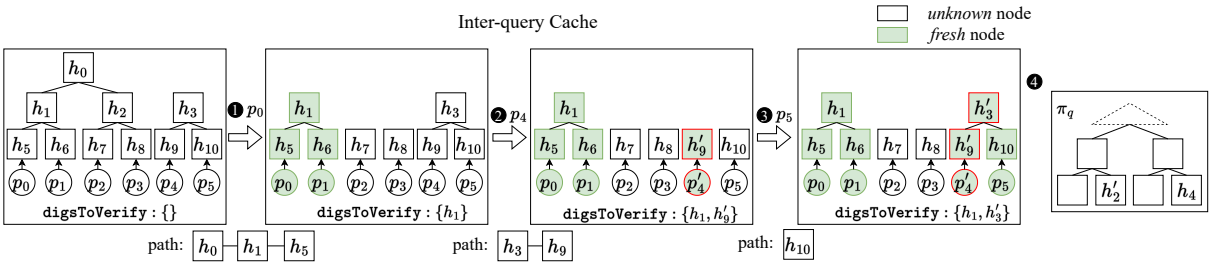

Fig. 7: Queries with Inter-query Cache

B. Bloom Filter Integrated Freshness Checking

To further reduce network communication costs, we propose using page update information to check the freshness of cached pages. To achieve this, we introduce the concept of a versioned bloom filter (VBF) and design a VBF-integrated algorithm for freshness checking. The VBF, which summarizes the historical update information of all pages, is managed by the V^{2}FS CI inside the SGX enclave. After processing each new block, the enclave constructs a V^{2}FS certificate with two additional fields: (i) a monotonically increased version number $v_{\mathcal{C}_{\text{V}^2\text{FS}}}$; and (ii) a constant-sized VBF that encodes the page update history w.r.t. the version number. Specifically, during the V^{2}FS maintenance, whenever a page indexed by $\langle \texttt{fPath}, \texttt{pid} \rangle$ is written, the index key is added to the VBF by setting the corresponding slots with the current version number $v_{\mathcal{C}_{\text{V}^2\text{FS}}}$.

To utilize the VBF, each leaf node in the query client's cache is augmented with two extra fields: (i) V_n, denoting the most recent $\mathcal{C}_{\text{V}^2\text{FS}}$ version number when the corresponding page is marked as *fresh*, and (ii) S_n, representing the set of slot indexes in the VBF w.r.t. the corresponding page's index key $\langle \texttt{fPath}, \texttt{pid} \rangle$. During query processing, if V^{2}FS identifies a required page in the cache that is marked as *unknown*, the VBF is used as the first step to check its freshness. The query client first retrieves the leaf node n associated with the accessed page in the cache. Then, it compares the version number V_n with the values stored at the corresponding slots of S_n in the VBF. If none of these values are greater than V_n (i.e., $V_n \geq \text{VBF}[pos],\ \forall\ pos \in S_n$), it indicates that the page in the cache is *fresh*, as it has not been updated since version V_n. Consequently, the query client marks the page as *fresh* and fetches it directly from the cache for query processing. Note that since the VBF is part of $\mathcal{C}_{\text{V}^2\text{FS}}$ and can be verified by the SGX public key, there is no need to add the corresponding leaf node to `digsToVerify`. However, if any of the values is greater than V_n, the VBF cannot guarantee the freshness of the target page in the cache due to potential false positives of the bloom filter. In this case, the query client falls back to the freshness validation algorithm introduced in Algorithm 5.

VI. Security Analysis

This section analyzes the security of the V^{2}FS-based query processing algorithms.

Theorem 1. *The V^{2}FS certificate construction algorithm proposed in our system is secure, if the underlying cryptographic primitives, the DCert, and the trusted hardware are secure.*

Proof. We prove this theorem by contradiction. If an adversary can forge a V^{2}FS certificate, it means either (i) the adversary can persuade the database engine in the enclave of the V^{2}FS CI to accept tampered pages, or (ii) the V^{2}FS certificate is built on a blockchain state that is not accepted by the current network. The former case is impossible since the V^{2}FS CI requires sufficient Merkle proofs to authenticate all accessed pages during database maintenance. On the other hand, the client will check the blockchain headers embedded in $\mathcal{C}_{\text{V}^2\text{FS}}$ against the current consensus in the network, which makes the latter case impossible. □

Theorem 2. *The VBF integrated freshness checking algorithm proposed in our system does not have a false-negative error.*

Proof. We also prove this theorem by contradiction. Assume that the VBF integrated freshness checking algorithm produces false-negative errors. This implies that there exists a leaf node n in the cache where $V_n = v_1$ and $S_n = \{pos_1, \ldots, pos_m\}$ satisfying the condition $v_1 \geq \text{VBF}[pos_i]$ for any $i = 1, \ldots, m$ Additionally, the page p_n associated with n has been updated with v_2 where $v_2 > v_1$. However, when the page is updated at v_2, the corresponding position (S_n) in the VBF should be set to v_2. This leads to a contradiction that all the values of the corresponding slots in S_n are less than v_1. □

Theorem 3. *The verifiable query processing algorithms proposed in our system are secure, if the underlying cryptographic primitives, the DCert, and the trusted hardware are secure.*

Proof. Since our system uses an off-the-shelf database engine to process the queries, a tampered or incomplete query result indicates that there are incorrect pages accessed by the database engine. We show this cannot happen for all three proposed algorithms by contradiction.

In Algorithm 4, all of the accessed pages are authenticated based on a Merkle proof w.r.t. $\mathcal{C}_{\text{V}^2\text{FS}}$. Therefore, incorrect page access can only happen when the corresponding Merkle proof is forged to convince a tampered or missing page or the adversary persuades the client with a forged $\mathcal{C}_{\text{V}^2\text{FS}}$. A successfully forged Merkle proof indicates that there exist two versions of the Merkle tree yielding the same root hash h_{ADS}. This means a successful hash collision in the underlying cryptographic hash function, which contradicts our assumption. On the other hand, $\mathcal{C}_{\text{V}^2\text{FS}}$ cannot be forged as proved in Theorem 1.

Similarly, when the cache is used during query processing in Algorithm 5, all the cache freshness checks are authenticated as part of Merkle proofs. As such, an incorrect freshness response

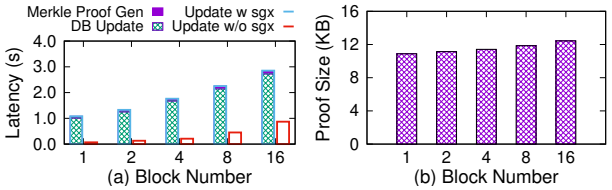

Fig. 8: V^{2}FS CI Database Update Performance

by the ISP can lead to the hash collision to the underlying cryptographic hash function.

Finally, the VBF integrated query processing algorithm is secure because the VBF integrated freshness checking algorithm guarantees zero false-negative errors as proved in Theorem 2. At the same time, the adversary cannot forge the VBF, which is authenticated by $\mathcal{C}_{\mathrm{V^2FS}}$. □

VII. Performance Evaluation

This section first covers the system implementation and experiment setup, including the dataset, workloads, and evaluation metrics. It then presents the system evaluation results.

A. System Implementation and Experiment Setup

Implementation. We have implemented our proposed system in Rust programming language using the SQLite database [29]. The database engine in our system is implemented using the Rusqlite library [30]. The V^{2}FS ADS employs BLAKE2b [31] as the cryptographic hash function and RocksDB [32] as the underlying storage. Besides, the SGX enclave is implemented using the Apache Teaclave SGX SDK [33]. The page size is set to 4KB, following the default setting in SQLite. The cache size is set to 1GB by default, as modern devices typically have sufficient memory. We set the VBF with 100,000 slots and five hash functions to achieve a false-positive probability of less than 1%. The DCert CI is based on the implementation in [20]. The V^{2}FS CI is deployed on a machine with an SGX-enabled Intel Xeon Gold 6330 CPU and 64GB Enclave Page Cache, while the ISP and query client are deployed on machines with Intel i7-7567U CPU and 32GB RAM. The network bandwidth between the ISP and the query client is 1Gbps.

Dataset. We evaluate the query performance of our system using a dataset extracted from the Ethereum [34] and Bitcoin [35] blockchains using Blockchain ETL [36]. The dataset covers the period from May 12, 2023, to May 18, 2023, comprising 16 tables with over 70 million records related to Bitcoin and Ethereum.

Workloads and Evaluation Metrics. We test the queries provided in the Awesome BigQuery Views project, which includes eight types of SQL queries for on-chain data analysis [37]. These queries encompass a broad range of relational operations such as selection, projection, order, aggregation, join, and union. The operations involved in each query are summarized in our technical report [38]. A total of nine workloads are generated for query evaluation. For each of the eight SQL queries, a workload is developed by randomly generating 20 unique queries of the same type. These queries follow a Zipfian distribution in terms of query time window to model a real-world usage pattern. Additionally, a mixed workload (denoted as "Mixed") is generated to simulate a more complex and varied query environment. It contains 40 randomly generated queries, with five from each query type. The evaluation metrics include (i) query latency, covering query execution, data transmission, and verification time, (ii) number of network requests issued by the query client for page transmission and freshness checking, and (iii) the VO size.

B. Experimental Results

Database Update Cost. Figure 8 shows the database update performance with and without SGX in the V^{2}FS CI. We vary the number of blocks inserted into the database to measure the block processing time and the size of Merkle proofs generated by the enclave. As the number of input blocks increases, the database update time also increases. The enclave introduces a performance degradation ranging from $3.2\times$ to $10.4\times$, as expected, due to the costly OCalls needed to interact with the outside-enclave storage layer. Processing multiple blocks in batches mitigates the performance degradation. This is because the page collections introduced in Section IV-B can effectively reduce the number of OCalls. The Merkle proof generation time constitutes a small proportion (around 6%) of the entire update time. With more input blocks, the Merkle proof size slightly increases from 10.8KB to 12.5KB.

Query Performance. The query performance of our system is shown in Figures 9 to 11, with a varying query time window from 3 to 48 hours. Four methods are compared: (i) *Baseline*: no optimization applied; (ii) *Intra*: utilizing an intra-query cache; (iii) *Inter*: employing an inter-query cache; and (iv) *Inter+Vbf*: incorporating an inter-query cache with the versioned bloom filter. Due to space constraints, we present results for three representative workloads: (i) Q1, comprising aggregate queries; (ii) Q2, containing linear scan queries; and (iii) Q6, involving nested queries. Additionally, we include the mixed workload in the experiment. We break down the query latency into (i) *exec*, computation conducted by the query client, and (ii) *net*, network transmission between the ISP and the query client. For *Inter* and *Inter+Vbf* methods, we further divide the network requests into (i) *check*, network requests for freshness checking, and (ii) *page*, network requests for page retrievals.

We make several interesting observations. First, as shown in Figure 9, compared with *Baseline*, *Inter* and *Inter+Vbf* improve the query performance by up to $4.1\times$ and $6.1\times$, respectively. The improvement is achieved by reducing page transmissions through the inter-query cache. This can be validated from Figure 10, where up to 88.9% of transmitted pages are reduced by *Inter* and *Inter+Vbf*. The additional performance boost in *Inter+Vbf* comes from the further reduced network requests during freshness checking. As shown in Figure 10, the utilization of VBF effectively reduces 99.7% of network requests for freshness checking, resulting in up to 45.6% improvement in query performance compared to the *Inter* method. Second, *Intra* improves query efficiency by up to $2.8\times$, reducing transmitted pages by up to 68.5%, except for Q1 and Q2. The limited effectiveness in Q1 and Q2 is due to the fact that these queries rarely access the same page repeatedly

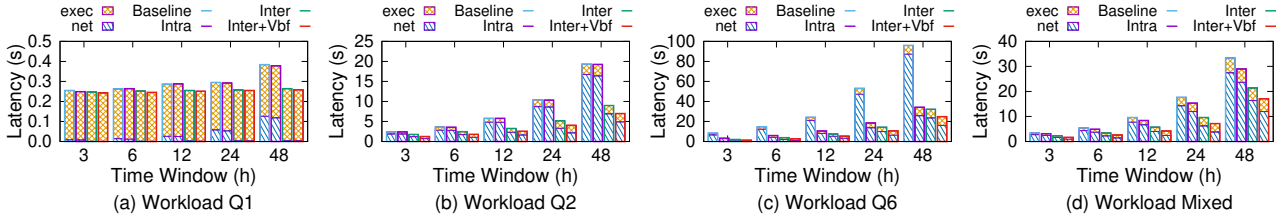

Fig. 9: Query Latency

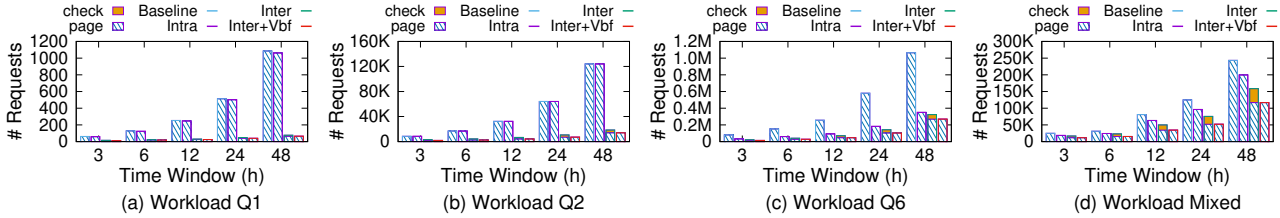

Fig. 10: # Network Requests

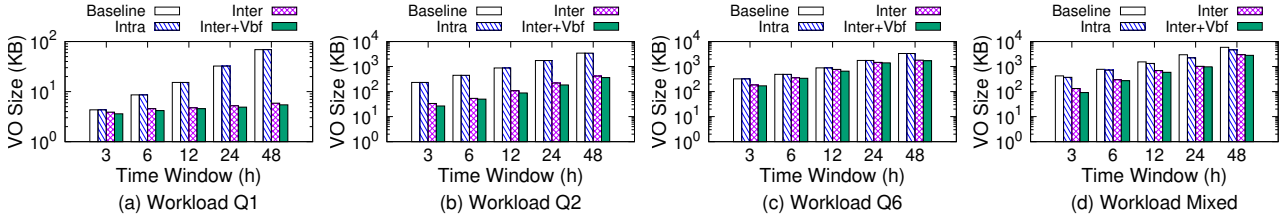

Fig. 11: VO Size

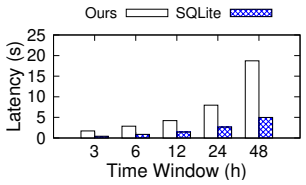

Fig. 12: Comparison with SQLite (Mixed Workload)

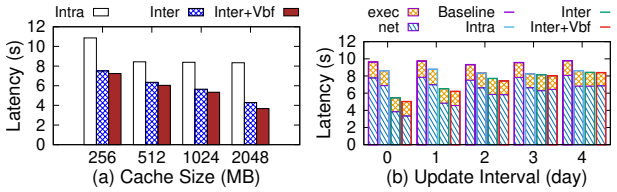

Fig. 13: Impact of Cache Size and Database Update

in a single query, causing the intra-query cache to be less effective in reducing page transmissions. Third, the network transmission time dominates the query latency, except for Q1. Queries in Q1 require only a few pages transmitted by network, constituting up to 9.1% of the entire query latency. For other queries, the network transmission takes up around 82.8% of the query latency. Lastly, Figure 11 shows that while the query time window extension results in an increase in VO size, it remains within a reasonable range and does not require significant bandwidth resources compared to page transmissions.

Figure 12 compares our system with the ordinary SQLite, which doesn't offer integrity guarantee. On the Mixed workload, our system is 2.9× to 3.9× slower than SQLite. We believe that this discrepancy in performance is acceptable owing to the additional integrity guarantee offered by our system.

Impact of Cache Size and Database Update. We next evaluate the impact of cache size and database update on query performance. For cache size, we vary it from 256MB to 2GB and analyze the query performance of the Mixed workload in Figure 13(a). Initially, *Intra* shows improved query time with larger cache sizes. However, its performance remains stable after reaching a cache size of 512MB. This is because the intra-query cache has become sufficiently large to accommodate all the pages in a single query. On the other hand, *Inter* and *Inter+Vbf* continue to benefit from larger cache sizes since they can cache pages across queries.

Regarding the impact of database update, we measure the query performance on the Mixed workload while varying the amount of updated data. As shown in Figure 13(b), *Baseline* and *Intra* perform consistently across different update intervals. However, with more data updated to the database, the effectiveness of *Inter* and *Inter+Vbf* in reducing page transmissions decreases. The reason is twofold. Firstly, inserting new data may cause cached pages to become stale, thereby reducing the efficiency of freshness checking. Secondly, new pages created by data updates will be requested, which further limits the chance of visiting a cached page. Nevertheless, *Inter* and *Inter+Vbf* still outperform *Baseline* and *Intra*.

VIII. Conclusion

In this paper, we present a pioneering system that enables verifiable multi-chain queries and achieves blockchain compatibility, database compatibility, and strong integrity guarantee simultaneously. In particular, we propose V^{2}FS, a novel virtual filesystem that enables easy integration with diverse database engines to support various verifiable queries over multi-chain data. Two cache optimizations and a bloom filter-integrated algorithm are also proposed to enhance query efficiency. Extensive security analysis and empirical studies validate the effectiveness and efficiency of our proposed system.

Acknowledgement This work is supported by CCF-Ant Research Fund (Project No. RF20210014) and Hong Kong RGC Grants (Project No. C2004-21GF, 12200022, 12202221, 12201520). Jianliang Xu is the corresponding author.

REFERENCES

[1] N. Voices. (2023) The graph (grt) and its use cases in gaming and nfts. [Online]. Available: https://nerdbot.com/2023/07/07/the-graph-grt-and-its-use-cases-in-gaming-and-nfts/

[2] T. Surve. (2023) What is the graph, and how does it work? [Online]. Available: https://cointelegraph.com/explained/what-is-the-graph-and-how-does-it-work

[3] Y. Zhang, J. Katz, and C. Papamanthou, "IntegriDB: Verifiable sql for outsourced databases," in *ACM CCS*, 2015, pp. 1480–1491.

[4] Y. Peng, M. Du, F. Li, R. Cheng, and D. Song, "FalconDB: Blockchain-based collaborative database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 637–652.

[5] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vSQL: Verifying arbitrary sql queries over dynamic outsourced databases," in *IEEE S&P*, 2017, pp. 863–880.

[6] W. Zhou, Y. Cai, Y. Peng, S. Wang, K. Ma, and F. Li, "VeriDB: An sgx-based verifiable database," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2182–2194.

[7] P. Antonopoulos, R. Kaushik, H. Kodavalla, S. Rosales Aceves, R. Wong, J. Anderson, and J. Szymaszek, "SQL Ledger: Cryptographically verifiable data in azure sql database," in *Proceedings of the 2021 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2021, p. 2437–2449. [Online]. Available: https://doi.org/10.1145/3448016.3457558

[8] X. Yang, Y. Zhang, S. Wang, B. Yu, F. Li, Y. Li, and W. Yan, "LedgerDB: a centralized ledger database for universal audit and verification," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3138–3151, 2020.

[9] C. Yue, T. T. A. Dinh, Z. Xie, M. Zhang, G. Chen, B. C. Ooi, and X. Xiao, "GlassDB: An efficient verifiable ledger database system through transparency," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, p. 1359–1371, 2023.

[10] C. Xu, C. Zhang, and J. Xu, "vChain: Enabling verifiable boolean range queries over blockchain databases," in *ACM SIGMOD*, 2019, pp. 141–158.

[11] H. Wang, C. Xu, C. Zhang, J. Xu, Z. Peng, and J. Pei, "vChain+: Optimizing verifiable blockchain boolean range queries," in *Proceedings of the 38th IEEE International Conference on Data Engineering*, Kuala Lumpur, Malaysia, May 2022, pp. 1928–1941.

[12] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi, "GEM^2-Tree: A gas-efficient structure for authenticated range queries in blockchain," in *IEEE ICDE*, 2019, pp. 842–853.

[13] C. Zhang, C. Xu, H. Wang, J. Xu, and B. Choi, "Authenticated keyword search in scalable hybrid-storage blockchains," in *IEEE ICDE*, 2021, pp. 996–1007.

[14] X. Dai, J. Xiao, W. Yang, C. Wang, J. Chang, R. Han, and H. Jin, "LVQ: A lightweight verifiable query approach for transaction history in bitcoin," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 1020–1030.

[15] "The Graph," https://thegraph.com, accessed: 2022-12-30.

[16] Z. Peng, H. Wu, B. Xiao, and S. Guo, "VQL: Providing query efficiency and data authenticity in blockchain systems," in *2019 IEEE 35th international conference on data engineering workshops (ICDEW)*. IEEE, 2019, pp. 1–6.

[17] H. Wu, Z. Peng, S. Guo, Y. Yang, and B. Xiao, "VQL: Efficient and verifiable cloud query services for blockchain systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1393–1406, 2021.

[18] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE S&P*, 2013, pp. 238–252.

[19] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 781–796.

[20] Y. Ji, C. Xu, C. Zhang, and J. Xu, "DCert: Towards secure, efficient and versatile blockchain light clients," in *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, Quebec, QC, Canada, Nov. 2022.

[21] R. C. Merkle, "A certified digital signature," in *Proc. CRYPTO*, 1990, pp. 218–238.

[22] V. Costan and S. Devadas, "Intel SGX explained," *Cryptology ePrint Archive, Paper 2016/086*, pp. 1–118, 2016.

[23] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[24] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels, "Tesseract: Real-time cryptocurrency exchange using trusted hardware," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2019, p. 1521–1538.

[25] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 185–200.

[26] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2019, p. 123–140.

[27] Z. Lai, C. Liu, and E. Lo, "When private blockchain meets deterministic database," *Proc. ACM Manag. Data*, vol. 1, no. 1, may. [Online]. Available: https://doi.org/10.1145/3588952

[28] C. Xu, C. Zhang, J. Xu, and J. Pei, "SlimChain: scaling blockchain transactions through off-chain storage and parallel processing," *Proceedings of the VLDB Endowment*, pp. 2314–2326, 2021.

[29] D. R. Hipp, "SQLite," https://www.sqlite.org/changes.html, 2022.

[30] "Rusqlite," https://github.com/rusqlite/rusqlite, accessed: 2022-12-30.

[31] "Blake2b," https://github.com/oconnor663/blake2_simd, accessed: 2022-12-30.

[32] "RocksDB," https://rocksdb.org, accessed: 2022-12-30.

[33] "Apache teaclave sgx sdk," https://github.com/apache/incubator-teaclave-sgx-sdk, accessed: 2022-12-30.

[34] G. Wood. (2014) Ethereum: A secure decentralised generalised transaction ledger. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[35] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[36] "Blockchain ETL," https://github.com/blockchain-etl, accessed: 2022-12-30.

[37] "Awesome bigquery view," https://github.com/blockchain-etl/awesome-bigquery-views, accessed: 2022-12-30.

[38] H. Wang, C. Xu, C. Zhang, J. Xu, X. Chen, Y. Yan, S. Tian, and H. Hu. (2023) V^{2}FS: A verifiable virtual filesystem for multi-chain query authentication (technical report). [Online]. Available: https://www.comp.hkbu.edu.hk/~db/v2fs.pdf