VERIFIABLE DATA SEARCH ATOP BLOCKCHAIN

HAOTIAN WU

PhD

The Hong Kong Polytechnic University

2023

The Hong Kong Polytechnic University
Department of Computing

Verifiable Data Search atop Blockchain

Haotian Wu

A thesis submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
July, 2022

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____ (Signed)

Haotian Wu (Name of student)

Abstract

As an emerging decentralized technology, blockchain has become a compelling paradigm for trusted data storage owing to the underlying techniques of hashing chain and consensus schemes. In addition to the on-chain data itself, blockchain can also be utilized to store key data outsourced from data owners via smart contract. It can be seen as an important complement and enhancement to existing cloud storage. However, untrusted clouds necessitate the verifiable data search atop the blockchain.

Apart from blockchain data, raw data or encrypted data can also be outsourced to the cloud. Therefore, in this thesis, we investigate three types of data, i.e., native blockchain data, outsourced raw data and outsourced encrypted data, in the scenarios containing both clouds and blockchain. For the native blockchain data, we employ clouds to provide efficient query services on the underlying data and design a Verifiable Query Layer (VQL) to make the query verifiable. In terms of outsourced data, we let clouds store the data and host query services over it. The blockchain will store some metadata via the smart contract and facilitate the query verification. For the outsourced raw data, we focus a complicated data structure, i.e., graph data, and enable privacy-preserving verifiable query by designing a novel authenticated data structure (ADS) named PAGB. To handle outsourced encrypted data, we propose a novel verifiable searchable symmetric encryption (SSE) scheme called Slicer to support

range search on numerical data. The effectiveness and practicality of our designs are demonstrated by theoretical analysis and extensive evaluations respectively.

Publications

Journal Articles

1. **Haotian Wu**, Zecheng Li, Rui Song, and Bin Xiao. Enabling Privacy-Preserving and Efficient Authenticated Graph Queries on Blockchain-Assisted Clouds, accepted in *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2023.

2. **Haotian Wu**, Zhe Peng, Songtao Guo, Yuanyuan Yang, and Bin Xiao. VQL: Efficient and Verifiable Cloud Query Services for Blockchain Systems, published in *IEEE Transactions on Parallel and Distributed Systems*, Jun. 2022.

3. Zecheng Li, **Haotian Wu**, Lap Hou Lao, Songtao Guo, Yuanyuan Yang, and Bin Xiao. Pistis: Issuing Trusted and Authorized Certificates With Distributed Ledger and TEE, published in *IEEE Transactions on Parallel and Distributed Systems*, Jul. 2022.

Conference Papers

1. **Haotian Wu**, Rui Song, Kai Lei, and Bin Xiao. Slicer: Verifiable, Secure and Fair Search over Encrypted Numerical Data Using Blockchain, in *Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS)*, Bologna, Italy, 10-13 Jul. 2022.

2. **Haotian Wu**, Jun Tao, and Bin Xiao. Towards a Stable and Truthful Incentive Mechanism for Task Delegation in Hierarchical Crowdsensing, in *Proc. of the IEEE International Conference on Communications (ICC)*, Dublin, Ireland, 7-11 Jun. 2020.

3. Zhe Peng, **Haotian Wu**, Bin Xiao, and Songtao Guo. VQL: Providing Query Efficiency and Data Authenticity in Blockchain Systems, in *Proc. of the IEEE International Conference on Data Engineering Workshops (ICDEW)*, Macao, China, 8-12 Apr. 2019.

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Bin Xiao, for inspiring me to conduct innovative research and providing me with tremendous ideas. His patience and valuable suggestions have given me abundant confidence in tackling difficult research problems, and his appreciation and acknowledgement of my work motivate me to persevere in my Ph.D. study.

I would also like to thank our previous and current group members (Dr. Zhe Peng, Dr. Shang Gao, Zecheng Li, Yu Zhou, Shengyuan Chen, LapHou Lao, Fan Liu, Xuelong Dai, Kaisheng Liang, Zhonghao Liu, Rui Song, Xinwei Du, Xiaohai Dai, and Hao Xu) for helping, directly or indirectly, my Ph.D. study. I am also grateful to my friends in Hong Kong for their company and encouragement. I especially would like to thank Shuzhen Zhang, Jie Xiong, Xindong Zhang, Ningning Hou, Kaiyan Cui, and Jie Zhang for their emotional support and selfless help.

I would like to dedicate this thesis to all people who have guided, instructed, or helped me along the way. My Ph.D. study cannot be accomplished without all of your kind support. The most importantly, I would like to thank my parents for their unconditional love and support. The thesis is dedicated to them all.

Hong Kong S.A.R., China Haotian Wu

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Data Storage on the Blockchain

Blockchain is a distributed ledger that is able to reliably record all transactions in a decentralized network. As depicted in Figure 1.1, a typical blockchain usually comprises a series of blocks that are chained in order by referring to the preceding block. A block mainly consists of a block header storing the attributes of the block like the timestamp and the hash value of its predecessor, and a block body, which contains the corresponding list of transaction data in the block. In the blockchain network, each full node keeps a copy of the ledger, the consistency of which is guaranteed by adopting various consensus algorithms. Smart contract is an advanced function offered by Ethereum through Ethereum Virtual Machine (EVM). It is a special type of address with trusted program that digitally performs a contract without third parties. The execution result of a smart contract is ensured to be correct since the program code is deterministic and can be executed by all miners. Like the account address, the contract address also contributes to the block header via the state root, which is the root of Merkle Patricia Tree (MPT) for all addresses. However, the difference is that the contract address contains a code hash for its program code, and a storage

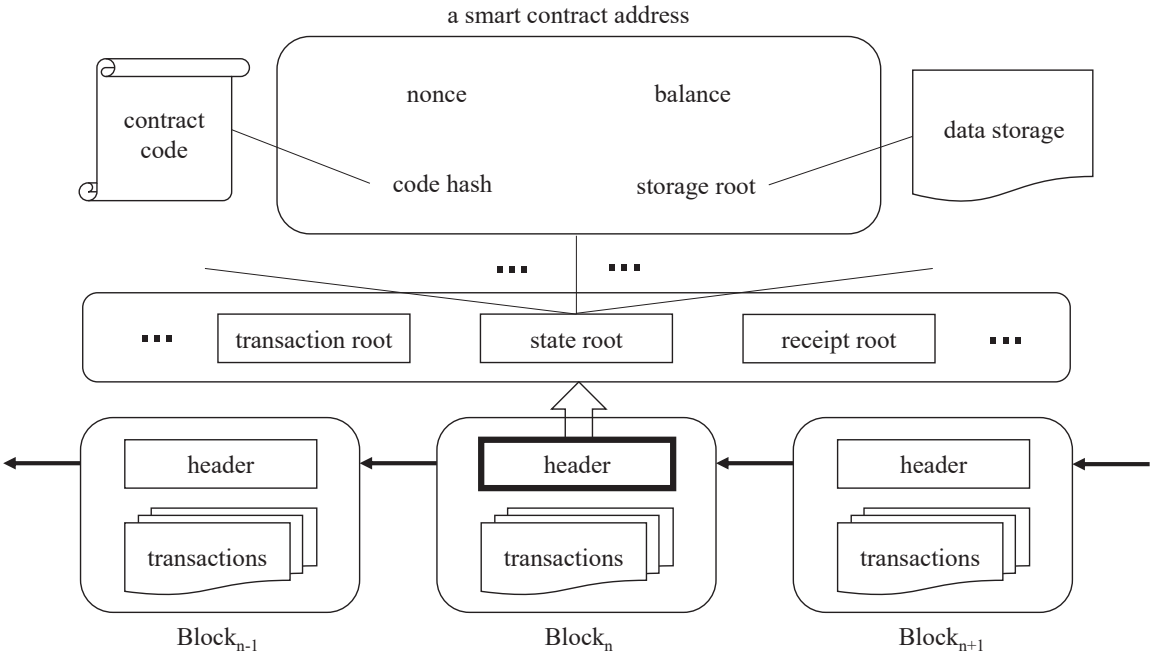


Fig. 1.1: Data storage in the smart contract on Ethereum.

root of the MPT on the persistent data storage. Besides the transaction data in the block body, the blockchain can also store the data outsourced by users into the data storage of a smart contract. Thus, the data storage on the blockchain opens the door for verifiable search over these data, including the native blockchain data and the outsourced data.

1.2 Problem Statement

Although the blockchain technology can offer a new approach to data storage, there still exists concern about the integrity of search results since the servers hosting search services may be untrusted. Therefore, this thesis focuses on the verifiable search problem over these data atop the blockchain. According to the storage field and storage form, we investigate the problem over three types of data, i.e., native

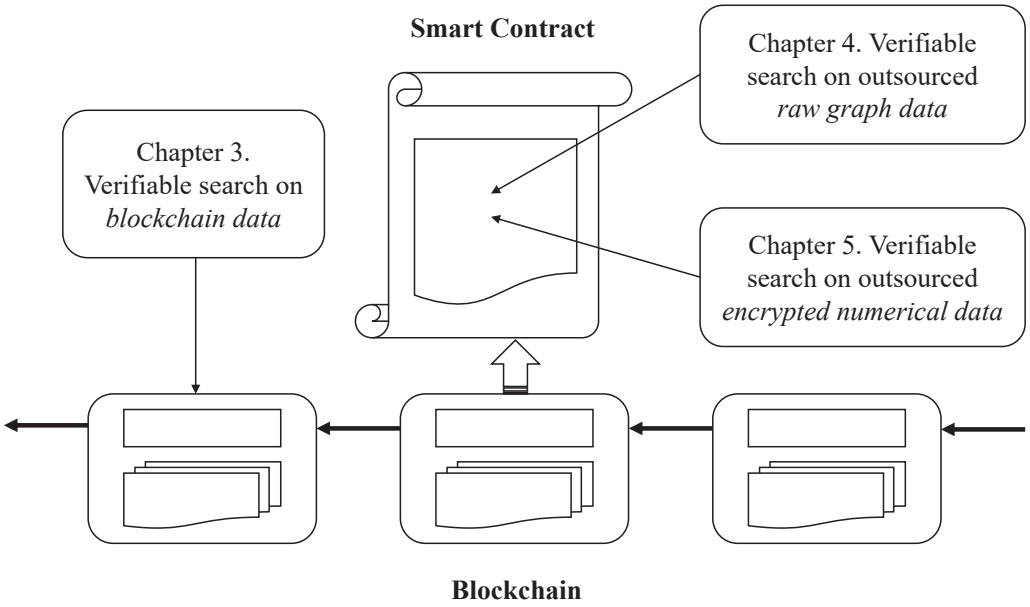


Fig. 1.2: Thesis structure.

blockchain data, outsourced raw data and encrypted data on the smart contract. We present the relationships between these three problems in Figure 1.2.

- **Native blockchain data.** Searching over the blockchain data is usually inefficient because its underlying database is not suitable for random reading. Some designs choose to employ clouds to store the blockchain data and host query services, but the consistency between the search results and the real blockchain data cannot be ensured. Hence, the research problem is how to provide efficient and verifiable query services over the native blockchain data.

- **Outsourced raw data.** For the outsourced raw data, we choose a popular but complicated data type, i.e., property graph, to demonstrate the search problem. It is challenging to guarantee both the soundness and completeness of query

results when realizing the verifiable search over graph data. In addition, we desire to guarantee that the search process will not leak any irrelevant information about the data. Therefore, the second problem becomes how to provide efficient and privacy-preserving authenticated graph query services using blockchain.

- **Outsourced encrypted data.** For the outsourced encrypted data, we step over from the primary keyword-file search to the range search over the numerical data. Moreover, since the data is encrypted, we require the public verification of search results so that the data user cannot maliciously deny the correct results returned from clouds. Thus, the third topic is how to enable public verification for encrypted numerical search by leveraging the blockchain.

1.3 Thesis Contributions

In this thesis, we propose novel methods to realize the verifiable search based on the trusted environment provided by the blockchain. We focus on the three aforementioned problems as detailed as follows.

1.3.1 Efficient and Verifiable Cloud Query Services on Blockchain Data

Despite increasingly emerging applications, a primary concern for blockchain to be fully practical is the inefficiency of data query. Direct queries on the blockchain take too much time by searching every block, while indirect queries on a blockchain database greatly degrade the authenticity of query results. To conquer this authenticity problem, in Chapter 3, we propose a Verifiable Query Layer (VQL) that can be deployed in the cloud to provide both efficient and verifiable data query services

for blockchain systems. The middleware layer extracts data from the underlying blockchain system and efficiently reorganizes them in databases. The database fingerprint will be first verified by miners and then written into the blockchain. Moreover, public users can verify the entire databases or several databases that interest them in the middleware layer. We implement VQL together with the verification schemes and conduct extensive experiments based on a practical blockchain system. The evaluation results demonstrate that VQL can efficiently support various data query services and guarantee the authenticity of query results for blockchain systems.

1.3.2 Privacy-Preserving and Efficient Authenticated Queries on Outsourced Graph Data

Prior research has introduced a new scenario of blockchain-assisted clouds where the data owner outsources original data to cloud servers and stores some metadata on the blockchain for verification. Despite research on some primary query types like key-value query and range query in this hybrid-storage scenario, other more complicated data types are not supported yet. In Chapter 4, we conduct pioneering research on authenticated queries for graph data, which is a popular data type due to many emerging applications, on the blockchain-assisted cloud. The primary challenge is how to design an authenticated data structure (ADS) that supports authenticated queries and can be easily maintained by the blockchain. To this end, we propose a novel ADS, named PAGB, based on the RSA accumulator and completeness set. It can also prevent the original data from being revealed to the public through blockchain or irrelevant queries. We further optimize our design to be more efficient in terms of communication and computation. The effectiveness and efficiency of PAGB are verified through theoretical analysis and extensive experiments.

1.3.3 Verifiable, Secure and Fair Search on Encrypted Numerical Data

Verifiable Searchable Symmetric Encryption (SSE) enables reliable and privacy-preserving search over encrypted data on untrusted clouds. However, most existing SSE designs only focus on the keyword-file search type. A more difficult and pervasive search, range search over encrypted numerical values, remains unsolved. Moreover, the fairness of search in the mutual distrusted scenario without public verification, where data users may maliciously deny the results after the local result verification, is not well addressed yet. In Chapter 5, we take the first step to study the public verification problem atop the blockchain for encrypted numerical search. We design a novel verifiable SSE scheme named Slicer based on a Succinct Order-Revealing Encryption (SORE) scheme to achieve range search on numerical data. We illustrate the security and practicality of our design through rigorous analysis and extensive evaluations respectively.

1.4 Thesis Outline

The remainder of the thesis is organized as follows. We first introduce the related work in Chapter 2. In Chapter 3, we propose a cloud query service called VQL to provide efficient and reliable query over the blockchain data. In Chapter 4, we design a novel ADS named PAGB to realize authenticated query over the outsourced graph data atop the blockchain. In Chapter 5, we present a novel verifiable SSE scheme called Slicer to achieve verifiable, secure and fair search over the encrypted numerical data using blockchain. Finally, Chapter 6 concludes the thesis and points out future work.

This thesis involves the following primary research outputs:

- Zhe Peng, **Haotian Wu**, Bin Xiao, and Songtao Guo. VQL: Providing Query Efficiency and Data Authenticity in Blockchain Systems, in *Proc. of the IEEE International Conference on Data Engineering Workshops (ICDEW)*, Macao, China, 8-12 Apr. 2019.

- **Haotian Wu**, Zhe Peng, Songtao Guo, Yuanyuan Yang, and Bin Xiao. VQL: Efficient and Verifiable Cloud Query Services for Blockchain Systems, in *IEEE Transactions on Parallel and Distributed Systems*, Jun. 2022.

- **Haotian Wu**, Zecheng Li, Rui Song, and Bin Xiao. Enabling Privacy-Preserving and Efficient Authenticated Graph Queries on Blockchain-Assisted Clouds, accepted by *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2023.

- **Haotian Wu**, Rui Song, Kai Lei, and Bin Xiao. Slicer: Verifiable, Secure and Fair Search over Encrypted Numerical Data Using Blockchain, in *Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS)*, Bologna, Italy, 10-13 Jul. 2022.

Chapter 2

Literature Review

Many efforts have been devoted to the combination of verifiable search and blockchain technology. According to the data type that we focus on, we present a comprehensive literature review in terms of three categories, i.e., query services on blockchain data, authenticated graph query on blockchain and verifiable search on encrypted numerical data.

2.1 Query Services on Blockchain Data

2.1.1 Blockchain

As the first successful application of blockchain, Bitcoin [64] has attracted much attention to the blockchain technique. It provides a new way to store transactions on the distributed ledger without the risk of tamper. Ethereum [81], as the successor of Bitcoin, expands the functions by introducing the design of smart contract, which enables more flexible operations on the cryptocurrency. Apart from cryptocurrency, the blockchain methodology contributes to other technologies as well. Provenance [6] establishes the auditable records behind all physical goods for suppliers. ChainSQL

[63] combines blockchain with distributed databases to facilitate a decentralized, auditable and efficient application platform for database users. Other efforts have been done to improve the anonymity and security of the blockchain [36, 74].

2.1.2 Efficient Query

Etherscan [3] is a block explorer and analytics platform where users can explore and analyze data from Ethereum blockchain. With the help of its virtual machine, Etherscan can also provide extra information like internal transactions and state changes in the smart contract. Project Toshi [2] is a fully implemented Bitcoin protocol and supported by PostgreSQL. It offers a RESTful API for large-scale web applications and blockchain data analysis. Blockchain.com [1] provides developers with RESTful service by encapsulating the blocks, transactions and address APIs in the Bitcoin. Aiming to support efficient and accurate queries for certificates, ECBC [89] utilizes a tree structure to facilitate the retrieval of historical transactions. EtherQL [54] employs the conventional database to provide efficient queries for blockchain data analysis. BlockSci [43] can support versatile analysis tasks for different blockchain systems by virtue of an in-memory and analytical database.

The explorers above contain rich information and enable users to explore blocks, transactions and accounts by providing basic interfaces, but the functions of these public APIs are limited. More complex queries (e.g., range queries) for blockchain data are not supported. Moreover, these systems do not provide verification functions to ensure the validity of the query result. In other words, those limitations are great obstacles to providing versatile queries and verifiable query services for blockchain systems.

2.1.3 Verifiable Query

Verifiable query technique that guarantees result integrity is also a hot research topic and has been extensively studied [26, 87]. These studies mainly focus on outsourced databases and can be categorized into two typical methods: circuit-based verifiable computation (VC) techniques and authenticated data structure (ADS). The VC approach like SNARKs [68] can support general queries over databases since it is able to verify arbitrary computation tasks from untrusted workers. But this method incurs a very high and sometimes unacceptable overhead. In addition, it requires a preprocessing step to hardcode the data and query information into the proving key and the verification key, which also degrades the efficiency. To remedy this issue, Ben-Sasson et al. [16] propose a variant of SNARKs called zk-SNARKs, where the size of the output circuit depends on the upper-bound size of the query program. More recently, vSQL [94] provides publicly verifiable SQL queries for dynamic databases by utilizing the interactive-proof protocol. However, it is only restricted to the relational database scenario.

The ADS method employs data structures tailored to specific queries, thus it is generally more efficient in comparison. One of the ADS methods is digital signature schemes, which can be used to authenticate the content of digital messages using asymmetric cryptography. Pang et al. [67] present a verifiable B-Tree by adding signed digests to the B+-tree to authenticate query results based on digital signature. Merkle Hash Tree (MHT), which is a hierarchical tree, belongs to the ADS method as well. GSSE [97] utilizes Merkle Patricia Tree, which is a variant of MHT combining with the prefix tree, to enable verifiable and secure data search in cloud services. Xu et al. [88] design a framework named vChain that adopts the accumulator-based ADS

scheme to achieve dynamic aggregation over various query attributes. To tackle the problem of inefficient search performance and impractical public key management in vChain, Wang et al. [80] further propose vChain+ to optimize the system. However, this approach entails great modification to the underlying blockchain structure and numerous preprocessing operations. Ji et al. [42] propose a decentralized certification framework to provide versatile verifiable queries on blockchain data. It achieves constant storage and validation costs via the underlying trusted execution environment.

2.2 Authenticated Graph Query on Blockchain

2.2.1 Authentication Schemes on Graphs.

Research of authentication on graph data usually focuses on the design of ADS. Martel et al. [60] propose to authenticate trees using Merkle Hash Tree (MHT) and extend it to the directed acyclic graph. In [38], Goodrich et al. employ the hashing scheme and digital signature scheme to authenticate graph data. However, the hash scheme may leak privacy since the verification of hashing involves the hash values of other unshared graph data. To address the privacy leakage issue, a collision-resistant hashing scheme combining with tree representation is introduced in [12]. One main disadvantage of the hashing scheme is that some update on the graph data usually incurs the prohibitive cost of computation. Authors in [38] also introduce path hash accumulator to address the path property query and path search problem. Zhu et al. [96] integrate the cryptographic accumulator with digital signature scheme to provide privacy-preserving authentication for graphs. Kundu and Bertino [48] propose a structural signature using tree traversal and aggregate signature. Based on redactable signatures, de Meer et al. [30] utilize the accumulator to construct a

provably secure scheme that enables the redaction of any nodes. Camacho and Hevia [22] give an efficient solution for directed trees based on hashing with common-prefix proofs.

2.2.2 Authenticated Query on Blockchain.

Authenticated query on blockchain, which enables the integrity verification of blockchain data or data outsourced to the blockchain, is becoming a hot topic in recent years. Some research works like [54, 61] exploit traditional databases to facilitate the efficient query on the blockchain data. However, these designs cannot guarantee the data integrity since the query results may be inconsistent with the blockchain data. Peng et al. [82] propose to write the database fingerprints into the blockchain so that the databases can be trusted. To realize the verifiable query over blockchain databases, Xu et al. [88] propose a novel vChain framework that employs the accumulator as its authenticated data structure. There are also some works that utilize the blockchain to provide verifiable query services over the outsourced data. Hu et al. [41] propose a novel searchable encryption scheme based on the smart contract to ensure the correctness of query results. However, the design is only limited to the file-keyword search and it is costly to store massive raw data in blockchain. Zhang et al. [93] put forth a gas-efficient ADS named GEM^2-tree in the scenario of hybrid-storage blockchain, where the raw data is outsourced to off-chain storage while their digests are recorded in the blockchain. It supports the authenticated range query by designing novel suppressed Merkle B-trees that can be stored on the smart contract. Based on GEM^2-tree, Zhang et al. [92] further extend to keyword search by building a Merkle B-tree for each keyword. Nevertheless, the proposed tree structure only supports range queries for key-value pairs and their keys can be revealed to the public

through the smart contract.

2.3 Verifiable Search on Encrypted Numerical Data

2.3.1 Reliable Searchable Encryption

Verifiable searchable encryption enables users to verify search results returned by untrusted clouds. They can be normally categorized into two types according to the underlying encryption scheme, i.e., verifiable searchable symmetric encryption and verifiable public key encryption with keyword search. Chai and Gong [24] propose the first verifiable symmetric searchable encryption based on a trie-like index named PPTrie. But it only provides keyword-file search on static data. In [76], Stefanov *et al.* achieve the verifiability of dynamic SSE by comparing the message authenticated code and further support forward security. Bost *et al.* [20] improve Stefanov's design and present generic solutions for verifiable SSE. ServeDB [83] designs a tree-based index with cube encoding to support verifiable range queries over dynamic encrypted data. Ge *et al.* [34] propose a verifiable SSE that supports efficient data dynamic update by using a novel accumulative authentication tag. GSSE [97] enables generic and verifiable encrypted search over dynamic data by leveraging Merkle Patricia Tree (MPT) and multiset hashing. It also designs a timestamp-chain structure to prevent replay attacks. Liu *et al.* [59] propose a verifiable searchable symmetric encryption scheme that enables data update for the multi-user setting. Nevertheless, all these SSE schemes cannot provide public verifiability, which enables the verification process to be delegated to a third-party auditor (TPA) without privacy leakage. Soleimanian and Khazaei [75] propose two publicly verifiable SSE constructions upon basic cryptographic primitives. Based on the public key encryption, Zheng *et al.* [95] propose the

first verifiable attribute-based keyword search (VABKS) scheme over static data. Sun *et al.* [77] present an efficient verifiable conjunctive keyword search scheme (VCKS) for dynamic data. However, these designs remain inefficient and require an extra trusted party due to the underlying asymmetric encryption scheme.

There are also some novel research directions that utilize emerging techniques to enhance the reliability of searchable encryption. Some attempts have been made to achieve reliable search over encrypted data by delegating the query processing to trusted execution environment (TEE) [71, 72]. However, these solutions require trusted hardware on clouds and the memory size of an enclave is quite limited. Besides, the search results in their designs cannot be publicly verified since all search processes are sealed in the TEE. Recently, several works have introduced the blockchain technology to devise verifiable searchable encryption schemes [21, 40, 41, 52]. In [41], Hu *et al.* directly store the whole encrypted indexes on the blockchain and execute the search through the smart contract. Nevertheless, this solution may incur a considerable cost of gas since the storage on the smart contract is expensive. To alleviate the burden of the smart contract, Cai *et al.* [21] offload the storage of encrypted files and indexes to the decentralized storage systems. But their design only supports keyword search over append-only encrypted data due to the immutability of blockchain. In addition, the verifiability is achieved by letting the selected arbiter shard re-execute the search process, which is quite centralized in a sense. In [40], Guo *et al.* design a verifiable and forward-secure SSE scheme by virtue of the blockchain. Li *et al.* [52] design a similar system with some improvements in terms of file deletion and on-chain storage. The blockchain-based designs can solve the problem of public verifiability in the mutual distrusted scenario. However, their schemes are limited to keyword-file

search and the costs of on-chain storage are still too high to scale.

2.3.2 Numerical Comparison over Encrypted Data

Order Preserving Encryption (OPE) [17] enables the numerical comparison by directly encrypting the plaintexts, making the ciphertexts preserve the numerical order of plaintext space. CryptDB [70] utilizes OPE to support functionally rich queries over encrypted databases. But OPE fails to guarantee the semantic security of the underlying encryption. It is also vulnerable to inference attacks since the order and frequency of plaintexts are revealed. To solve this problem, Chenette *et al.* [27] propose the first efficient order-revealing encryption (ORE) scheme, which allows the public comparison between ciphertexts. But it reveals the location of the first bit where two ciphertexts differ. Lewi and Wu [51] introduce two new ORE constructions for small domains and large domains respectively. Their design only leaks the location of the first different block instead of a bit. In [31], Demertzis *et al.* present a range SSE scheme by employing a novel tree-like directed acyclic graph. Guo *et al.* [90] design an enhanced ORE scheme to further reduce the leakage for range queries in key-value stores. All schemes above do not consider the verifiability of the numerical comparison when clouds become dishonest.

Chapter 3

Efficient and Verifiable Cloud Query Services on Blockchain Data

Despite increasingly emerging applications, a primary concern for blockchain to be fully practical is the inefficiency of data query. Direct queries on the blockchain take much time by searching every block, while indirect queries on a blockchain database greatly degrade the authenticity of query results. To conquer the authenticity problem, we propose a Verifiable Query Layer (VQL) that can be deployed in the cloud to provide both efficient and verifiable data query services for blockchain systems. The middleware layer extracts data from the underlying blockchain system and efficiently reorganizes them in databases. To prevent falsified data from being stored in the middleware, a cryptographic fingerprint is calculated based on each constructed database. The database fingerprint will be first verified by miners and then written into the blockchain. Moreover, public users can verify the entire databases or several databases that interest them in the middleware layer. We implement VQL together with the verification schemes and conduct extensive experiments based on a practical blockchain system. The evaluation results demonstrate that VQL can efficiently support various data query services and guarantee the authenticity of query results

for blockchain systems.

3.1 Overview

Cryptocurrencies embodied by Bitcoin and its descendants, acting as a modern form of digital currency, have sparked a surge of innovation in decentralized computing. Blockchain, as the fundamental technology of cryptocurrencies, offers many characteristic advantages including decentralized storage and immutability. Besides payment, the blockchain technique can be used in a far wider area such as smart contract [46], supply chain management [79], healthcare [85], distributed storage [8] and IoT [32]. Current blockchain-based systems have tremendous potential in reducing operating costs, increasing resistance to manipulation, preventing fraud and facilitating execution of contracts.

Though the blockchain technique can bypass data storage fraud using distributed ledger with consensus mechanisms, most current schemes only provide limited query services. In pursuit of excellent writing performance, many blockchain systems adopt the key-value database as the underlying database, e.g., LevelDB for Bitcoin and Go client of Ethereum. However, this kind of databases are usually based on LSM-tree [66], which provides barely satisfactory reading performance due to the complicated processing operations, especially for random reading [84]. In addition to the query inefficiency, the query types that native clients support are also limited. Thus, how to provide versatile queries efficiently for all kinds of applications has not been well solved yet.

One approach to tackling the problem of query limitation in the blockchain system is to maintain several extra structures on the peer node, e.g., Project Toshi [2]

and ECBC [89]. Project Toshi saves much more information and indexes besides the native client for richer queries. ECBC builds a tree structure to support efficient query on transactions. In the Bitcoin network, for instance, the raw blockchain data does not contain the balance value of each address. Thus, query service providers can pre-compute and maintain the current balance of each address using the extra list structure so that they can quickly return the result of the balance query without traversing all transaction data. Regretfully, this architecture does not meet the requirement of various queries since the balance list can solely solve the balance query problem. In other words, the extra data structure needs to be customized for the predefined query type. Assume that the Bitcoin node has already supported the query for the address balance. When a user wants to further query about several transaction details related to an address, the peer node still needs to adopt the direct query, searching all blocks in the blockchain for the result. This scheme brings about additional space cost because the node has to maintain an extra and specialized transaction list for each address.

Another method is to take the indirect query by searching the database with high reading performance for blockchain data instead of the original database adopted by the native client. EtherQL [54] integrates the typical database with the Ethereum to expedite the process of data query. Blockchain.com [1] is able to provide the address information since it stores historic transactions in the database in advance. BlockSci [43] incorporates an in-memory database to boost the data query for blockchain analysis. However, these systems assume that the server always returns correct results based on the blockchain data. In fact, the server may return incorrect results that

conflict with the true blockchain data due to some interests or security vulnerabilities [73]. In this case, a feasible mechanism to verify the data authenticity is highly desired. Therefore, our further research problem will be: *Can we manage to provide efficient and verifiable query services for blockchain systems?*

This problem involves the following challenges that need to be addressed: (1) Supporting versatile query services on blockchain data with high efficiency for different applications. (2) Ensuring the data consistency between the queried data and the underlying blockchain data. (3) Providing a verification scheme for query users to validate partial data on the cloud that interests him.

We give an affirmative answer to the problem in this chapter by systematically designing and implementing a Verifiable Query Layer (*VQL*), which is a cloud-based middleware layer providing efficient and verifiable query services for blockchain systems. Superior to the existing designs, our proposed cloud service is capable of meeting the demands of query efficiency and data authenticity simultaneously. A novel framework called vChain [88] achieves the verifiable boolean queries over blockchain data by exploiting an accumulator-based authenticated data structure. However, this work requires radical modification of the existing blockchain systems.

Our system consists of three layers including the underlying blockchain network, the middleware layer and the upper application layer. To cater to various queries from the application layer, the middleware layer will first extract and reorganize the data stored in the underlying blockchain and then store them into the databases. To ensure the validity of the middleware data, each constructed database will generate a fingerprint, which is a cryptographic hash value based on the content and properties of the database (e.g., name, size, timestamp, etc.). This fingerprint will then be verified

by miners and further stored in the blockchain for users to check. By virtue of the immutability of the blockchain, this verification scheme can prevent any falsified data from being stored by the middleware layer. Public users can also download the entire blockchain data to verify the databases if they do not trust the cloud service. In addition, we provide a simplified query result verification scheme to enable users to just check the validity of the databases that their query involves. Given the abundance and popularity of Ethereum-based applications, in this chapter, we employ one of Ethereum testnets to illustrate the feasibility and effectiveness of our proposed system. Our architecture can also be extended to other blockchain applications as VQL can be adapted to any given blockchain system. We conclude our contributions in this chapter as follows:

- The *VQL* is a new cloud query service with a three-layer architecture, which efficiently supports various query services in the blockchain system, e.g., from account query to complicated range query, with no need to browse each block in the whole blockchain. The databases in the middleware are dynamically constructed and updated.

- Our proposed cloud service provides a public verification scheme on the constructed databases to ensure its consistency with the underlying blockchain. The fingerprints of databases are verified and stored in the blockchain by the miner. Miners or users with blockchain data can verify the correctness of a database using these fingerprints.

- We utilize the authenticated data structure to manage fingerprints and put forth a simplified query result verification algorithm for users to verify the received

result without downloading all blockchain data. Users can issue the verification request about the databases involved and efficiently validate their fingerprints.

- We develop the middleware prototype along with the verification schemes and conduct extensive evaluations based on Ethereum testnet and MongoDB. The results demonstrate that *VQL* can efficiently support various query and verification services for blockchain systems.

The remainder of this chapter is organized as follows. We first introduce the preliminary techniques used in our design in Section 3.2 . Then we present the system design and authenticity analysis in Section 3.3, and show the system implementation and evaluations in Section 3.4. We finally conclude the chapter in Section 3.5.

3.2 Preliminaries

3.2.1 Blockchain

The first implementation of the blockchain-based application is the Bitcoin system [64]. By maintaining a distributed ledger, the Bitcoin system creates a decentralized, open and Byzantine fault-tolerant transaction paradigm, which conforms to the requirements of a new cryptocurrency infrastructure. A blockchain network contains the following features:

Transparency: The network is accessible to all participants. Any participant can get the current state of the blockchain system based on the records in the blockchain.

Consensus: All peer nodes in the network will reach consensus on the blockchain (i.e., no unintentional forks). A valid block discovered by an honest peer will be recorded on the blockchain and accepted by other peers.

Immutable and verifiable: Once a block is discovered and globally accepted, any further modification of this block is impossible. All participants can verify the current state based on the records in the blockchain.

3.2.2 Merkle Patricia Tree

The Merkle Patricia Tree (MPT) [4] is first introduced in Ethereum [81], which is a cryptographically authenticated data structure combining the Trie Tree and the Merkle tree. MPT can be used to store (key, value) bindings and there are three kinds of nodes provided in an MPT, i.e., Leaf Nodes (LN), Branch Nodes (BN) and Extension Nodes (EN). A leaf node represents [key, value] pair, where key is the public prefix and value is the terminal value at the node. An extension node also represents [key, value] pair, but here value is the hash of the next node. The branch node is a 17-element array node and used to store viable leaf nodes or extension nodes when the prefixes of keys differ. Among the 17 elements, the first 16 elements are the hex characters, representing the possible prefix of the next node. The last element is used to store the final target value if the path has been fully traversed. In MPT, each node is encoded in Recursive Length Prefix (RLP) code, which is designed to encode arbitrarily nested arrays of binary data, and denoted by its hash. It is noted that the MPT is fully deterministic, which means given the same (key, value) bindings, the MPT constructed from them is guaranteed to be the same regardless of their insertion order and thus have the same root hash.

The superiority of MPT is that it provides $O(\log n)$ efficiency for inserts, deletes and searches, while node insertion and deletion in Merkle Tree incur huge time cost. Moreover, with a publicly known root hash, anyone can prove that there exists a given value at a specific path in the MPT by providing the nodes along the way.

3.3 VQL Design

In this section, we present the design of our proposed VQL that supports efficient and verifiable data query services for blockchain-based applications. We first introduce the overview of the system architecture and the structure of the middleware layer. In order to guarantee the consistency between the middleware databases and the underlying blockchain, we then propose a database verification scheme to prevent falsified data from being stored in the middleware. We further simplify the verification process and put forth the simplified query result verification scheme for ordinary users who do not have blockchain data. This scheme enables query users to validate the query results by downloading partial database data. Finally, we conduct a comprehensive analysis on the data authenticity of our proposed design.

3.3.1 System Architecture

In this subsection, we introduce the architecture of our proposed cloud query service model and the middleware structure along with its update scheme. As illustrated in Figure 3.1, our cloud service model involves three parties, i.e., a blockchain as a distributed database storing a ledger, a middleware layer supporting efficient data query services through reorganizing the blockchain data, and an application layer providing various services for users.

Underlying Blockchain

In the blockchain system, transactions generated from users are stored in the blocks and form a public ledger. Some blockchain platforms such as Ethereum provide APIs to access the transactions stored in each block. In our system, we utilize these

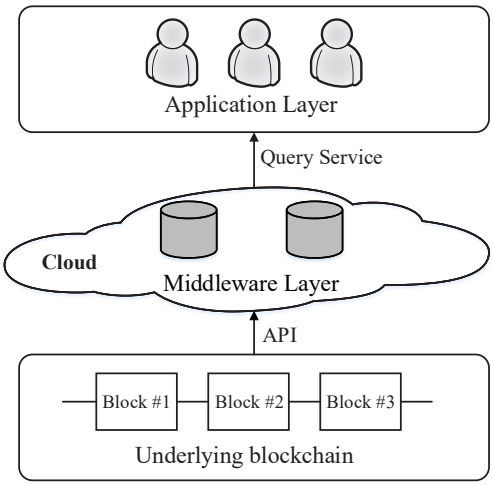


Fig. 3.1: Middleware-based cloud query service model for blockchain applications.

APIs to extract blocks, transaction and balance information stored in the blockchain. This service model can also be applied to other blockchain systems like logistics and supply chain, which record the information of goods delivery and market transaction using consortium blockchain.

Middleware Layer

Based on the blockchain data, the middleware layer extracts and reorganizes all information, e.g., block, transaction and balance, and constructs databases to support efficient data query and data analysis. Figure 3.2 gives an illustration of the designed middleware structure. Our middleware consists of a list of micro databases that contains the data generated in each time interval (e.g., every day) after the specified time point. Each database has a header that contains a cryptographic hash value of the database and some database properties. The hash value of the database can be utilized to verify the data integrity of the database.

Given the underlying blockchain, the algorithm shown in Algorithm 1 updates the

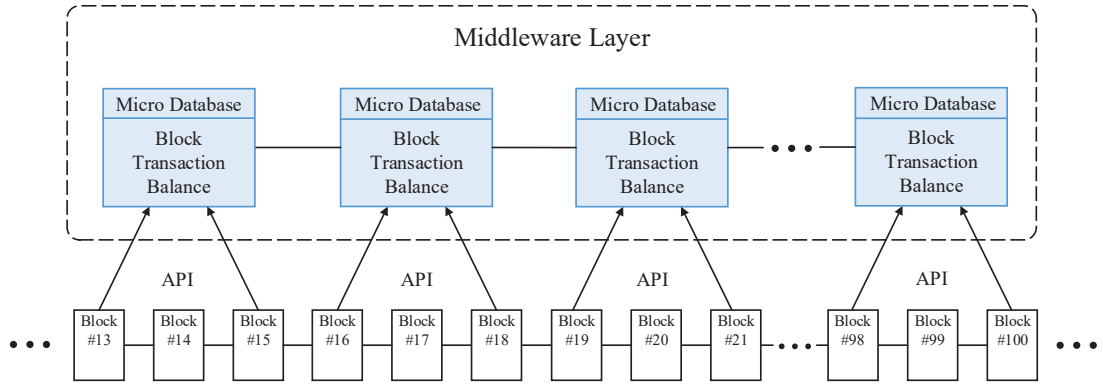


Fig. 3.2: Structure of the middleware layer.

Algorithm 1: Middleware update for Ethereum

Input: BC: Underlying blockchain
Output: DB: Middleware database

1: **for** each day **do**
2: Extract block, transaction and balance information from BC;
3: Calculate balance records;
4: Construct a new micro database mDB containing blocks, transactions and balance;
5: Fingerprint(mDB);
6: Merge mDB into DB;
7: **end for**
8: **return** DB;

middleware layer with newly generated blocks. With the blocks being generated in the blockchain, the system will reorganize the blockchain data and update the middleware layer at a specific frequency, e.g., once a day as shown in the algorithm. At the end of each day, based on the new blocks that have been validated and confirmed by the miners, the middleware layer will be updated in time and support up-to-date query services. The middleware extracts the block, transaction and balance information from the blockchain data and constructs the corresponding databases. Specifically, all attributes of original objects including block and transaction details are retained

while the account balance change on each day will be additionally calculated for query purposes. Then all block, transaction and balance items will be inserted into corresponding databases respectively. After the data extraction, the fingerprint of these databases will be calculated. In order to avoid unnecessary modification of the databases, the middleware will extract information and construct databases only from the immutable blocks in a 'pull-based' method. It is noted that the frequency of daily update is our tentative setting, which can adjust based on the query requirement of different applications, e.g., hourly update for logistics system.

Application Layer

Since the middleware layer constructs databases based on a mature database software, it can provide various data query services for the application layer. Thus, the application layer can efficiently conduct various data analysis and machine learning tasks based on the blockchain data. Besides providing query services for normal users and data platforms, our application layer can also support public audit services for audit institutions. The auditors are able to audit the information in the underlying blockchain using easily verifiable evidence returned by the middleware layer.

3.3.2 Database Verification Scheme

In this subsection, we describe the verification scheme of databases, which can be carried out by miners and public users, that guarantees the consistency between the middleware and the underlying blockchain.

Miner Verification Scheme

Figure 3.3 illustrates the database verification process of the middleware based on the blockchain system. As shown in this figure, various transactions generated by users are stored in the blockchain by the miners. First, the middleware layer extracts transactions stored in the blockchain and reorganizes these data in the databases to provide efficient query services. Second, to prevent falsified data from being stored in the middleware, we generate a unique *fingerprint* for each constructed database in the middleware layer. Finally, the constructed fingerprint of each database will be verified by miners and then stored in the underlying blockchain.

User Database Verification Scheme

We provide a public database verification scheme to guarantee that the data recorded in the middleware layer is consistent with the blockchain and can be verified. Our proposed middleware layer can be deployed in the cloud to be accessed by the public users for data query. Query users can usually trust the query results returned from the middleware layer since the databases stored in the layer have already been verified by miners. In case users have questions about the databases, they can fetch the block data from any honest miner and verify the authenticity of databases using the database fingerprint as the miners do.

Database Fingerprint

The database fingerprint uniquely represents the constructed individual database in the middleware layer. In our design, the fingerprint of the database is determined by two terms, i.e., the data content stored in the database and the property of the constructed database. For the data stored in the database, we first export the data

in a unified and cross-platform format. Then a cryptographic hash value of these data will be calculated based on this format of file using a hash function, e.g., SHA-256. This hash value can be used by miners to check the consistency between the data stored in the database and the underlying blockchain data. The constructed database property contains the database name, the database time, the database size and the database software version. The property of the database can be used to construct the database in the subsequent database verification stage. Finally, the fingerprint can be generated by hashing on the two elements above. It is noted that the fingerprint value is calculated based on the data itself rather than the files stored on the disk. Therefore, the fingerprint is platform-independent, which ensures that miners can obtain an identical fingerprint as long as the data stored in the database is the same.

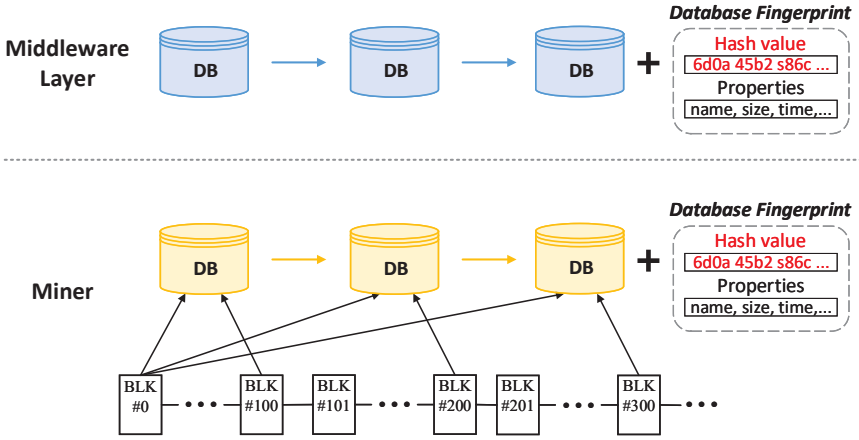


Fig. 3.3: Database verification scheme.

Database Verification

Based on the database and its fingerprint, miners can verify the constructed databases in the middleware to guarantee the consistency between the middleware

data and the underlying blockchain data. After constructing a new micro database to support efficient data query, the middleware layer will first give out its fingerprint.

The algorithm shown in Algorithm 2 describes the proposed database verification scheme for the constructed middleware layer. Since the miner stores the blockchain data locally, he can also construct another database based on his own local data using the same database generation program. The corresponding fingerprint will then be generated by the miner using the predefined hash function on this local database. Thus, for each miner, he can verify the consistency of data between the middleware layer and the underlying blockchain through comparing the two fingerprint values, i.e., the database fingerprint published by the middleware layer and the database fingerprint calculated by the miner based on his blockchain data.

Finally, after successfully verifying the consistency between the middleware layer and the underlying blockchain, miners will store the database fingerprint in the form of a transaction in the blockchain. The transaction transfers zero value from the miner to our middleware with the fingerprint information filled in the data field. The fingerprint is also inserted to an authenticated data structure, i.e., MPT, whose state will be written into the blockchain as well. Once the database fingerprint is recorded in the blockchain, this record cannot be tampered with in terms of the consensus scheme. In the application layer of our system, applications can query data from the middleware layer with trust after checking the database fingerprint stored in the blockchain.

[1]The detailed explanation on the MPT update and synchronization will be further elaborated on in Section 3.3.3

Algorithm 2: Miner Database verification

Input: DB_{mid}: The database constructed in the middleware layer to be verified; DB_{bc}: The database constructed from blockchain by miner; $root_{bc}$: Root of MPT that maintained by miner; BC: Underlying blockchain.

Output: return `ACCEPT` if the database is verified correct; otherwise, return `REJECT`.

1: Get Fingerprint(DB_{mid}) from middleware;
2: Construct DB_{bc} from BC;
3: **if** Fingerprint(DB_{mid}) = Fingerprint(DB_{bc}) **then**
4: Insert Fingerprint(DB_{mid}) into MPT and synchronize MPT to middleware[1];
5: Write Fingerprint(DB_{mid}) and MPT root $root_{bc}$ into BC;
6: **return** `ACCEPT`;
7: **else**
8: **return** `REJECT`;
9: **end if**

Information Record in Blockchain

We propose to write the information regarding the database fingerprints into the underlying blockchain via transactions. These fingerprints can be publicly accessed and ensured by the consensus algorithm to be immutable. In this way, we can enable users to validate the databases using recorded fingerprints without modifying the underlying block structures. Each time the miner verifies the validity of databases in the middleware layer, in addition to the database fingerprints themselves, he also records the root of Merkle Patricia Tree, which is used to store all database fingerprints. This tree root is a deterministic hash generated by all database fingerprints and provides a form of cryptographic authentication to the data structure. In other words, the tree root represents a unique state of the entire tree. Therefore, we write the tree root hash into the blockchain as well for the application layer to check. Note that, the information writing policy may differ when our verification scheme is applied in diverse blockchain systems, such as the public blockchain, private blockchain, and consortium

blockchain [55]. In case of private blockchain or consortium blockchain, miners can be forced to write some certain information into the block of specific height. However, in the scenario of public blockchain, due to the propagation of transaction information and the competition among transactions, the information cannot be guaranteed to be written in the stipulated block.

Failed Verification Situation

During the database verification process, we also consider the failed verification situation. If the local fingerprint calculated by the miner is different from that provided by the middleware, an error report will be sent to the middleware layer. When the middleware layer receives a certain amount of failed verification reports, it will execute a diagnostic procedure to check the correctness of database until no error reports arrive. In case of extreme situations, e.g., a fork due to network partition, the middleware and miners will find the correct chain to catch up with. Meanwhile, the databases will be rebuilt and the fingerprints are revoked. The failed verification report scheme will help the middleware to correct false database fingerprints.

3.3.3 Simplified Query Result Verification Scheme

The database verification scheme in the last subsection is designed for the miners. For query users without blockchain data, they need to download the entire blockchain from credible miners and verify all databases by constructing them. It is a quite radical method to guarantee the authenticity of databases, but sometimes it is unnecessary for an ordinary user to download all data for just one simple query. To remedy this issue, we propose to employ the authenticated data structure for fingerprint management and put forth a simplified query result verification algorithm

to ease the process of result verification for query users.

Merkle Patricia Tree for Database Fingerprints

Due to the uncertain factors in public blockchain systems, e.g., network delay and transaction fee, the middleware and users cannot get the height of the block where the database fingerprint is stored in advance. The block height is determined only after the fingerprint is indeed written into one block and confirmed by miners. Using the confirmed block height, users can find the fingerprint in the specified block and check its correctness. Thus, we employ Merkle Patricia Tree to store these [fingerprint, height] pairs since it is able to prove the existence or non-existence of a given database fingerprint. In this way, query users can directly check the correctness of the given database fingerprint without searching the block containing the information. It is noted that the MPT is maintained by miners and will be updated each time miners finish verifying the validity of databases and writing database fingerprints into the blockchain. Moreover, the MPT data will also be synchronized to the middleware layer by miners so that the cloud can provide Merkle proofs for query users. The reason why miners adopt MPT instead of directly returning true or false for every fingerprint request from users is because providing all users with validation services is over-demanding for miners. The MPT enables miners only need to show the MPT root while leaving the proof work to cloud servers.

We use an example shown in Figure 3.4 to instantiate the database fingerprint storage in MPT. We presume that initially there are four database fingerprints as presented in the key-value list, in which the key is the database fingerprint hash and the value represents the height of the block where the fingerprint information is written. Using these fingerprints, we can build the Merkle Patricia Tree as given in

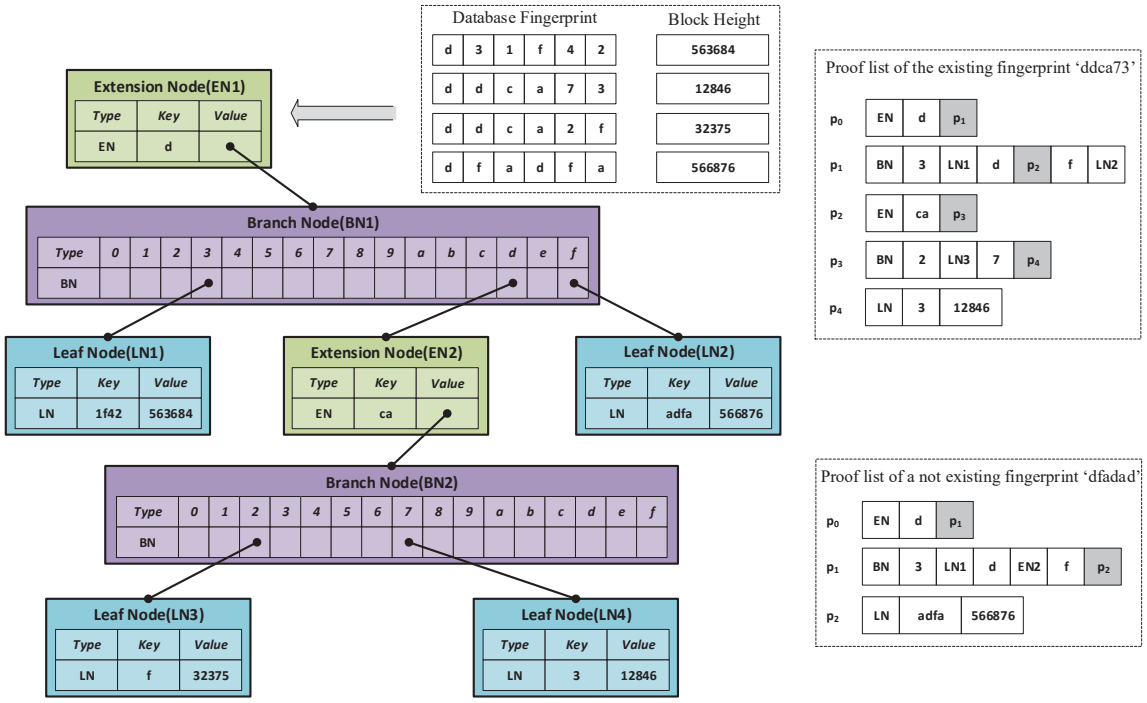


Fig. 3.4: An illustrative example of database fingerprints MPT.

the figure. Here we neglect the detailed descriptions of the operations in MPT, e.g., insertion, update and deletion, since there have already been some implementations available.

Simplified Query Result Verification Process

Figure 3.5 illustrates the relationships among the miners, users and the middleware-based cloud in our data verification scheme. Our system comprises three parties: *miners*, who mine the blocks and maintain credible block data in the underlying blockchain layer; *users*, who lie in the application layer and send queries to the cloud about the data in blockchain; *cloud query services*, which belong to the middleware layer and provide query services for users. The dashed arrow from miners to the cloud

services signifies the aforementioned miner database verification, while the solid arrows represent the interactions in the simplified query result verification scheme. In our simplified scheme, miners only need to synchronize the MPT to the cloud services and provide the MPT root hash for query users if they request verification. Each time the user sends a data query to the cloud, the server will return a query result along with the database back-up files that this query involves and their corresponding Merkle proofs. This function of database back-up and reconstruction can be supported by some commercial database systems, e.g., MongoDB. Combining with the credible MPT root hash obtained from miners, the user can easily check the validity of those database fingerprints based on the Merkle proofs. If the user wants to further confirm the information about the root hash and database fingerprints, he can search the blocks according to the corresponding block height stored in MPT.

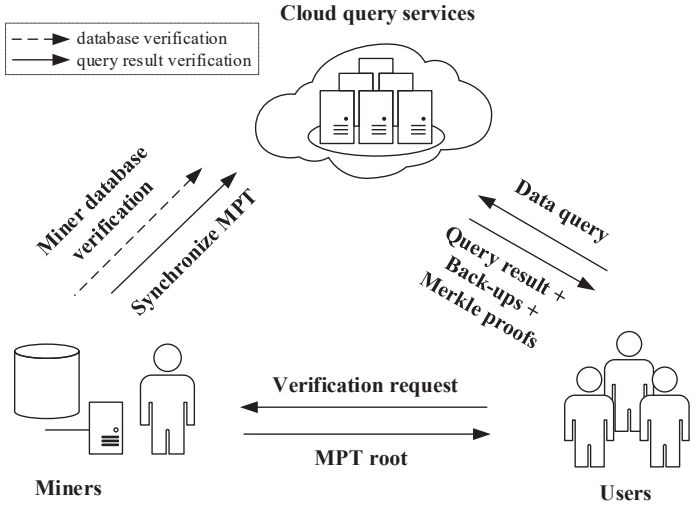


Fig. 3.5: Data verification scheme.

Algorithm 3 shows the simplified query result verification algorithm performed by query users. When the user requests a data query to the middleware, he will get a query result $result_{mid}$ from the server together with the fingerprints of all databases

involved, i.e., DBs. After downloading the corresponding database back-up files, the user can reconstruct these databases and calculate their fingerprints. Meanwhile, he will send a verification request to the miners and thereby get the latest MPT root hash $root_{bc}$, which signifies the newest state of all verified databases. With the database fingerprints calculated, the user can send them to the middleware layer and obtain the Merkle proof for every fingerprint. Based on the proof for each fingerprint, he will calculate the root hash $root_m$ by themselves and then compare with the true root hash $root_{bc}$. When the two root hashes are equal and the key is in accord with the path, the correctness of this fingerprint can be guaranteed. The process of proving the presence of the fingerprint using MPT root and Merkle proof is included in *Prove* function and will be detailed in the Merkle proof part. If all databases involved are confirmed correct, then the user can query the databases that are locally constructed from back-up files and get the query result $result_l$. When this result is identical to the previous result $result_{mid}$ from the middleware, the user can finally trust and accept the result.

Merkle Proof for Fingerprints

Now suppose a query user wants to check the existence of the database fingerprint 'ddca73' which already exists in the MPT (see Figure 3.4). The value of this key is stored in the leaf node LN4 and its search path from root to leaf is {EN1, BN1, EN2, BN2, LN4}. Based on the path, our middleware layer can provide a Merkle proof, which is a list of RLP code of the nodes along the path (see the right part of Figure 3.4), for the user to prove the existence of the key. In this case, the Merkle proof for 'ddca73' is a 5-element array, i.e., p_0 to p_4. Each node is referenced inside the previous element except the root node p_0. Using this list, the user can check the

Algorithm 3: Simplified query result verification

Input: $root_{bc}$: Root of MPT stored in blockchain; DBs: Middleware databases that the user query involves; $result_{mid}$: Query result provided by middleware layer; $proof$: Merkle proof of a given database fingerprint;

Output: return `ACCEPT` if $result_{mid}$ is correct; otherwise, return `REJECT`.

```
1: Get the latest MPT root root_bc recorded in blockchain from miners;
2: verified ← FALSE;
3: for each DB ∈ DBs do
4:    Construct DB from the back-ups in the middleware layer;
5:    Send Fingerprint(DB) to the middleware layer;
6:    Get the Merkle proof proof from the middleware layer;
7:    verified ← Prove(root_bc, fingerprint, proof);
8:    if not verified then
9:       break;
10:   end if
11: end for
12: if verified then
13:   Query DBs locally and get the query result result_l;
14:   if result_mid = result_l then
15:      return ACCEPT;
16:   else
17:      return REJECT;
18:   end if
19: else
20:   return REJECT;
21: end if
```

correctness of the value and RLP code of each element in the array successively from head to tail, i.e., in the order from root to leaf. If the root hash finally calculated is identical to the publicly known root value and the prefixes along the path equal to the fingerprint, then this database fingerprint is considered to truly exist. Algorithm 4 shows the pseudo-code of the *Prove* algorithm executed by the query user to verify whether the database fingerprint exists in MPT.

Similarly, we can also utilize the Merkle proof to prove the non-existence of a given

Algorithm 4: Prove algorithm

Input: $root_{bc}$: Root of MPT stored in blockchain; $fingerprint$: Fingerprint of the database to be checked; $proof$: a n-element list of p_i, i.e., Merkle proof of the given database fingerprint;

Output: return `TRUE` if $fingerprint$ exists in MPT; otherwise, return FALSE.

```
1: if Hash(p_0) ≠ root_bc then
2:    return FALSE;
3: end if
4: for i ← 0 to n − 1 do
5:    if i = n − 1 then
6:       if key in p_i conforms to fingerprint then
7:          return TRUE;
8:       else
9:          return FALSE;
10:      end if
11:   end if
12:   if i < n − 1 then
13:      if key in p_i conforms to fingerprint and key's value = RLP(p_{i+1}) then
14:         continue;
15:      else
16:         return FALSE;
17:      end if
18:   end if
19: end for
```

key. Suppose a user reconstructs a database using broken or tampered files and thus calculates a wrong fingerprint, e.g., 'dfadad' in the figure, which does not exist in the MPT. Our server will return the Merkle proof based on the search path {EN1, BN1, LN2}, i.e., p_0 to p_2 as shown in the figure. Here the hash of root p_0 can be verified by calculating from head to tail and still equals to the root hash obtained from miners. Nevertheless, the prefixes generated by the proof differ from the fingerprint key, which means the fingerprint does not exist in the MPT.

3.3.4 Data Authenticity Analysis

Since the user receives the query result from the middleware, as long as the queried database is consistent with the underlying blockchain, the authenticity of the queried data is guaranteed. Thus, we conduct the data authenticity analysis from three aspects: the rewarding scheme for miners, the integrity of databases and the verifiability of query results.

Rewarding Scheme for Miners

In our cloud service, the verification of databases in the middleware layer is realized by miners, which may cost some computing resources and storage space. Thus, a rational rewarding scheme is required to incentivize miners to verify the databases. Owing to the different demands and scenarios of the blockchain systems, the rewarding schemes for miners in our query model may differ. For the private blockchain system, since the miners and middleware layer are private to provide services, the verification and record fees are not needed. As for the consortium blockchain system, depending on the various agreements between communities in the consortium, the middleware layer may need to pay the fees or not. When applied to the public blockchain system, our middleware will give some rewards to the miners or mining pools [50] who successfully validate the databases.

Since the above rewarding scheme for database verification is not supported by the existing blockchain systems, we give two possible solutions to make our verification scheme practical. First, we can implement a new blockchain system based on an existing open-source project, incorporating the incentive mechanism for verification. In order to get the rewards, miners can validate the middleware databases and record

corresponding database fingerprints into the blockchain. The validation process, the rewarding mechanism and the management of fingerprints are all hard-coded in the blockchain peer nodes. Second, we can deploy the smart contract on the current blockchain system to facilitate the verification process of miners. Miners can construct the database from their own blockchain data using the same database generation code provided by our middleware. Then the database fingerprint can be calculated and sent to the smart contract for confirmation. Finally, our middleware will announce the correct fingerprint and send the rewards to the miners who correctly verified the database. More miners will participate in the verification task if the reward is attractive enough. The smart contract can also maintain the MPT storage of database fingerprints in the simplified query result verification process.

The Integrity of Databases

The consistency between the databases in the middleware layer and the underlying blockchain data is realized through the database verification scheme by the miners. Each time a new database is constructed, the middleware layer will back up the database and publish the back-up files and its fingerprint. In the meantime, miners can construct another database based on his blockchain data following the same rules and calculate its fingerprint using the predefined hash function. If the fingerprint of this database is the same as the one given by the middleware, then the database is verified correct. Moreover, the integrity information is immutable since the fingerprint will be written into the blockchain after verification and managed by the MPT structure.

The Verifiability of Query Results

After the integrity of databases in the middleware layer is guaranteed, the query results that users receive should also be consistent with the middleware databases. We provide two methods to realize the verifiability of query results, i.e., user verification in the database verification scheme and the simplified query result verification scheme. The user database verification requires users to download all blockchain data and check the consistency like miners, the authenticity analysis of which is just conducted. It is noted that when we request data from the miners, we will first connect to the anchor nodes in the blockchain network, which means the miners we query are assumed to be absolutely reliable. Therefore, the situation of malicious miners is trivial and out of the scope of this thesis. The simplified query result verification scheme allows users to download only the involved databases rather than all databases and check the validity of their fingerprints by leveraging the MPT structure. Since the databases are reconstructed based on the back-up files and their fingerprints are calculated locally by users, the authenticity of the involved databases can be ensured if these fingerprints indeed exist in the MPT maintained by miners. Finally, users can query the valid databases locally and check whether the result is consistent with the query result returned by the middleware layer.

3.4 Implementations and Evaluation

To test the feasibility and performance of our cloud query service, we implement a prototype on a testnet of the well-known blockchain system Ethereum.

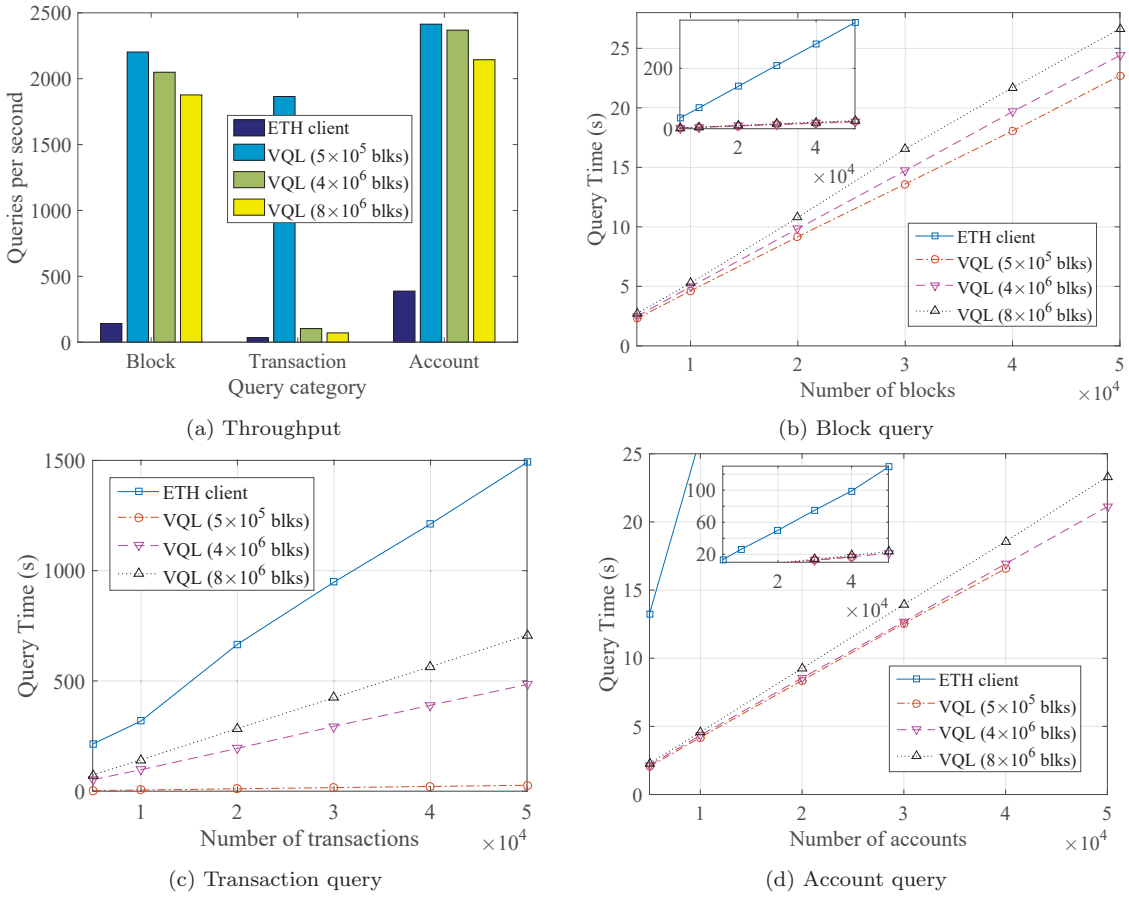
3.4.1 Prototype Implementation

Our middleware supports user-friendly APIs for user applications and APIs for the underlying blockchain. The user application APIs support various queries and database verification for auditing, including the query interface and validation interface for the block, the transaction and the balance information. Meanwhile, the blockchain APIs support query functions to collect records from the blockchain, e.g., the data request interface for the block, the transaction and the global state of the blockchain. We employ the popular document-oriented database MongoDB for data storage of the middleware. The reason why we use MongoDB is that it can support efficient query on general and rich data, e.g., arbitrary forms of transactions and smart contracts. It can also achieve good reading performance by building the indexes. The MPT for fingerprint storage is implemented in JavaScript and stored in LevelDB. To evaluate the system performance without the interference of network communication, we build up the experiment platform on a cloud server with Intel Xeon 2.67GHz CPU and 32 GB RAM, running Ubuntu 16.04 LTS. Our data query services are based on the blockchain data of Rinkeby network, one of the popular Ethereum testnets, with the block height varying from 0 to 8,000,000.

3.4.2 Performance Evaluation

The process of synchronization from scratch in blockchain systems usually needs to be done only once because of the fact that blockchain data is immutable. Moreover, the time cost of the synchronization process is generally dominated by the network bandwidth and the performance of the physical machine. Nodes with low network bandwidth or bad performance may take several days to catch up with other peers.

Therefore, the evaluation of blockchain synchronization is out of the scope of this thesis. We test various data query services in terms of throughput, block query, transaction query, account query and range query. We contrast our proposed VQL with the Geth client, which is an official Go implementation of the Ethereum protocol, in terms of query efficiency.



(a) Throughput

(b) Block query

(c) Transaction query

(d) Account query

Fig. 3.6: Query performance of ETH client and VQL.

Throughput

We first evaluate the throughput performance of our proposed system VQL comparing with ETH client. The ETH client synchronizes the blockchain to about

8,000,000 and our VQL also organizes all information within the same block height. In addition to the comparison between the ETH client and VQL, we also evaluate the query efficiency of VQL with different blocks synchronized, i.e., 500,000, 4,000,000 and 8,000,000. Three kinds of queries are conducted, including querying a block by the block number, querying a transaction by transaction hash, and querying the balance of an account by address. As shown in Figure 3.6a, the throughput of VQL is about 13.2, 2.1, 5.5 times as that of ETH client in terms of block, transaction and account query respectively. When we query a block by the block number, the VQL and ETH client can support 1.88K queries/s and 142 queries/s, respectively. For querying a transaction by the transaction hash, the VQL and ETH client are able to process about 70.6 queries/s and 33.5 queries/s. If we query the balance of an account by address, both systems can achieve higher throughput (i.e., 2.14K queries/s and 387.9 queries/s) because of the relatively smaller amount of accounts. The results show that our proposed VQL can achieve higher throughput than the native ETH client. From the performance of VQL under different load scenarios, we can see that the increasing number of synchronized blocks will degrade the query throughput of all query categories. The throughput of transaction query drops rapidly because the amount of transactions increases sharply as the block height grows.

Block Query

In our experiments, query efficiency is a critical criterion for the proposed query supported system. In the blockchain, various transactions generated by users are stored in the blocks. Thus, we first compare the block query time of different systems (e.g., ETH client and VQL) to show the query efficiency of our system. ETH client provides a JSON RPC API to support the block query. Accordingly, we develop an

API in the middleware layer to provide query services about blocks.

Since a single block query can usually be completed in milliseconds, we query for a randomly selected list of blocks and record the time of completing these queries. We conduct experiments of block query based on scenarios with block number from 0 to 50,000. As shown in Figure 3.6b, the block query time is compared with ETH client and VQL will different loads. With more blocks queried, the query time is significantly increased using ETH client, while the time of VQL can still remain at a relatively lower level. This ETH client requires plenty of query time, for example, 351.9 seconds in the evaluation of the 50,000-block scenario. On the contrary, our proposed system VQL can save much query time, which optimizes the data storage for faster queries (e.g., 26.6 seconds in 50,000-block scenario). From the comparison between different loads of VQL, we observe that the number of synchronized blocks has limited effects on the query efficiency since the query time only increases slightly.

Transaction Query

The query about individual transaction information is also supported in our system and we conduct experiments on the query time of transactions. The native ETH client provides limited APIs for the retrieval of transaction details while our VQL can support queries on transactions by all attributes that a transaction has. In this experiment, we choose the common API, i.e., query by the transaction hash, to present the comparison of the query efficiency.

As shown in Figure 3.6c, the transaction query time is compared between ETH client and VQL with different loads. Since a single transaction query can usually be completed very fast, we query for a bunch of randomly selected transactions to

evaluate the time. We test cases with different numbers of transactions in the experiment, from 0 to 50,000 transactions. The number of transactions almost linearly promotes the query time in all cases. But VQL takes only about half of the time that ETH client uses to query the same amount of transactions under the same data load. As for the comparison between different number of synchronized blocks in VQL, the query time of 8,000,000-block scenario is much longer than that of 500,000-block case. It is because the volume of transactions grows dramatically when the block height increases.

Account Query

In our middleware layer, each constructed micro database contains two parts of data: the transaction details and the balance details of all accounts. Apart from the original blockchain data, the account balance also provides an extra historical balance description for each account, e.g., balance change in each day. Since the native ETH client only provides the API for current balance query, we also test the same function in our VQL system.

As shown in Figure 3.6d, we conduct experiments to evaluate the query time of account balance for ETH client and VQL. Because the query time of a single account is too small to measure, we still query for a randomly selected list of accounts to test the efficiency of balance queries. Scenarios with different numbers of accounts, from 0 to 50,000 accounts are tested in the experiment. We can see that the query time of account balance increases linearly as the number of accounts grows. The query of account balance with the same amount in VQL can be completed within one fifth of the time that the ETH client takes, i.e., 128.9s. This is because, in our proposed middleware, the information of account balance is calculated in advance and well

		ETH client	VQL
Temporal range query	Block	40.53s	0.04ms
	Transaction	1625.88s	0.036ms
	Balance	10.23s	0.041ms
Numerical range query	Block	—	0.034ms
	Transaction	—	0.033ms
	Balance	—	0.036ms

Table 3.1: Evaluation of range query.

organized in databases. In addition, the comparison between different loads in VQL shows that the number of synchronized blocks slightly promotes the query time.

Range Query

Besides the individual account query, range query is also supported by the middleware layer since the application layer is usually required to conduct various data analysis and machine learning tasks. For these tasks, many features will be extracted through a range of data, e.g., accounts that have transactions in one day or transactions with amount over 100 ETH. Our middleware can provide this ability of data query within a specific range while the native ETH client cannot perfectly support.

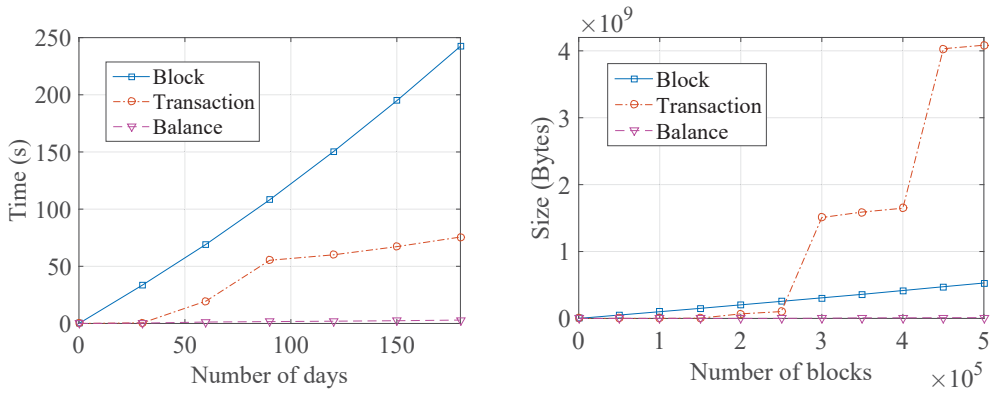
In our experiments, we conduct performance evaluation about range query for block, transaction and account, respectively. Considering many applications related to data analysis, we implement two kinds of range queries, i.e., temporal range query and numerical range query, for the information of block, transaction and balance. The temporal range query means the query on blockchain data within a specific time range, e.g., the transactions generated last month. The numerical range query represents the query on some numerical fields of the data, e.g., the transactions with value less than 1 ETH. As shown in Table 3.1, the time of different range query categories is compared with ETH client and VQL. We query blocks generated in one day and record the

query time. The VQL can finish the query with 0.04ms while the ETH client needs 40.53s. Then we query transactions within a randomly chosen day and record the time used. Our VQL completes the query within 0.036ms and the ETH client costs 1625.88s. Finally, we query the account balances that have changes in one day, which means transactions are performed between these accounts. The experiment result shows that the VQL needs 0.041ms while the ETH client uses 10.23s. It is noted that the ETH client does not directly support temporal range query. To achieve it, we traverse the blocks using the block number and get the transactions inside. However, the numerical range query cannot be supported even using this method since the ETH client has to read all blockchain data to judge the numerical values, which is excessively time-consuming. Therefore, we mark the inapplicability using $'-'$ in the table for numerical range query. In general, the proposed VQL needs much less time to finish different range queries than the ETH client. Our VQL shows remarkable advantages over the ETH client due to the well-organized micro databases in the middleware, which are very efficient for range queries.

Database Verification

Database verification efficiency is also an important criterion for our proposed system. We set the generation frequency of database fingerprint to be once a day, which means the middleware produces the fingerprints for block, transaction and balance using the respective daily data. As shown in Figure 3.7a, we record the time of database verification after the blocks are generated in the blockchain each day. When the middleware layer has constructed databases based on the blockchain for 180 days, the verification time of block databases for a miner is 242.4s and that of transaction databases is 75.6s. The balance databases take the least time, i.e., 2.98s

for 180 days, since the size of involved information is quite small. With more daily databases generated by the middleware, the database verification time increases. We can see that there is a fluctuation in the transaction database between 50 and 100 days. This is because the amount of transactions in these days suddenly grows, which leads to more verification time. Thus, our proposed system is able to efficiently verify databases constructed in the middleware layer and applicable to practical blockchain systems.



(a) Verification time (b) Database size

Fig. 3.7: Performance of miner database verification.

Database Size

Considering the storage space efficiency, we also test the size of databases to be verified in the middleware layer during the database verification process. We record the size of each database for block, transaction and balance as the blocks are generated in the blockchain. As shown in Figure 3.7b, when the middleware layer has constructed databases for 500,000 blocks in the blockchain, the size of transaction database stored in the middleware layer reaches about 4GB while the size of block database is around 500 MB. We can observe from the figure that the size of the

transaction database increases notably twice due to the large amount of transactions in some blocks. The balance database always occupies the least storage and its size is only 12MB even when the number of blocks reaches 500,000. Thus, our proposed system can efficiently store the databases constructed in the middleware layer to provide query services and database verification.

Proof Cost in MPT

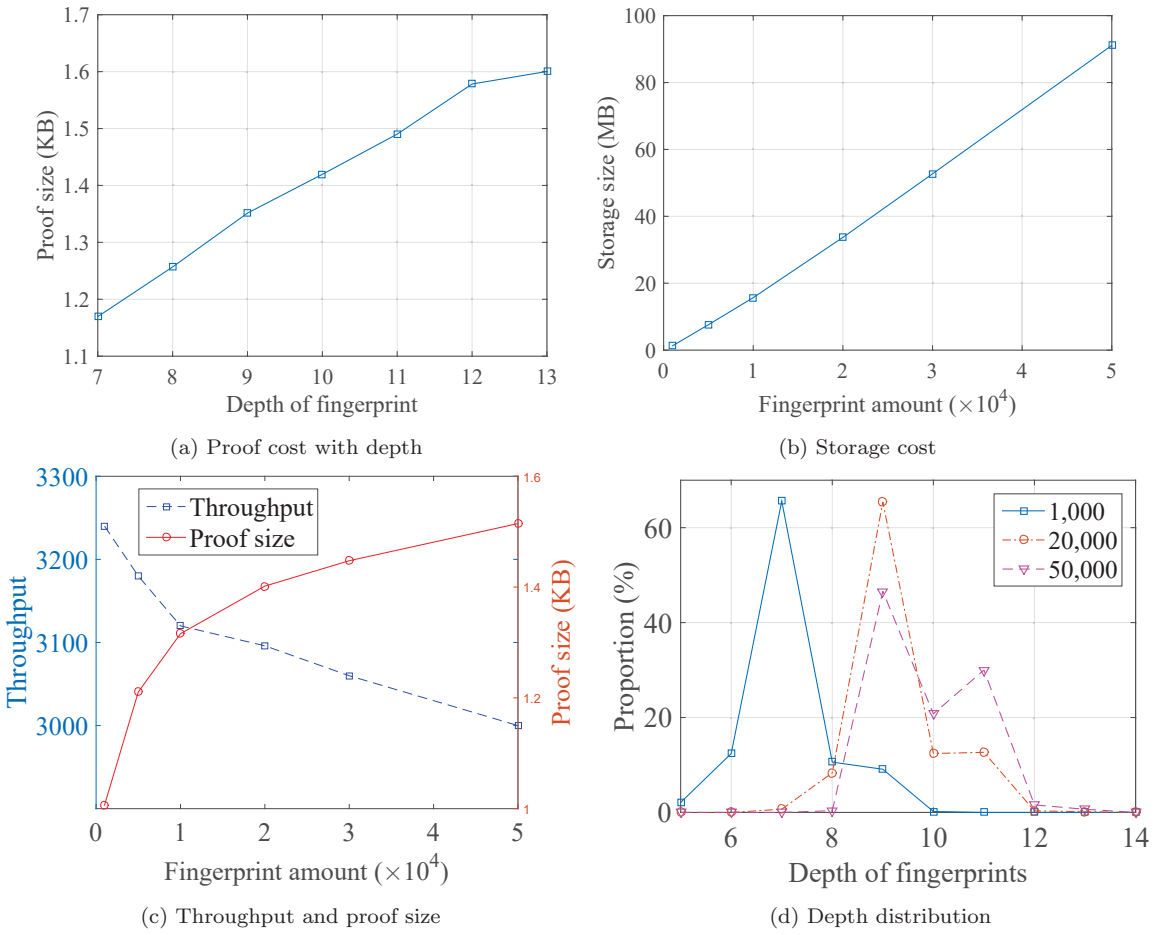
The cost of simplified query result verification is dominated by the communication overhead incurred by Merkle proof. The size of Merkle proof is mainly decided by the number of layers in MPT. The deeper the leaf node locates in MPT, the longer its search path becomes. Thus, we measure the size of proof that the middleware layer returns for each database fingerprint. In our evaluation, we employ SHA-256 hash function to generate the fingerprint for the database, thus the key to be stored in MPT has 256 bits. We insert 2,000 keys into the MPT and record the average length of Merkle proof that MPT provides by invoking the prove function for each key. As presented in Figure 3.8a, the size of Merkle proof is only a few kilobytes and closely associated with the depth of key. The depth of the fingerprint is principally distributed between 7 and 13, and the proof size gradually increases as the depth grows, which conforms to our previous analysis. This is because Merkle proof is a list of nodes along the path and the RLP code of one node is about 100 bytes. Compared with the size of the block data needed in miner database verification, the overhead of giving the Merkle proof is practically negligible.

Storage Cost in MPT

Since the MPT for database fingerprint is updated by miners and will be synchronized to the middleware layer, it will cost storage space in both miners and the middleware layer. In order to show the storage cost of MPT with the amount of fingerprint increasing, we investigate the size of the LevelDB database files generated by the MPT when the total amount is 1,000, 5,000, 10,000, 20,000, 30,000, 50,000. Observing from Figure 3.8b, we can see that the amount of fingerprint linearly promotes the storage cost, which indicates that MPT does not bring about much cost of extra storage space as the amount of fingerprint grows. The storage cost increases to 90 MB when the fingerprint amount reaches 50,000, which is relatively small compared with the size of databases constructed in the miner database verification process. Therefore, the storage cost is acceptable to achieve our simplified query result verification scheme.

Throughput and Proof Size

In our simplified verification scheme, the middleware layer will return a Merkle proof for each query from users. Thus, we investigate how many verification requests the middleware is able to handle concurrently and how much overhead it costs to return a Merkle proof. The performance is presented in Figure 3.8c, which includes the throughput and proof size under various number of fingerprints. We observe that the throughput of returning proofs decreases when the amount of fingerprints grows. This is because the MPT becomes larger when more fingerprints are stored, which leads to longer search time for each fingerprint. The middleware can support 3,000 verification requests per second with 50,000 fingerprints stored, which is acceptable



(a) Proof cost with depth

(b) Storage cost

(c) Throughput and proof size

(d) Depth distribution

Fig. 3.8: Performance of simplified query result verification.

as well. Meanwhile, we can also see that the average size of Merkle proof rises slowly when the number of fingerprints increases.

Depth Distribution

We further investigate the reason behind the proof size and observe that the distribution of fingerprint depth greatly changes under different cases. Figure 3.8d shows how fingerprint depth distributes under scenarios with different fingerprint amounts. When the amount of fingerprints is 1,000, the depth mainly distributes around 7 and the proportion of 7 exceeds 65%. As the amount increases, the majority of fingerprint depth rises slightly. The depth of 9 accounts for more than 60% of the whole fingerprints when the total amount reaches 20,000. In the scenario with 50,000 fingerprints, the proportion of depth 11 gradually grows to about 30%, leading to a higher average depth. Combining with the previous observation from the proof cost in Figure 3.8a, the increasing average proof size conforms to the distribution of fingerprints. Compared with the size of database itself in the middleware, the proof cost in our simplified query result verification is relatively small.

3.5 Chapter Summary

In this chapter, we propose VQL, a cloud query service layer that can provide efficient and verifiable query services for the blockchain system. The proposed framework has a three-layer architecture, including the underlying blockchain network, the middleware layer and the application layer. To realize this system, first, the middleware layer extracts the data stored in the underlying blockchain and reorganizes them in databases to provide various query services efficiently for the upper application layer. Second, to prevent falsified data from being stored in the middleware, a cryptographic

hash value, named as fingerprint, is calculated based on each constructed database. Finally, the database fingerprint is recorded in the blockchain after being verified by miners. In order to ensure the data integrity, we design the database verification scheme for miners and the simplified query result verification scheme for public users. We implement VQL on the cloud and conduct extensive experiments based on a practical blockchain system Rinkeby. The evaluation results demonstrate that VQL can effectively and efficiently support various data query services and guarantee the authenticity of query results for the blockchain system. Our proposed query service can be deployed on the cloud for practical applications and accessed by public users for efficient and versatile data queries.

Chapter 4

Privacy-Preserving and Efficient Authenticated Queries on Outsourced Graph Data

Prior research has introduced a new scenario of blockchain-assisted clouds where the data owner outsources original data to cloud servers and stores some metadata on the blockchain. Despite some research on key-value query and range query in this hybrid-storage scenario, other more complicated data types are not supported yet. In this chapter, we conduct pioneering research on authenticated queries for graph data, which is a popular data type due to many emerging applications, on the blockchain-assisted cloud. The primary challenge is how to design an authenticated data structure (ADS) that supports authenticated queries and can be easily maintained by the blockchain. To this end, we propose a novel ADS, named PAGB, based on the RSA accumulator and completeness set. It can also prevent the original data from being revealed to the public through blockchain or irrelevant queries. We further optimize our design to be more efficient in terms of communication and computation. The effectiveness and efficiency of PAGB are verified through theoretical analysis and extensive experiments.

4.1 Overview

With the rapid advance of cloud computing, many data owners including individuals and enterprises desire to outsource their data to cloud service providers. However, this paradigm needs data owners to fully trust the cloud servers, which does not conform to practical cases. The potentially compromised cloud service providers may return incorrect or incomplete results to query users. Besides the intentional behaviors due to some commercial interests, the dishonest behaviors may also arise from software bugs, hardware failures and other security vulnerabilities [78]. This issue can be thwarted by introducing verifiable query processing that requires cloud servers to prove the authenticity of its results [9, 11].

Thanks to the emergence of the blockchain technology, the smart contract [81] provides a new paradigm for trusted storage and trusted computation environment. Some data owners choose to store data on blockchain as a complement to cloud servers. However, storing original data directly on-chain is almost infeasible due to the limited storage space and excessive gas cost. One approach to this problem is to employ the blockchain-assisted cloud [13, 56, 93], where both on-chain storage and off-chain storage are adopted. The data owners still outsource their raw data to off-chain storage like Google Cloud Storage or Amazon Simple Storage Service, but only store some metadata (e.g., hash value of original data) on-chain as notarization of the outsourced data. To ensure the integrity of the data retrieved from the untrusted off-chain storage, the on-chain metadata can be utilized to authenticate the query result.

The major problem that the introduction of blockchain can tackle is the freshness of the outsourced data, which is barely solved by most existing designs [91]. This

defect renders the system vulnerable to the replay attack where the cloud may return a stale or currently invalid signature to query users. In existing authentication schemes, data owners usually generate the *authenticated data structure* (ADS) , including tree-based structures [83, 87], aggregate signatures [47, 48] and cryptographic accumulators [38, 96], based on his local data. Then they will sign the ADS using the standard digital signature scheme and provide the signature for query users to check. Due to the online burden, the data owners will not keep or distribute the signature to all query users after each update, but delegate it to the cloud instead. This approach is vulnerable to replay attacks [14], i.e., the cloud may return a stale or currently invalid signature to query users. Fortunately, the blockchain-assisted cloud is immune to attacks on data integrity, including data tampering and replay attacks, because the integrity of ADS is guaranteed by the safety of blockchain and the freshness of ADS can be ensured by the liveness of blockchain. Since the ADS is accommodated in the blockchain, data owners are also free from storing and distributing the ADS while still being able to provide authenticated query services. In addition, all updates on the outsourced data can be publicly recorded in the blockchain for further forensics and provenance.

Some primary query types like key-value queries, range queries [93] and file-keyword [41] queries have been well investigated. However, graph data, which is another significant but more complicated data structure, has not been supported yet. Graph data has attracted a tremendous amount of interest thanks to many emerging applications based on the graph model, like social networks [58], biological networks [62], healthcare [86] and knowledge management [69]. Adopting the aforementioned hybrid-storage model, data owners can resort to cloud servers for the storage and

management of graph data using some mature graph databases, e.g., Neo4j [5] and TITAN [7]. Meanwhile, the metadata will be generated from their original data and sent to the blockchain. In this way, the integrity of the query result obtained from the cloud can be verified based on the trusted ADS maintained by the blockchain.

To provide authenticated queries on graph data in blockchain-assisted clouds, several challenges lie ahead: (1) The ADS needs to be lightweight and can be easily maintained by the blockchain since the cost of storage and computation is high on the blockchain. Traditional methods like tree-based ADS and aggregate signatures, require much storage and excessive modification on updates since they do not support dynamic data. Hence, these methods are no longer applicable to the blockchain. (2) The query result usually has to satisfy the demands of soundness and completeness, the latter of which requires no valid answer missing, making the authentication design more difficult. Traditional *Merkle Hash Tree* (MHT) based ADS [48, 93] is incapable of providing proofs of non-existence for an element, e.g., proving the queried node has no outbound edges. (3) The cloud and the blockchain cannot reveal any extraneous information to users except the knowledge of the query itself as there may be some privacy-sensitive data that cannot be public.

To address these challenges, we design a novel ADS, named *Privacy-preserving Accumulator for Graphs in Blockchain* (PAGB), that provides authenticated queries on graph data in the blockchain-assisted cloud while guaranteeing the soundness and completeness of query results. We take the property graph [10, 25], one graph type represented by knowledge graph that has enormous potential in artificial intelligence [57], as an example to illustrate the effectiveness. In order to meet the demand of result completeness and privacy preserving, we construct a completeness set to extract

extraneous pairs from the original graph so that the non-existence of one item can be proved. On the blockchain side, we calculate the prime representative for each object as the metadata to update its accumulator, which also hinders the potential leakage of privacy. To further improve the efficiency, we optimize the solution by reducing the verification cost and accelerating the process of multiplication.

To the best of our knowledge, our PAGB is the first query authentication scheme for property graph data using blockchain while possessing the desirable characteristics of dynamics and privacy preserving. Our design can be applied to practical systems since the verification for the query user can be finished in several milliseconds based on an extra proof of hundreds of bytes. In addition, the gas consumption for ADS maintenance in blockchain is relatively low as well. We summarize the contributions made in this chapter as follows:

- We take the first step to formulate the authenticated query problem over property graph data in the scenario of the blockchain-assisted cloud.

- We design a privacy-preserving ADS called PAGB based on the completeness set and dynamic accumulator. We then propose authenticated query processing algorithms that satisfy the soundness and completeness for a number of queries.

- We further make the design more practical by incorporating non-interactive protocols and designing a binary tree multiplication method.

- We formally analyze the design and conduct extensive experiments to evaluate the performance, which demonstrates the effectiveness and efficiency of our design.

Table 4.1: Notations.

Symbol	Meaning
x, X	a prime number, and a prime number set
v, V	a node, and a node set
e, E	an edge, and an edge set
na, nav	node attribute, and node attribute value
ea, eav	edge attribute, and edge attribute value
λ	security parameter
$H_{prime}(\cdot)$	random function that computes a prime representative for the input
$\phi(\cdot)$	Euler totient function

The structure of this chapter is organized as follows. We first present the preliminaries in Section 4.2 and give the problem formulation in Section 4.3. We then describe the construction, maintenance and query processing of PAGB in Section 4.4 and further optimize the design in Section 4.5. We formally analyze our design in Section 4.6 and conduct the evaluation of our design to validate the performance in Section 4.7. Section 4.8 finally concludes this chapter.

4.2 Preliminaries

In this section, we briefly present some related preliminaries that will be used in our solution design. The notations we often use in this chapter are listed in Table 4.1.

4.2.1 Smart Contract

To trigger the execution of the smart contract, a blockchain user can send a transaction with input data to the contract address. Since the execution may involve

data storage in blockchain and computational operations, some fees will be charged by the miner. In the smart contract of Ethereum platform [81], the transaction cost is denominated in *gas* and a maximum gas limit, i.e., 8,000,000, is set to avoid excessive consumption of resources. The transaction cost is determined by the size of the compiled contract code and the execution cost run by the virtual machine, which consists of the storage of global variables and the runtime of functions. According to [81], storing a new value into the blockchain costs 20,000 gas while writing to the existing storage costs 5,000 gas. Since the storage cost in the smart contract is expensive, massive data storage is inapplicable to the blockchain.

4.2.2 Cryptographic Primitives

Cryptographic Hash Function. Given an arbitrary-size string as input, a cryptographic hash function can output a fixed-size hash value. It is infeasible for a probabilistic polynomial time (PPT) adversary to find the original string given the output value since the function is preimage resistant. The hash function is also collision resistant, which means it is difficult to find two different strings with an identical hash value.

Cryptographic Accumulator. A cryptographic accumulator is a one-way and collision resistant structure that maps a set to a fixed-size digest. According to the underlying contract implementation, a cryptographic accumulator can be commonly categorized into the RSA accumulator [15] and the bilinear-map accumulator [65]. The RSA accumulator employs modular exponentiation with an RSA modulus while the bilinear-map accumulator uses elliptic curve operations instead. One feature of the RSA accumulator, which we choose in this thesis, is that it can prove the membership and non-membership of an arbitrary element by generating the corresponding witness.

Let n be the RSA modulus and g be the generator. The accumulation value $\mathsf{Ac} = \mathsf{g}^{x_p} \bmod \mathsf{n}$, where X is the prime set and x_p is the product of all elements in X, i.e., $x_p = \prod_{x \in X} x$. The RSA accumulator can provide the following functions [53]:

- $\mathsf{Setup}(1^\lambda)$: It takes 1^λ as input and outputs a random λ-bit modulus n that satisfies: $n = pq$, where p and q are random safe primes. It also generates a generator $g \in QR_n \setminus \{1\}$, where QR_n is the group of quadratic residues modulo n.

- $\texttt{Accumulation}(X)$: It takes a set of prime numbers X as input and outputs the accumulation value Ac using $Ac = g^{x_p} \bmod n$, where x_p is the product of all numbers in X, i.e., $x_p = \prod_{x \in X} x$.

- $\texttt{MemWit}(x)$: It can generate a proof of existence for the member element. It outputs the membership witness for the element $x \in X$ as: $mw = g^{x_p/x} \bmod n$.

- $\texttt{VerifyMem}(x, mw)$: It can verify the validity of the membership witness. It takes the element x and the corresponding witness mw as input, and outputs `True` if $mw^x \bmod n$ equals Ac, otherwise outputs `False`.

- $\texttt{NonmemWit}(x)$: It can produce a proof to prove that one element does not exist in the set. If $x \notin X$, it will first find a and b such that $ax_p + bx = 1$, and then outputs a and $d = \mathsf{g}^b \bmod \mathsf{n}$ as the non-membership witness nmw.

- $\texttt{VerifyNonmem}(x, a, d)$: It can verify the validity of the non-membership witness. It takes the non-existent element x and its witness a, d as input, and outputs `True` if $\mathsf{Ac}^a d^x \bmod \mathsf{n}$ equals g, otherwise outputs `False`.

More elaborated construction and authentication of our accumulator will be presented in Section 4.4.

Non-interactive Proof of Exponentiation. Non-interactive Proof of Exponentiation (NI-PoE) is a succinct variant of Protocol PoE (Proof of Exponentiation) that can be utilized to improve the efficiency of verification [18]. Let $\mathbb{G}$ be a group and both the prover and verifier know (x, u, w). By reducing x to a residue r, it only takes constant time for the prover to convince the verifier that $w = u^x$ holds in $\mathbb{G}$. Specifically, NI-PoE has the following functions:

- `ProveNIPoE`(x, u, w): given x, u and w, it will first generate a prime number l based on (x, u, w). Then a quotient q will be calculated $q = \lfloor x/l \rfloor$ and a proof Q will be returned using $Q = u^q \bmod \mathtt{n}$.

- `VerifyNIPoE`(x, u, w, Q): given x, u, w and the proof Q, it will first generate the same prime number l using (x, u, w) and calculate a residue r as $r = x \bmod l$. It will return `True` if $Q^l u^r \bmod \mathtt{n}$ equals w, otherwise return `False`.

Non-interactive Proof of Knowledge of Exponent. Non-interactive Proof of Knowledge of Exponent (NI-PoKE2) is a non-interactive version of Protocol PoKE2 (Proof of Knowledge of Exponent) using the Fiat-Shamir heuristic [18]. It provides a succinct argument of knowledge of discrete logarithm by converting the exponent to some small items. Specifically, the verifier is only given (u, w) while the prover is able to convince the verifier that $w = u^x$ holds in $\mathbb{G}$. Since the prover does not tell the verifier the value of x, NI-PoKE2 can save the communication overhead of sending x. To achieve this, NI-PoKE2 has the following functions:

- `ProveNIPoKE2`(x, u, w): given x, u and w, it will first generate an element g in

group $\mathbb{G}$ based on (u, w) and another element $z = g^x \bmod \mathsf{n}$. Then a prime number l is generated based on (u, w, z) and a number α is produced using (u, w, z, l). Based upon l, a quotient q and a residue r are calculated using $q = \lfloor x/l \rfloor$ and $r = x \bmod l$ respectively. Finally, it will return a proof including $z, Q = ug^{\alpha q} \bmod \mathsf{n}, r$.

- `VerifyNIPoKE2`(u, w, z, Q, r): given u, w and the proof z, Q, r, it will produce the same g, l and α using the same methods as the prover. It will return `True` if $Q^l(ug^\alpha)^r \bmod \mathsf{n}$ equals $wz^\alpha \bmod \mathsf{n}$, otherwise return `False`.

4.3 Problem Formulation

In this section, we will formulate the problem of authenticated graph query on blockchain-assisted clouds.

4.3.1 System Model

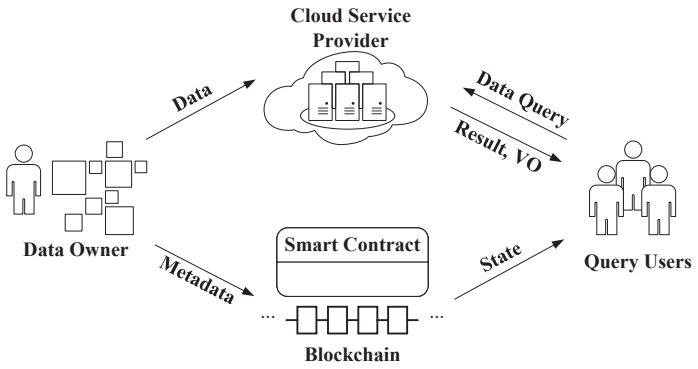


Fig. 4.1: Authenticated Query on Blockchain-assisted Clouds.

In this subsection, we introduce the framework of authenticated graph query in the hybrid storage of cloud and blockchain. As illustrated in Figure 4.1, our system

consists of four parties, i.e., a data owner (DO) who wishes to outsource his data to the cloud, a cloud service provider (CSP) who stores data and provides query services, a blockchain system (BC) that supports smart contract, and query users who issue authenticated queries to the cloud. When the DO has some data to insert or update, he will first calculate the metadata of the graph, which will be elaborated on in Section IV. Then the metadata will be sent to the BC and the CSP for the construction or update of ADS. In addition, the original graph data is outsourced to the CSP so that the query service on the data can be supported. Note that the ADS will be maintained by both the CSP and BC using the metadata. The ADS in the CSP is used to generate the *verification object* (VO) for the query user to verify, while the ADS in the BC can provide the trusted state due to the authenticity of smart contract. The data freshness guaranteed by BC may be weaker if the block confirmation time becomes longer, but the time cost, e.g., about 1 minute for Ethereum, is acceptable in most circumstances where data update is not frequent.

4.3.2 Property Graph

In this chapter, we focus on the property graph to illustrate our solution. But actually, our design is also applicable to any general directed graph. We model a property graph as $G = (V, E)$, where $V \subseteq \mathbb{N}$ is the node set and $E \subseteq V \times V$ is the edge set. Each node in V has a node property that contains one or more node attributes (e.g., name, URL, etc.), which can be normally represented via a set of key-value pairs. Thus, a node $v \in V$ can be denoted by $(id, [\langle na_k, nav_k \rangle])$, where id is the unique identifier and $[\langle na_k, nav_k \rangle]$ is the set of its node attribute na_k and corresponding attribute value nav_k. We use an ordered pair of nodes $e = (v_i, v_j) \in E$ signify a directed edge from node v_i to node v_j, where $v_i, v_j \in V$. Here v_i and v_j

are only the values of their identifiers. Each edge in E has the edge property, which, similar to the node property, is a sequence of edge attributes (e.g., the relation weight). Formally, we can denote the edge $e \in E$ as $(v_i, v_j, type, [\langle ea_k, eav_k \rangle])$, where $type$ is the edge type and $[\langle ea_k, eav_k \rangle]$ is the set of the edge attribute ea_k with respective value eav_k. Note that the value of the node attribute and edge attribute can be in many forms like a numerical value, a string value or even a list of objects [10], which we will not further discuss.

The authenticated queries on property graphs in this chapter include the following types:

- **node property query**: Users may wish to query about the property of a certain node. Specifically, a node property query can be in three forms as follows:
  - node property(id, na, nav): return whether node id has the attribute na with a value of nav.
  - node property(id, na): return the value of attribute na for node id.
  - node property(id): return all attributes $\{na_k\}$ and their values $\{nav_k\}$ that node id contains.
- **edge property query**: Query users can also get the property information about a certain edge. Analogously, we have the following forms of an edge property query:
  - edge property$(v_i, v_j, type, ea, eav)$: return whether the edge from v_i to v_j with $type$ has the attribute ea with a value of eav.

  - edge property$(v_i, v_j, type, ea)$: return the value of attribute ea for edge from v_i to v_j with $type$.
  - edge property$(v_i, v_j, type)$: return all attributes $\{ea_k\}$ and their values $\{eav_k\}$ that edge from v_i to v_j with $type$ contains.
  - edge property(v_i, v_j): return all relation types that edge from v_i to v_j has.

- **connectivity query**: In addition to property queries, users may hope to get the connectivity information of the graph. More detailedly, we have the following forms:
  - outbound$(v_i, [type_k], K)$: return all destination nodes that node v_i can reach within K steps while the type of each edge along the path is in $[type_k]$.
  - inbound$(v_j, [type_k], K)$: return all source nodes that can reach node v_j within K steps while the type of each edge along the path is in $[type_k]$.

We use an example of the knowledge graph shown in Figure 4.2 to illustrate the authenticated query problem on graphs. We assume that there are four entity nodes, i.e., *Liverpool*, *George Harrison*, *John Lennon* and *The Beatles*, as presented in green boxes. Each node has a unique identifier *id* and two related attributes together with respective values. The directed edges linking the nodes, which are marked in blue boxes, represent the relation between these entities. Each edge uses a *type* identifier to show the detailed relation. In addition, each relation edge is described by two attributes, i.e., a *source* attribute to denote the knowledge source and a *weight* attribute to express the strength of this assertion.

When a user issues a request of node property query like node property(‘*John*

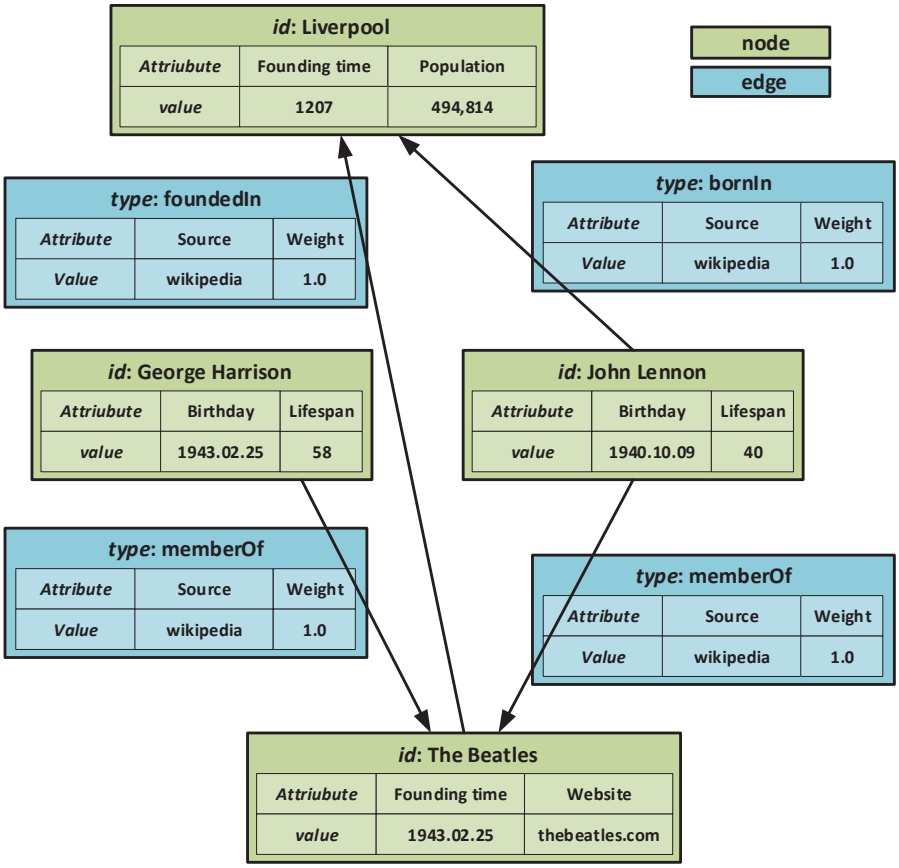


Fig. 4.2: An Illustrative Example of Knowledge Graph.

Lennon', '*Birthday*', '*1940.10.09*'), the result he gets is `True`. If the value of birthday he sends is wrong, the result will be `False`. A user may also send node property('*The Beatles*', '*Website*') to get the value of The Beatle's website, but a query like node property('*The Beatles*', '*Birthday*') will receive a result that the specified attribute does not exist. We can find all attributes of node *Liverpool*, i.e., {'*Founding time*', '*Population*'}, with the query of node property('*Liverpool*'). The edge property queries are quite similar to the node property queries, thus they can be formed likewise. Specifically, a user can query about all relation types, i.e., *memberOf*, from the node *John Lennon* and *The Beatles* using edge property('*John Lennon*, *The Beatles*').

For the connectivity query, users may get the node *The Beatles* and *Liverpool* from *George Harrison* by using outbound('*George Harrison*', {'*memberOf*', '*foundedIn*'}, 2). Similarly, we can find the two members of The Beatles, i.e., *George Harrison* and *John Lennon*, through the query of inbound({'*memberOf*'}, '*The Beatles*', 1). It is noted that all results mentioned above should be accompanied with a corresponding VO for verification.

4.3.3 Threat Model

In our hybrid-storage system, we model the DO and BC as fully trusted parties. The query users are assumed to be curious about the outsourced data, which means they may attempt to learn extraneous information apart from query results. The CSP does not care about the data and will not leak it to others. However, the CSP is considered untrusted as it may be dishonest about query results. This can arise from the consideration of resource saving, hardware errors, malware or media failures. To this end, the untrusted CSP needs to provide a VO, which contains the verification information, along with its query result. After examining the state of ADS in the BC, the query user can validate the soundness and completeness of the query result using VO:

- **Soundness.** All objects in the result conform to the query conditions and originate from the DO.
- **Completeness.** No valid object is missing from the result set.

The objective problem in this chapter is how to provide query processing and result verification for graph queries based on our system model and threat model. Meanwhile, we need to achieve the following goals when designing the ADS:

- **Dynamics.** The data dynamics including the operations of insertion, deletion and update on the outsourced data should be supported.

- **Privacy preserving.** We assume the CSP in our model is trusted for privacy. The BC is trusted but may passively leak information since it is publicly accessible. Therefore, the original data can only be retrieved from the DO and CSP, otherwise privacy leakage occurs. In addition, the query user cannot get any extraneous information of the graph except the knowledge required by the query itself.

- **Efficiency.** The efficiency is embodied in the following three aspects. 1) Gas efficiency: It costs a reasonable amount of gas for the smart contract to maintain the ADS accommodated in the BC. 2) Communication efficiency: The verification scheme to guarantee the data integrity incurs small network bandwidth usage, i.e., the size of VO is small. 3) Computational efficiency: The time cost of ADS maintenance, VO generation and VO verification is acceptable.

4.4 PAGB Design for Graph Data

In this section, we first present the construction and maintenance of the proposed PAGB. We then illustrate the authenticated query processing for target query problems and perform security analysis on our design.

4.4.1 PAGB Construction

Our PAGB construction mainly consists of four phases, i.e., setup, completeness construction, prime representation and accumulation. As shown in Algorithm 5, the construction involves the participation of the DO, CSP and BC. The DO executes all

four phases locally during the construction and then gives the constructed ADS to the BC. The CSP also does the accumulation to maintain the ADS.

Algorithm 5: PAGB Construction on Graph Data

Input: security parameter λ, node set V, edge set E;

DO
- $(\mathsf{p}, \mathsf{q}, \mathsf{n}, \mathsf{g}) \leftarrow \mathtt{Setup}(1^\lambda)$; $X \leftarrow \phi$;
- $C \leftarrow \mathtt{CompletenessConstruction}(V, E)$;
- **for** $v \in V$ **do**
  - $X \leftarrow X \cup H_{prime}(v)$;
- **for** $e \in E$ **do**
  - $X \leftarrow X \cup H_{prime}(e)$;
- **for** $c \in C$ **do**
  - $X \leftarrow X \cup H_{prime}(c)$;
- $\mathsf{Ac} \leftarrow \mathtt{Accumulation}(X)$;
- Send n, g, X, V, E, C to CSP;
- Send n, g, X, Ac to BC;

CSP
- Store V, E into the graph database;
- $\mathsf{Ac}^* \leftarrow \mathtt{Accumulation}(X)$;
- Send Ac^* to BC;

BC
- Check whether Ac equals Ac^*;
- Record n, g and Ac via the smart contract;

Setup. The goal of the setup phase is to generate the public key and secret key used in PAGB. It is initially conducted by the DO and only conducted once. The DO invokes the key generation algorithm $\mathtt{Setup}(1^\lambda)$, where λ is the security parameter, to get the system parameters, i.e., p, q, n and g. Among the four parameters, p and q belong to the secret key and are generated and kept only by the DO. The remaining n and g are the public key and available to all parties.

Completeness Construction. The completeness construction phase aims to generate completeness elements so that the information leakage of each query can be limited to the answer itself when guaranteeing the result completeness. For instance, suppose we have already stored node $v = (id, [\langle na_k, nav_k \rangle])$ into PAGB. Then a user wishes to get the attribute value of na in node v by issuing a node property query, i.e., $\mathsf{node\ property}(id, na)$. But actually node v does not have the attribute na. Intuitively, the CSP needs to return v and its membership witness as the VO to convince the query user of the non-existence of na. As a consequence, all information of v is revealed to the query user, whereas all attributes of v are not relevant to the query. The user can get extraneous knowledge about the graph data, which violates the principle of privacy preserving. To this end, the CSP needs to give (id, na) and its non-membership as the VO so that the features of result completeness and privacy preserving can both be guaranteed. In order to further cover all query types, we propose the completeness construction algorithm as depicted in Algorithm 6.

This algorithm creates a completeness set C to hold all completeness elements required by the authenticated query types listed in Section 3.2. It extracts the identifier id, each attribute (id, na) and each attribute-value pair (id, na, nav) for every node. Similarly, for every edge, the identifier $(v_i, v_j, type)$, each attribute $(v_i, v_j, type, ea)$ and each attribute-value pair $(v_i, v_j, type, ea, eav)$ are also added to the set. Moreover, we also record the pair (v_i, v_j), the node outbound type $(v_i, \mathsf{out}, type)$, the node inbound type $(v_j, \mathsf{in}, type)$, all types per pair $(v_i, v_j, [type_k])$, all nodes per outbound type $(v_i, \mathsf{out}, type, [v_k])$ and all nodes per inbound type $(v_j, \mathsf{in}, type, [v_k])$.

Prime Representation. As in [53], the input domain of the accumulator must be prime numbers. Since the sets of nodes, edges and completeness generated in the last

Algorithm 6: Completeness Construction

Input: node set V, edge set E;

Function `CompletenessConstruction`(V, E)

- $C \leftarrow \phi$; $Dic \leftarrow \phi$;
- **for** $v = (id, [\langle na_k, nav_k \rangle]) \in V$ **do**
  - $C \leftarrow C \cup id$;
  - **for** $\langle na, nav \rangle \in [\langle na_k, nav_k \rangle]$ **do**
    - $C \leftarrow C \cup (id, na) \cup (id, na, nav)$;
- **for** $e = (v_i, v_j, type, [\langle ea_k, eav_k \rangle]) \in E$ **do**
  - $C \leftarrow C \cup (v_i, v_j) \cup (v_i, v_j, type)$;
  - **for** $\langle ea, eav \rangle \in [\langle ea_k, eav_k \rangle]$ **do**
    - $C \leftarrow C \cup (v_i, v_j, type, ea) \cup (v_i, v_j, type, ea, eav)$;
  - $Dic[(v_i, v_j)]$.append$(type)$;
  - $Dic[(v_i, \mathsf{out}, type)]$.append$(v_j)$;
  - $Dic[(v_j, \mathsf{in}, type)]$.append$(v_i)$;
  - $C \leftarrow C \cup (v_i, \mathsf{out}, type) \cup (v_j, \mathsf{in}, type)$;
- **for** $key, value \in Dic$ **do**
  - $C \leftarrow C \cup (key, value)$;
- **return** C;

phase are all arbitrary tuples of several objects, we need to map these tuples to prime numbers through the prime representation phase. Some solutions to solve this have been proposed in prior works like [35, 37]. In this thesis, we generate the random prime representative through a random oracle algorithm $H_{prime}(\cdot)$, which computes a prime representative for the input. It first concatenates the object in order with reserved notations and converts it into a string, based on which a hash value can be produced. The hash can be transformed to a corresponding integer and the output prime is the smallest prime not less than this integer.

Accumulation. The accumulation phase is aimed to generate the ADS state, i.e., the accumulation value of the accumulator, for query authentication. The algorithm first computes the product of all elements and then calculates the accumulation value using $\mathsf{Ac} = \mathsf{g}^{\prod_{x \in X} x} \bmod \mathsf{n}$. On the BC side, it will produce Ac and directly give it to the BC for further maintenance. As for the CSP side, the algorithm will first store the node set and edge set into the graph database for normal search. Then he also computes his own ADS state Ac^* and sends it to the BC. After receiving the two accumulation values from the DO and the CSP, the BC will check the validity of the CSP's Ac^* by comparing it with the trusted Ac from the DO. The BC will store Ac on the smart contract as well as the public key n and g if the validity passes. Note that $\phi(\mathsf{n}) = (\mathsf{p}-1) * (\mathsf{q}-1)$ can be utilized on the BC side to speed up the accumulation via $\mathsf{Ac} = \mathsf{g}^{\prod_{x \in X} x \bmod \phi(\mathsf{n})} \bmod \mathsf{n}$.

4.4.2 PAGB Maintenance

In this subsection, we present the algorithms for PAGB maintenance, including graph insertion, deletion and update.

Insertion. When the DO has some graph data to add, which may contain nodes

Algorithm 7: PAGB Insertion or Deletion

Input: node to add or delete v, edge to add or delete e;

Graph Data Insert or Delete (by DO)

- $X_{add} \leftarrow \phi$; $X_{del} \leftarrow \phi$;;
- **if** *Add* **then**
  - $(C_{add}, C_{del}) \leftarrow$ `CompletenessInsert(`v, e`)`;
  - $X_{add} \leftarrow X_{add} \cup H_{prime}$`(`$v$`)` $\cup H_{prime}$`(`e`)`;
- **if** *Delete* **then**
  - $(C_{add}, C_{del}) \leftarrow$ `CompletenessDelete(`v, e`)`;
  - $X_{del} \leftarrow X_{del} \cup H_{prime}$`(`$v$`)` $\cup H_{prime}$`(`e`)`;
- **for** $c \in C_{add}$ **do**
  - $X_{add} \leftarrow X_{add} \cup H_{prime}$`(`$c$`)`;
- **for** $c \in C_{del}$ **do**
  - $X_{del} \leftarrow X_{del} \cup H_{prime}(c)$;
- Send $X_{add}, X_{del}, v, e, C_{add}, C_{del}$ to CSP;
- Send X_{add}, X_{del} to BC;

Authenticated Query Insert or Delete (by CSP)

- **if** *Add* **then**
  - $V \leftarrow V \cup v$; $E \leftarrow E \cup e$;
- **if** *Delete* **then**
  - $V \leftarrow V \setminus v$; $E \leftarrow E \setminus e$;
- $C \leftarrow C \cup C_{add}$; $C \leftarrow C \backslash C_{del}$;
- $X \leftarrow X \cup X_{add}$; $X \leftarrow X \backslash X_{del}$;
- $\mathsf{Ac}^* \leftarrow \mathsf{g}^{\prod_{x \in X} x} \bmod \mathsf{n}$;
- Send Ac^* to BC;

PAGB Insert or Delete (by BC)

- **if** $\mathsf{Ac}^{* \prod_{x \in X_{del}} x} == \mathsf{Ac}^{\prod_{x \in X_{add}} x} \bmod \mathsf{n}$ **then**
  - $\mathsf{Ac} \leftarrow \mathsf{Ac}^*$;

and edges, the original data will be sent to the CSP. Meanwhile, the metadata is computed by the DO and then sent to the CSP and BC so that their accumulation values can be updated. The process of PAGB insertion is described in Algorithm 7. Due to the newly added node v and edge e, the DO will invoke the function `CompletenessInsert`(v, e) to update the completeness set. This function computes the completeness elements that will be added to and removed from the accumulator. For v, e and the elements to be added and deleted, i.e., C_{add} and C_{del}, we will find their prime representatives and then send them to the CSP and BC. Based on the primes X_{add} and X_{del}, the CSP will update his accumulator as $\mathsf{Ac}^* \leftarrow \mathsf{g}^{\prod_{x \in X \cup X_{add} \setminus X_{del}} x} \bmod \mathsf{n}$ and send Ac^* to the BC for check. After receiving the primes and the CSP's accumulation value, the BC will validate Ac^* by checking $\mathsf{Ac}^{*\prod_{x \in X_{del}} x} \stackrel{?}{=} \mathsf{Ac}^{\prod_{x \in X_{add}} x} \bmod \mathsf{n}$, and accept it if it is correct.

The completeness insertion function, as shown in Algorithm 8, is aimed to replace the elements in the completeness set due to the newly inserted node and edge. In another word, the algorithm will delete old completeness elements and add new ones. The elements to be deleted are stored in C_{del}. These elements consist of all types per pair $(v_i, v_j, [type_k])$, all nodes per outbound type $(v_i, \mathsf{out}, type, [v_k])$ and all nodes per inbound type $(v_j, \mathsf{in}, type, [v_k])$. They become stale as the new edge e is inserted. As the elements to be added, i.e., C_{add}, they contain all updated completeness elements, which is almost the same as the construction in Algorithm 6.

Deletion. The deletion procedure of PAGB is similar to the insertion operation as shown in Algorithm 7. The main difference is that the DO will generate the change of completeness set using `CompletenessDelete`(v, e). The process of completeness deletion is presented in Algorithm 9. For the part of attributes that node v and edge

Algorithm 8: Completeness Insertion

Input: node to add v, edge to add e;

Function `CompletenessInsert`(v, e)

$C_{add} \leftarrow \phi$; $C_{del} \leftarrow \phi$;
$C_{add} \leftarrow C_{add} \cup id$;
for $\langle na, nav \rangle \in [\langle na_k, nav_k \rangle]$ **do**
  $C_{add} \leftarrow C_{add} \cup (id, na) \cup (id, na, nav)$;
$C_{add} \leftarrow C_{add} \cup (v_i, v_j) \cup (v_i, v_j, type)$;
for $\langle ea, eav \rangle \in [\langle ea_k, eav_k \rangle]$ **do**
  $C_{add} \leftarrow C_{add} \cup (v_i, v_j, type, ea) \cup (v_i, v_j, type, ea, eav)$;
$C_{del} \leftarrow C_{del} \cup (v_i, v_j, C[(v_i, v_j)])$;
$C[(v_i, v_j)]$.append$(type)$;
$C_{add} \leftarrow C_{add} \cup (v_i, v_j, C[(v_i, v_j)])$;
$C_{del} \leftarrow C_{del} \cup (v_i, \mathsf{out}, type, C[(v_i, \mathsf{out}, type)])$;
$C[(v_i, \mathsf{out}, type)]$.append$(v_j)$;
$C_{add} \leftarrow C_{add} \cup (v_i, \mathsf{out}, type, C[(v_i, \mathsf{out}, type)])$;
$C_{del} \leftarrow C_{del} \cup (v_j, \mathsf{in}, type, C[(v_j, \mathsf{in}, type)])$;
$C[(v_j, \mathsf{in}, type)]$.append$(v_i)$;
$C_{add} \leftarrow C_{add} \cup (v_j, \mathsf{in}, type, C[(v_j, \mathsf{in}, type)])$;
$C_{add} \leftarrow C_{add} \cup (v_i, \mathsf{out}, type) \cup (v_j, \mathsf{in}, type)$;
$C \leftarrow C \cup C_{add}$; $C \leftarrow C \backslash C_{del}$;
return (C_{add}, C_{del})

e have, we can simply delete them from the completeness set. As for the remaining three categories, that is all types per pair $(v_i, v_j, [type_k])$, all nodes per outbound type $(v_i, \mathsf{out}, type, [v_k])$ and all nodes per inbound type $(v_j, \mathsf{in}, type, [v_k])$, besides reducing the lists, we also need to remove the element of (v_i, v_j), $(v_i, \mathsf{out}, type)$ and $(v_j, \mathsf{in}, type)$ if their corresponding lists become empty.

Update. Intuitively, the update operation can be seen as inserting a new node or edge after deleting the old one. Hence, the update of PAGB and completeness set can be easily achieved by successively invoking the corresponding deletion and insertion algorithms, which we will not further discuss.

4.4.3 Authenticated Query Processing

In this subsection, we describe the query processing and result verification of the target graph queries, i.e., node property query, edge property query and connectivity query. It is noted that we only address the authentication challenges in this chapter. The storage of graph data in the databases and the efficiency of data query are beyond the scope of our work since there are already some mature graph databases available.

At a high level, the design rationale of the authenticated query processing is to give answers of the exact level that the query requires. If the object exists, then the membership is returned. Otherwise, the non-existence is given without revealing other additional information. We also define leakage functions revealed by corresponding algorithms.

Node Property Query. For the node property query, we provide the following fine-grained queries.

- $\mathsf{node\ property}(id, na, nav)$: If node id has the attribute na with a value of

Algorithm 9: Completeness Deletion

Input: node to delete v, edge to delete e;

Function `CompletenessDelete`(v, e)

$C_{add} \leftarrow \phi$; $C_{del} \leftarrow \phi$;
$C_{del} \leftarrow C_{del} \cup id$;
for $\langle na_k, nav_k \rangle \in [\langle na_k, nav_k \rangle]$ **do**
  $C_{del} \leftarrow C_{del} \cup (id, na) \cup (id, na, nav)$;
$C_{del} \leftarrow C_{del} \cup (v_i, v_j, type)$;
for $\langle ea, eav \rangle \in [\langle ea_k, eav_k \rangle]$ **do**
  $C_{del} \leftarrow C_{del} \cup (v_i, v_j, type, ea) \cup (v_i, v_j, type, ea, eav)$;
$C_{del} \leftarrow C_{del} \cup (v_i, v_j, C[(v_i, v_j)])$;
$C[(v_i, v_j)]$.remove($type$);
if `len`$(C[(v_i, v_j)])$ $==$ *0* **then**
  $C_{del} \leftarrow C_{del} \cup (v_i, v_j)$
else
  $C_{add} \leftarrow C_{add} \cup (v_i, v_j, C[(v_i, v_j)])$;
$C_{del} \leftarrow C_{del} \cup (v_i, \mathsf{out}, type, C[(v_i, \mathsf{out}, type)])$;
$C[(v_i, \mathsf{out}, type)]$.remove($v_j$);
if `len`$(C[(v_i, \mathsf{out}, type)])$ $==$ *0* **then**
  $C_{del} \leftarrow C_{del} \cup (v_i, \mathsf{out}, type)$
else
  $C_{add} \leftarrow C_{add} \cup (v_i, \mathsf{out}, type, C[(v_i, \mathsf{out}, type)])$;
$C_{del} \leftarrow C_{del} \cup (v_j, \mathsf{in}, type, C[(v_j, \mathsf{in}, type)])$;
$C[(v_j, \mathsf{in}, type)]$.remove($v_i$);
if `len`$(C[(v_j, \mathsf{in}, type)])$ $==$ *0* **then**
  $C_{del} \leftarrow C_{del} \cup (v_j, \mathsf{in}, type)$
else
  $C_{add} \leftarrow C_{add} \cup (v_j, \mathsf{in}, type, C[(v_j, \mathsf{in}, type)])$;
$C \leftarrow C \cup C_{add}$; $C \leftarrow C \backslash C_{del}$;
return (C_{add}, C_{del})

nav, the CSP returns the result `True` along with the membership witness of $H_{prime}(id, na, nav)$ for verification. Otherwise, it will return the result `False` and the non-membership witness of $H_{prime}(id, na, nav)$. The leakage function is $\mathcal{L}_{n1} = (True, False)$.

- node property(id, na): If node id has the attribute na, the CSP returns the value of attribute na for node id as the result. In addition, a membership witness of $H_{prime}(id, na, nav)$ will be sent for verification. If the attribute na does not exist in node id, it will report the non-existence and give the non-membership witness of $H_{prime}(id, na)$ as the VO. The leakage function is $\mathcal{L}_{n2} = (nav, \phi)$.

- node property(id): If node id exists, the CSP returns all attribute-value pairs $[\langle na_k, nav_k \rangle]$ that node id contains and the membership witness of $H_{prime}(v)$ for verification. Otherwise, it will report the non-existence of id and the non-membership witness of $H_{prime}(id)$. The leakage function is $\mathcal{L}_{n3} = ([\langle na_k, nav_k \rangle], \phi)$.

Edge Property Query. Similarly, we also support the following authenticated queries for edge property problems.

- edge property$(v_i, v_j, type, ea, eav)$: If edge from v_i to v_j with $type$ has the attribute ea with a value of eav, the CSP returns the result `True` along with the membership witness of $H_{prime}(v_i, v_j, type, ea, eav)$ as the VO. Otherwise, it will give the result `False` and the non-membership witness of $H_{prime}(v_i, v_j, type, ea, eav)$. The leakage function is $\mathcal{L}_{e1} = (True, False)$.

- edge property$(v_i, v_j, type, ea)$: If edge from v_i to v_j with $type$ has the attribute ea, the CSP returns the value of attribute ea for edge $(v_i, v_j, type)$ as the result. In addition, a membership witness of $H_{prime}(v_i, v_j, type, ea, eav)$ will be sent for

verification. If the attribute ea does not exist in edge $(v_i, v_j, type)$, it will report the non-existence and give the non-membership witness of $H_{prime}(v_i, v_j, type, ea)$ for verification. The leakage function is $\mathcal{L}_{e2} = (eav, \phi)$.

- edge property$(v_i, v_j, type)$: If edge from v_i to v_j with $type$ exists, the CSP returns all attribute-value pairs $[\langle ea_k, eav_k \rangle]$ that edge $(v_i, v_j, type)$ contains and the membership witness of $H_{prime}(e)$ for verification. Otherwise, it will report the non-existence of $(v_i, v_j, type)$ and the non-membership witness of $H_{prime}(v_i, v_j, type)$. The leakage function is $\mathcal{L}_{e3} = ([\langle ea_k, eav_k \rangle], \phi)$.

- edge property(v_i, v_j): If edge from v_i to v_j exists, the CSP returns all relation types that edge (v_i, v_j) contains and the membership witness of $H_{prime}(v_i, v_j, [type_k])$. Otherwise, it will report the non-existence of (v_i, v_j) and the non-membership witness of $H_{prime}(v_i, v_j)$. The leakage function is $\mathcal{L}_{e4} = ([type_k], \phi)$.

Note that the CSP will return the result and corresponding VO of just one single witness for both node property query and edge property query. To verify the result, the query user only needs to generate the prime representative of the tuple and check the correctness of the witness using the aforementioned `VerifyMem` or `VerifyNonmem`.

Connectivity Query. Different from single witness in the prior two queries, the connectivity query often produces a result of node set and the corresponding VO of witness set. We take the outbound query as an example to illustrate our algorithms.

- outbound$(v_i, [type_k], K)$: As described in Algorithm 10, we propose the `OutboundQuery` function that will be executed by the CSP to generate the result $NodeSet$ and the VO $ProofSet$ based on the depth-first search. For each edge type in the desired type list $[type_k]$, the original node v will check whether it has such an

outbound relation type. On one hand, if node v has such an outbound edge, the destination nodes will be added to the result set. Meanwhile, the tuple (v, out, $type, C[(v, \mathsf{out}, type)]$) and its membership witness are added to the VO for verification. As for the destination nodes in the successor set, when K is larger than 1, which means further search is required, the $\mathsf{OutboundQuery}$ function will be invoked recursively by each node with parameter K changing to $K-1$. On the other hand, if v does not have the desired type, the authenticated query needs to return the tuple (v, out, $type$) along with its non-membership witness. The leakage function is $\mathcal{L}_{c1} = ([\langle v_k, [(v_k, \mathsf{out}, [type_k])]\rangle])$, where v_k is any node that initial node v_i can reach along the way and $[(v_k, \mathsf{out}, [type_k])]$ means the node list that v_k connects to via outbound edges with $[type_k]$.

Algorithm 11 shows the `VerifyOutbound` function for the query user side to validate the result and VO. Similar to the CSP, it will traverse the nodes from v to get the set of nodes by calling the recursive function `GetOutbound`. If the set is identical to the result $NodeSet$ returned by the CSP, then the result will be accepted. In the function `GetOutbound`, for each desired type relation, the user will find out whether the original node has the successor nodes. If not, it will verify the validity of the non-membership witness using `VerifyNonmem`. Otherwise, the user can verify the membership witness with `VerifyMem` and then add the destination nodes to the search list. If K is greater than 1, the traversal will proceed from each successor node.

- $\mathsf{inbound}(v_j, [type_k], K)$: the procedure is almost the same as the $\mathsf{outbound}$ except changing out to in. Accordingly, the leakage function is $\mathcal{L}_{c2} = ([\langle v_k, [(v_k, \mathsf{in}, [type_k])]\rangle])$.

Algorithm 10: Authenticated Outbound Query (by CSP)

Input: initial node v, a set of desired types $[type_k]$, step number K;

```
1  Function OutboundQuery(v, [type_k], K)
2      NodeSet ← φ;  ProofSet ← φ;
3      for type ∈ [type_k] do
4          if C[(v, out, type)] ≠ φ then
5              NodeSet ← NodeSet ∪ C[(v, out, type)];
6              o ← (v, out, type, C[(v, out, type)]);
7              if o has not been checked then
8                  x ← H_prime(o);
9                  mw ← MemWit(x);
10                 ProofSet ← ProofSet ∪ ⟨o, mw⟩;
11             if K > 1 then
12                 for node ∈ C[(v, out, type)] do
13                     (nSet, pSet) ← OutboundQuery(node, [type_k], K − 1);
14                     NodeSet ← NodeSet ∪ nSet;
15                     ProofSet ← ProofSet ∪ pSet;
16         else
17             o ← (v, out, type);
18             if o has not been checked then
19                 x ← H_prime(o);
20                 nmw ← NonmemWit(x);
21                 ProofSet ← ProofSet ∪ ⟨o, nmw⟩;
22     return NodeSet, ProofSet;
```

Algorithm 11: Result Verification of Outbound Query (by Query User)

Input: initial node v, a set of desired types $[type_k]$, step number K, result Set $NodeSet$, VO $ProofSet$, accumulation value Ac from BC;

```
1  Function VerifyOutbound(v, [type_k], K, NodeSet, ProofSet, Ac)
2      nSet ← GetOutbound(node, [type_k], K, ProofSet, Ac);
3      Check whether nSet equals NodeSet;
4  Function GetOutbound(v, [type_k], K, ProofSet, Ac)
5      nSet ← φ;
6      for type ∈ [type_k] do
7          Find ⟨o, mw or nmw⟩ regarding (v, out, type) in ProofSet;
8          x ← H_prime(o);
9          if o is (v, out, type) then
10             VerifyNonmem(x, nmw, Ac);
11         else
12             VerifyMem(x, mw, Ac);
13             nSet ← nSet ∪ C[(v, out, type)];
14             if K > 1 then
15                 for node ∈ C[(v, out, type)] do
16                     nSet ← nSet ∪
                       GetOutbound(node, [type_k], K − 1, ProofSet, Ac);
17     return nSet;
```

4.5 Optimization

In this section, we propose two optimization methods, i.e., batch verification and product calculation, to reduce time cost and network overhead in practice.

4.5.1 Batch Verification

When processing some authenticated queries, e.g., connectivity queries, the CSP needs to return the query result along with the VO, which consists of numerous membership and non-membership witnesses. The large amount of witnesses not only brings too much communication cost, but also takes a lot of time for the query users to verify the witnesses one by one. Inspired by [18], we utilize the batch technique to improve the efficiency of query processing.

Membership Witness. Assume we have a prime number set X that contains all elements in the accumulator and a subset $X_A \subseteq X$ is the set that we need to prove its membership. The product of all elements in X is signified as $x_p = \prod_{x \in X} x$. Instead of calculating the witness for each element in X_A, we can directly compute the product of these elements, denoted as $x_a = \prod_{x \in X_A} x$, and provide the witness of this product. In detail, through the function $\texttt{MemWit}(X_A)$, we can get the witness $mw = \mathsf{g}^{x_p/x_a} \bmod \mathsf{n}$ for X_A. Using this witness, the query user can verify the membership by the function $\texttt{VerifyMem}(x_a, mw)$, i.e., whether Ac equals $mw^{x_a} \bmod \mathsf{n}$. When the amount of elements in X_A gets larger, the value of x_a rises and it takes more time for the user to calculate mw^{x_a}.

We hereby employ the aforementioned NI-PoE to reduce the computation complexity of the query user. The CSP can invoke $\texttt{ProveNIPoE}(x_a, mw, \mathsf{Ac})$ to generate the proof Q and send it to the query user. After receiving the proof, the user can

call the verification function $\texttt{VerifyNIPoE}(x_a, mw, \mathsf{Ac}, Q)$ to check the validity. This NI-PoE method only needs an extra λ-bit of proof Q to avoid many modular exponentiation operations for the query users.

Non-membership Witness. Different from the membership witness, i.e., a λ-bit integer, the non-membership witness is a proof that consists of two elements. Assume we have another prime number set X_B that $X_B \cup X = \phi$, and the product of all elements in X_B is x_b, i.e., $x_b = \prod_{x \in X_B} x$. Since x_p and x_b are co-prime, the witness generation first calculates the Bezout coefficients $a, b \in \mathbb{Z}$ satisfying the relation $ax_p + bx_b = 1$. Then the CSP will return a and $d = \mathsf{g}^b \bmod \mathsf{n}$ as the witness and the query user needs to check whether $\mathsf{Ac}^a d^{x_b} \bmod \mathsf{n}$ equals g. Suppose the number of elements in X_B is $|X_B|$ and each element is a s-bit prime integer. As to the batch verification, the coefficient a is $|X_B|s$-bit and gets larger when $|X_B|$ rises, which leads to a big size of VO and much computation complexity for the user to check.

To avoid transmitting the coefficient a from the CSP to the user, we exploit NI-PoKE2 method to prove the validity of Ac^a to the user without telling a. Specifically, the CSP calls the function $\texttt{ProveNIPoKE2}(a, \mathsf{Ac}, \mathsf{Ac}^a)$ to generate the non-interactive proof (z, Q, r) and send it to the query user. The correctness of Ac^a will then be validated by the user using $\texttt{VerifyNIPoKE2}(\mathsf{Ac}, \mathsf{Ac}^a, z, Q, r)$. With the value of Ac^a, the non-membership witness can be verified easily. According to the function $\texttt{ProveNIPoKE2}$, the size of proof (z, Q, r) is $s + 2\lambda$ bits, which is much smaller than $|X_B|s$ bits. This can help maintaining the constant size of VO and reducing a lot of modular exponentiation operations during the verification phase.

Since the size of x_b itself is very large, it is costly to calculate d^{x_b} as well. Based on the correct Ac^a, we can further employ NI-PoE to save the computation

time. Specifically, the CSP can invoke $\texttt{ProveNIPoE}(x_b, d, \mathsf{g}(\mathsf{Ac}^a)^{-1})$ to convince the query user that $\mathsf{Ac}^a d^{x_b} = \mathsf{g}$ holds with the proof Q. The query user then calls $\texttt{VerifyNIPoE}(x_b, d, \mathsf{g}(\mathsf{Ac}^a)^{-1}, Q)$ to validate the result.

Batch Algorithms. With the batch verification techniques, some minor modification on the authenticated query processing of connectivity query is needed. In Algorithm 10, the tuples and their prime representatives are still produced, but the membership or non-membership witness will not be calculated individually by `MemWit` nor `NonmemWit`. Instead, all membership primes and non-membership primes will be multiplied and aggregated into one witness respectively. As for the result verification in Algorithm 11, it will only check the correctness of the tuples and perform the search in advance. The validity of the membership witness and the non-membership witness will be verified at the last step using the batch verification.

4.5.2 Product Calculation

During the setup phase of the accumulator, we need the product of the representatives of elements, i.e., the prime numbers, to do the accumulation. Meanwhile, the batch verification also involves the product computation of the batch prime number set. Intuitively, we can multiply them one by one to calculate the product of these elements. The underlying multiplication method can be the grade-school multiplication or the Karatsuba algorithm, which is a fast multiplication algorithm of $O(n^{1.58})$ complexity for two large integers. However, this successive multiplication method proceeds quite slowly when it comes to a dataset of large size regardless of the multiplication method. This is because the prior product will contain too many bits and the next element is much smaller, degrading the performance of the Karatsuba algorithm. To this end, we propose a binary tree multiplication method to expedite

the process of product computation.

Our method will first expand the prime number set X to the size that equals the smallest 2^n, i.e., power of 2, not less than $|X|$ and pad the set with integer 1. Hence, the elements can be transformed into the form of binary tree and the multiplication operations can be conducted from the bottom to the top to produce the final result. Figure 4.3 shows an example of our method working on a prime number set of 7 elements. The total amount is expanded from 7 to 8 by padding the set with one default value 1, i.e., the last leaf marked dark in the figure. When the two integers are both large, e.g., p_{12} and p_{34}, the Karatsuba algorithm works faster than the grade-school multiplication. The efficiency improvement becomes more notable as the two multipliers get larger. Therefore, we use the grade-school multiplication at the low layers, e.g., the leaf layer, and employ the Karatsuba algorithm for the upper layers (marked in green).

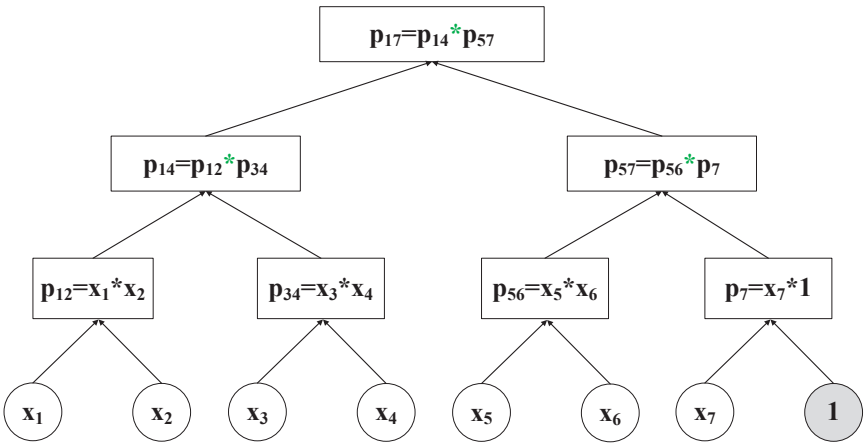


Fig. 4.3: Binary Tree Multiplication.

4.6 Design Analysis

4.6.1 Security Analysis

We first give the formal definition of security for our ADS and authenticated algorithms.

Definition 1 (Secure)**.** *The authenticated graph query algorithms on ADS is sound and complete if for any PPT adversary $\mathcal{A}$, the probability to succeed in the following experiment is negligible.*

- *$\mathcal{A}$ chooses a graph dataset D. The algorithms generate the ADS and the corresponding state $\boldsymbol{Ac}$ via D and give them to $\mathcal{A}$;*
- *In response to a query Q, $\mathcal{A}$ returns a result R and a proof VO to the query user. $\mathcal{A}$ conducts a successful attack if the VO passes the verification using $\boldsymbol{Ac}$ and R meets: $\{o|o \notin Q(D) \wedge o \in R\} \neq \phi \vee \{o|o \in Q(D) \wedge o \notin R\} \neq \phi$.*

This property indicates that it is infeasible for a malicious CSP to convince the query user if the result is incorrect or incomplete. We then prove that our algorithms achieve this security property.

Lemma 1. *The construction of our underlying RSA accumulator is secure.*

Proof. The proof of the security of our underlying RSA accumulator is omitted here, as it has been well investigated under the strong RSA assumption in [53]. □

Theorem 1. *Our authenticated graph query algorithms based on PAGB are secure if the hash function is collision resistant and the underlying RSA accumulator is secure.*

Proof. Based on Lemma 1, we prove it by contradiction.

$\{o|o \notin Q(D) \wedge o \in R\} \neq \phi$ states that there exists some objects in the result R that do not conform to the query Q on the original dataset D. During the process of prime representation, each object in D will first produce a hash value using the concatenation of reserved notations and then find a corresponding prime representative through random oracle. Due to the security property of the RSA accumulator, the membership witness of such an incorrect result cannot be forged, which means that there exist two objects with different contexts but identical hash value. This indicates the collision occurs in the underlying hash function, which violates our assumption.

$\{o|o \in Q(D) \wedge o \notin R\} \neq \phi$ means that some qualified objects are missing from the result R. Our authentication query algorithms require the CSP to give the proof

of non-existence. For the node property and edge property query, the CSP needs to provide the corresponding non-membership witness if the queried object does not exist. As for the connectivity query, during the depth-first search along the path, the CSP is required to return the non-membership witness if the node does not have such an edge. Moreover, the membership witness of the list of successor nodes is also needed. Since the unforgeability of the witness is guaranteed by the security of the RSA accumulator, a missing answer can only be caused by a hash collision between the valid result and the tampered result. Then we get a contradiction to the given assumption. □

4.6.2 Privacy Analysis

Besides the leakage revealed during the query, i.e., $\mathcal{L}_{query} = (\mathcal{L}_{n1}, \mathcal{L}_{n2}, \mathcal{L}_{n3}, \mathcal{L}_{e1}, \mathcal{L}_{e2}, \mathcal{L}_{e3}, \mathcal{L}_{e4}, \mathcal{L}_{c1}, \mathcal{L}_{c2})$ in Section 4.3, we also define the leakage from the BC side as $\mathcal{L}_{bc} = (|x|_q)$, where $|x|$ is the bit length of the prime representative and q is the amount of all elements. Based on $\mathcal{L}_{query}$ and $\mathcal{L}_{bc}$, we present the formal definition of privacy preserving in our design as below:

Definition 2 (Privacy preserving)**.** *The authenticated graph query algorithms on ADS achieve the property of privacy preserving if for all PPT adversaries $\mathcal{A}$, the probability to succeed in the following experiment is negligible.*

- *$\mathcal{A}$ chooses a graph dataset D. The algorithms generate the ADS and the corresponding state $\boldsymbol{Ac}$ via D and give them to $\mathcal{A}$;*
- *In response to a query Q, $\mathcal{A}$ returns a result R and a proof VO to the query user. The privacy of D is successfully leaked to $\mathcal{A}$ if the following condition is satisfied: $\{l|l \in \{R, VO\} \wedge l \notin \{\mathcal{L}_{query}, \mathcal{L}_{bc}\}\} \neq \phi$.*

This property states that it is infeasible for the query user to get any extraneous information from R and VO except the desired leakages, i.e., $\mathcal{L}_{query}$ and $\mathcal{L}_{bc}$. We then show that our proposed PAGB ensures the privacy preserving property.

Theorem 2. *Our authenticated graph query algorithms based on PAGB are privacy-preserving if the underlying hash function is preimage resistant.*

Proof. From the authenticated query side, our query processing algorithms will not leak any information other than $\mathcal{L}_{query}$. As shown in Section 4.3, we choose to return the non-existence proof instead of offering the entire data of the desired object, which prevents any extraneous information irrelevant to the query being accessed. For node property and edge property query, we make the CSP return the answer of the exact granularity that query Q has. It will return the answer if D has, otherwise, it will report the non-existence instead of presenting the entire related data. When processing the connectivity query, the CSP only gives the list of all successor nodes, i.e., $(\mathcal{L}_{c1}, \mathcal{L}_{c2})$, which are the basic knowledge of the connectivity information. Therefore, $\mathcal{A}$ cannot learn any irrelevant information about the data.

As for the blockchain side, the original graph data will not be revealed to the public through the blockchain. We can prove this case by contradiction. The metadata sent from the DO to the smart contract is a list of prime numbers, which are generated based on the hash values of objects in the graph data. Hence, the only leakage is the bit length and total amount of these numbers, i.e., $\mathcal{L}_{bc}$. Any leakage of the original data from the blockchain indicates that $\mathcal{A}$ can infer the original objects only from the hash values, which violates the assumption of preimage resistance.

□

Despite the leakage profiles of our query type having been explicitly defined, our query processing algorithms are vulnerable to some leakage-abuse attacks. Under these attacks, adversaries with some prior knowledge will be able to infer some useful information of original data based on the given leakages. For instance, suppose an adversary has the knowledge that there is only one girl in the dataset, then he can easily tell a node with a 'female' gender property is actually Alice. Hence, her name property can be seen as extra leakage since it is out of the defined leakage profile.

4.6.3 Complexity Analysis

We also investigate the complexity of our design. Assume n, e, a denote the node number, edge number and average attribute number respectively, the size of completeness set is $O(a * (n + e))$. The complexity of Prime Representation is $O(\log^4 p)$, where p is the prime number. It will find the primes for $O(\log p)$ times in average and

each time Miller–Rabin primality test is invoked, whose complexity is $O(\log^3 p)$. The PAGB maintenance, including insertion, deletion and update, takes time of $O(\log s)$, where s is the bits of prime. The complexity of property queries is $O(1)$ and the path query is of $O(q)$ size, where q is the number of qualified nodes. The single membership and non-membership witness `Prove` and `Verify` both take $O(s)$ time while batch `Prove` and `Verify` both take $O(b * s)$ time, where b is the batch size. The gas consumption in our PAGB, which will be presented in Section 4.7, is relatively lower than other designs like [41, 93]. The first reason is that we store only two variables on the smart contract instead of all original data since storing a new value is quite expensive on the blockchain. The other reason is that the modular exponentiation in our PAGB can be finished in time of $O(\log s)$, where s is the bit number of the exponent.

4.7 Implementations and Evaluation

In this section, to better demonstrate the effectiveness and efficiency of our design, we conduct the evaluation for the DO, CSP and the query users, including the test programs[1]. In addition, we construct a smart contract in Solidity and deploy it to the official Ethereum test network *Rinkeby*. The contract address and its owner account in Rinkeby are

- 0x8279c1f690af25d0b5777856eb6253f7e2333913.
- 0x71de049070119ab79b6189be6cb4acb099a291ca.

We run the experiments on a machine with i9-9900K CPU, 32 GB memory and 1 TB SSD. All tests are written in Python 3.8.0 and conducted only in one CPU core

[1]Online at https://github.com/tripleday/PAGB.

rather than in parallel to assess the worst case. Our design is able to be accelerated rapidly by parallel computation if applied to practical systems. In our experiments, we select three types of real-world graph datasets, i.e., ca-CondMat[2], cage15[3] and ConceptNet5[4], to comprehensively evaluate the performance of our design. The edge attributes of ConceptNet5 are reduced to 2 for simplicity. The public key n in our RSA accumulator generation is set to be 256-bit and the prime representatives are limited to be 128-bit integers, which are not absolutely secure but suffice for normal applications.

Table 5.1 lists the comparison between our design and other relevant designs for authenticated queries. In order to ensure fairness, we skip the quantitative comparison with other designs since most related designs do not possess the first three features listed in the table. The only work that we can compare is the traditional MHT adopted in [93]. However, the cost of storing graph data in the format of MHT using smart contract is too prohibitive to present, of $O(n^2)$ storage on BC rather than our $O(1)$ storage (n is the node number). Besides, MHT is incapable of privacy-preserving authenticated queries since all keys are exposed on the smart contract.

4.7.1 Accumulator Performance

Accumulator Setup. We first evaluate the performance of the accumulator setup, which includes prime representation, product calculation and accumulation, in Figure 4.4. We contrast our binary tree multiplication method with the normal successive multiplication for product calculation. When the amount of elements reaches

[2]Available at https://snap.stanford.edu/data/ca-CondMat.html.
[3]Available at https://sparse.tamu.edu/vanHeukelum/cage15.
[4]Available at http://conceptnet5.media.mit.edu/.

Table 4.2: Comparison with Existing Designs.

Design	Authentication Method(s)	General Graph	Property Graph	Dynamics	Privacy Preserving	ADS Freshness
[22]	Hashing with Common-prefix Proofs	×	×	√	√	×
[30]	Cryptographic Accumulator with Merkle Hash Tree	×	×	×	√	×
[48]	Tree Traversals and Aggregate Signatures	√	×	×	√	×
[12]	Hashing with Tree Representation	√	×	×	√	×
[33]	Redactable Graph Hashing	√	×	×	√	×
[96]	Cryptographic Accumulator with Digital Signature	√	×	√	√	×
[93]	Suppressed Merkle B-tree	×	×	√	×	√
Our PAGB	Dynamic Accumulator with Blockchain	√	√	√	√	√

1,000,000, our proposed method is over 50 times more efficient than the simple grade-school multiplication. The time cost of prime representation and accumulation are both nearly linear to the size of the element set. When the size is up to 1,000,000, the prime representation, multiplication and accumulation take about 1,100, 12,900 and 60 seconds respectively.

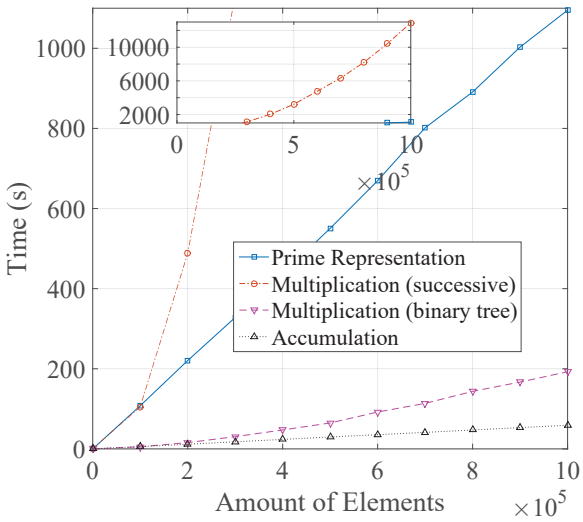


Fig. 4.4: Time Cost of Accumulator Setup.

Figure 4.5 shows the memory cost of our proposed binary tree multiplication. We

evaluate the method in the best case (the total number is a power of 2), the worst case (the total number is 1 plus power of 2) and the case of 1,000,000 elements. Since the multiplication proceeds from the bottom layer to the root, we count the total size of all elements involved in each step. We can observe that the size drops rapidly as the process goes to the root of the binary tree and finally remains at a level that is proportional to the total number. When the case switches from the best case to the worst one, the initial size at the leaf layer will increase a lot because our method will pad the set with a lot of '1' at first. However, it will diminish quickly to the normal size within 5 steps. This is because in the worst case, the number of padding '1' decreases and extra memory cost for them becomes negligible as a result.

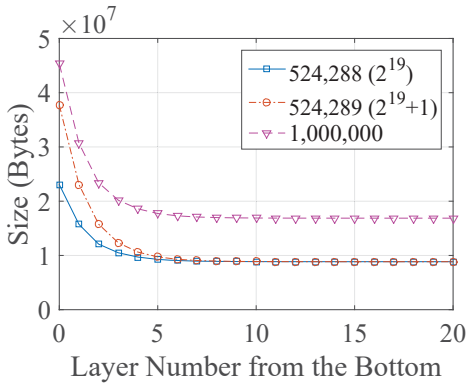


Fig. 4.5: Memory Cost of Binary Tree Multiplication.

Single Witness. Figure 4.6 depicts the generation time of one single membership witness and non-membership witness, which are widely adopted in the node property query and edge property query. We can see that the time costs of the two witnesses are close and both accrue almost proportional to the amount of elements. The reason behind both cases sharing similar proportional relationship is that the main time costs of membership and non-membership witness generation both depend on the modular

exponentiation, where the size of the power is determined by the number of existing elements. It takes about 1 minute to generate a membership or non-membership witness when the size reaches $1,000,000$.

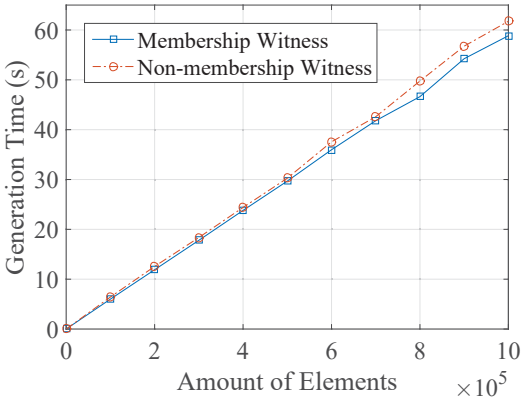


Fig. 4.6: Time Cost of Single Witness Generation.

Gas Consumption. We also test the gas consumption of the PAGB maintenance in the smart contract. Since both the add and deletion operations of the graph data turn out to be an element insertion into the accumulator, we present the transaction cost and execution cost with increasing element amount in Figure 4.7. The average gas cost for an extra element is only about 25,000 gas. Compared with traditional structures like tree-based ADS in [93], where all data and tree roots are stored on smart contract, our PAGB is much more efficient because only the accumulation value is updated.

Completeness Set. As for the accumulator setup based on various graph datasets, the main differences lie in the size of their completeness sets. We evaluate the completeness construction for different graph datasets in Figure 4.8. As shown in Figure 4.8a, the completeness set generated from ca-CondMat is small and only a bit bigger than the node set. This is because ca-CondMat is a simple directed graph with no

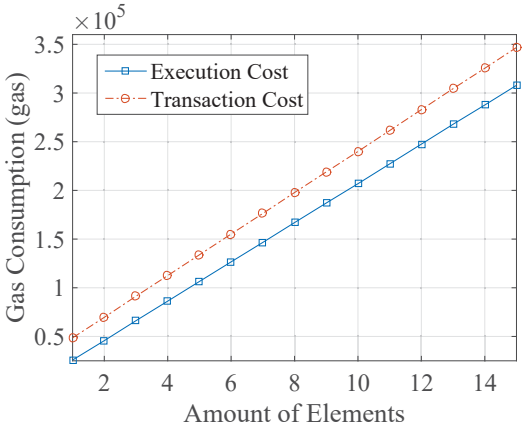


Fig. 4.7: Gas Consumption.

attributes or edge types, during the completeness construction, the algorithm only needs to generate elements about outbound and inbound nodes. As for the cage15 dataset that has one edge attribute, known as the weight, it also produces tuples for edge attributes. Figure 4.8b shows that the corresponding completeness set is larger than the node set and edge set. Figure 4.8c depicts the completeness construction of ConceptNet5, which is a directed graph with 24 edge types and 2 edge attributes. The size of completeness set is about 4.7 times the sum of the node and edge set because of many elements for edge types during the construction.

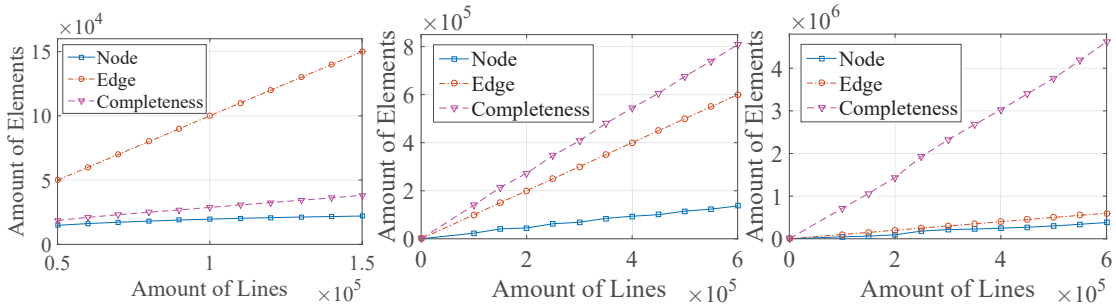
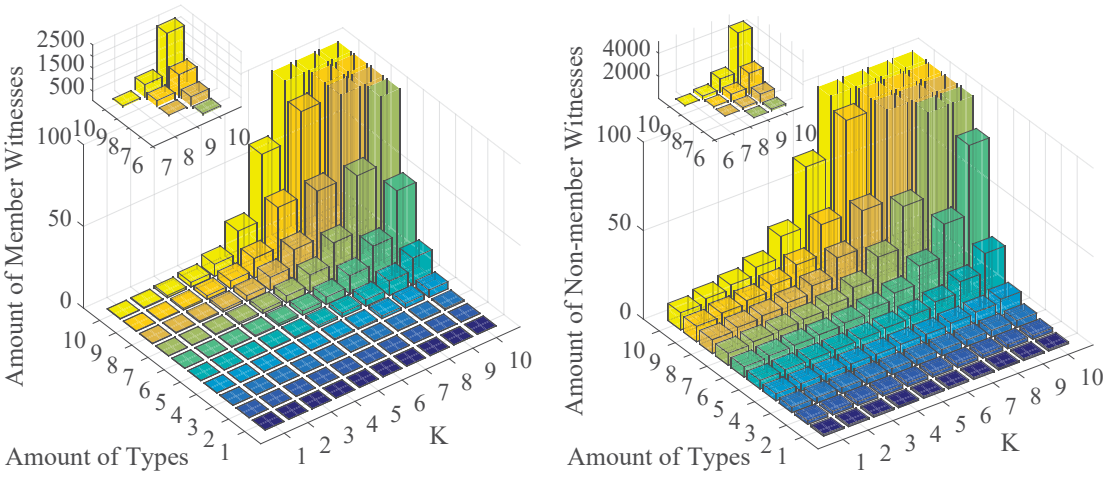


Fig. 4.8: Graph Data Completeness Construction.

4.7.2 Batch Verification

To illustrate the necessity of the batch verification, we present the amount of membership witnesses and non-membership witnesses that a connectivity query on ConceptNet5 involves in Figure 4.9a and Figure 4.9b respectively. We take the ConceptNet5 as an example and show the amount of membership witnesses and non-membership witnesses under an outbound query in Figure 4.9a and Figure 4.9b respectively. Both the step K and the size of type list $[type_k]$ vary from 1 to 10. When either of K and the amount of types are small, the average amount of membership witnesses needed remains small as well. However, the average amount of non-membership witnesses is relatively larger than that of membership witnesses. In addition, from Figure 4.9b we can see that, K affects the amount of non-membership witnesses more than the type amount. Observe that the amount of membership and non-membership witnesses will increase rapidly when K and amount of types both become larger.

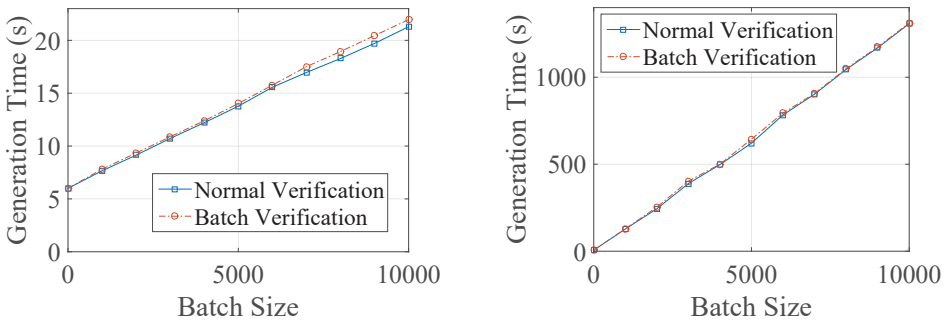


(a) Membership witness

(b) Non-membership witness

Fig. 4.9: Witnesses of A Connectivity Query on ConceptNet5.

We further evaluate the performance of the batch technique in terms of witness generation and witness verification. We conduct the experiment on a set that contains 100,000 elements with the batch size increasing from 1,000 to 10,000. As shown in Figure 4.10, compared with the normal verification which verifies the product directly without any protocols, the batch witness generation only takes negligible extra time. However, from Figure 4.11 we can observe that, the verification speed can be raised about 5 and 10 times for membership witness and non-membership witness respectively. Moreover, the batch proofs in these two cases also remain constant in size, 120 bytes for membership witness and 344 Bytes for non-membership witness. Therefore our design achieves both computational efficiency and communication efficiency.

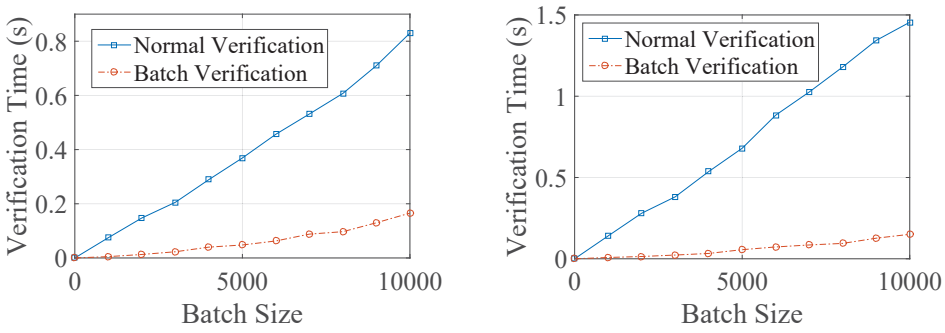


(a) Membership witness (b) Non-membership witness

Fig. 4.10: Batch Witness Generation.

4.8 Chapter Summary

In this chapter, we take the first step to investigate the authenticated graph query problem on the blockchain-assisted cloud. To address the challenges of authenticated query and privacy leakage, we put forth a novel ADS, named PAGB, that can be easily maintained by the blockchain at a low cost. The freshness and integrity of ADS can be perfectly ensured by the blockchain. We further improve the efficiency



(a) Membership witness

(b) Non-membership witness

Fig. 4.11: Batch Witness Verification.

by virtue of the non-interactive protocols and binary tree multiplication. Our design possesses the desirable features of dynamics and privacy preserving, and theoretical analysis and prototype implementation demonstrate its effectiveness.

Our work offers an unprecedented paradigm for graph data storage on blockchain along with its authenticated query. It opens up a new direction for storage of complicated data structures on blockchain. The main constraint of our design is that we assume the CSP will not leak any data since the outsourced data is raw rather than encrypted. Moreover, the query types on graph data we can support do not include some complicated ones that involve numerical values, e.g., shortest path query. In the future, besides the above constraints, it will also be interesting to investigate how to support other complex graph queries such as subgraph queries and graph similarity queries since our basic connectivity information may help.

Chapter 5

Verifiable, Secure and Fair Search on Encrypted Numerical Data

Verifiable Searchable Symmetric Encryption (SSE) enables reliable and privacy-preserving search over encrypted data on untrusted clouds. Most existing SSE designs only focus on keyword-file search. However, a more difficult but useful search, range search over encrypted numerical values remains unsolved. Moreover, the fairness of search in the mutual distrusted scenario without public verification, where data users may maliciously deny the results after the local result verification, is not well addressed yet. In this chapter, we take the first step to study the public verification problem atop the blockchain for encrypted numerical search. We design a novel verifiable SSE scheme named Slicer based on a Succinct Order-Revealing Encryption (SORE) scheme to achieve range search on numerical data. We illustrate the security and practicality of our design through rigorous analysis and extensive evaluations respectively.

5.1 Overview

Outsourcing data to clouds has become a strong trend for data owners to relieve great storage costs and heavy online burdens. Since the outsourced data may involve private information, e.g., medical records or business secrets, data owners usually encrypt the data while maintaining the ability to search over it. Some powerful and generic techniques like multi-party computation and homomorphic encryption cannot be applied to this scenario due to practical inefficiency despite high security. To this end, searchable symmetric encryption (SSE), which is a structured encryption based on symmetric encryption, has been extensively studied [29, 44] owing to its prominent efficiency.

Most traditional SSE schemes assume that clouds are honest but curious, which means they will honestly follow the stipulated protocols but attempt to learn information about the outsourced data. This assumption, however, does not always suffice in practical scenarios where dishonest clouds may deviate from the protocols and return non-conforming results. To alleviate this concern, verifiable SSE has become one of the focuses of active research (e.g., [20, 24, 59, 77, 95, 97]). Nevertheless, there still remain the following three limitations that have not been well addressed.

First, most verifiable SSE schemes [34, 59, 75, 77, 95, 97] are limited to the keyword-file search type, and incapable of *range search* over the data content. However, numerical data exists ubiquitously in the real world, such as ages in medical records, and transaction values in business secrets. How to enable the encrypted search over numerical data is challenging, since the solution of employing the traditional keyword-file search to traverse all values is totally infeasible.

In addition, many existing verifiable SSE designs [20, 24, 34, 59, 76, 97] let data

users locally verify the search results based on the assumption that data users will honestly report the verification outcome. But in practice, to avoid paying search fees, data users are strongly motivated to repudiate the search results despite of their correctness. Therefore, *public verification* of search results is highly desired to ensure fairness in this scenario, where data users and clouds are mutual distrusted. To achieve this, we should properly address two challenges: 1) The public verification cannot reveal any privacy of original data; 2) The process of public verification needs to be trusted. A recent work ServeDB [83] enables verifiable range queries over encrypted data, but its verification requires the decryption of data, which violates the first rule. Several designs [40, 41, 52] based on blockchain have been proposed to tackle the problem. Unfortunately, none of them support the range search over numerical data and excessive data is required to be stored on blockchain.

Lastly, data updates are also significant in real-world applications. This requirement entails the following two main challenges. First, the *data freshness* should be guaranteed in the multi-user scenario where data users may not be the data owner. Data users need to be convinced that the search results are from the newest data. Moreover, *forward security* [19], which prevents the insertion operation from leaking whether the newly added data matches former searches, is another important privacy requirement for data dynamics.

In this chapter, we are the first to investigate *the public verification problem atop the blockchain for encrypted numerical search*. To support numerical search, we devise a Succinct Order-Revealing Encryption (SORE) scheme that works like a slicer to slice an order condition into several slices, each of which can be treated as a keyword search. Further, we design a public verification algorithm for these slices using multiset hash

and RSA accumulator. We adopt the blockchain as the trusted party to fairly execute the public verification and guarantee the data freshness. We also incorporate the trapdoor permutation to achieve forward security so that insertion privacy can be guaranteed. In general, our contributions are summarized as follows:

- We take the first step to propose a framework of verifiable encrypted search over numerical data using blockchain. It supports public verification so that fairness can be ensured in the mutual distrusted scenario.
- We step over from the normal keyword search to the numerical search by devising the SORE scheme. Further, we design a novel verifiable and secure SSE scheme called Slicer, including Build, Search and Insert protocols.
- We strictly prove the correctness and security of the proposed SORE scheme and the encrypted search protocol.
- We implement a prototype and conduct extensive experiments to evaluate the performance. The result validates the effectiveness and efficiency of our design.

The rest of this chapter is organized as follows. We first give the preliminaries in Section 5.2. We then formulate the problem in Section 5.3 and describe our design in Section 5.4. We further analyze the design in Section 5.5 and present the evaluation in Section 5.6. Section 5.7 finally concludes our chapter.

5.2 Preliminaries

In this section, we briefly present some related preliminaries that will be used in our solution design.

Symmetric Encryption. A symmetric encryption scheme usually consists of three algorithms $\{KGen, Enc, Dec\}$: $KGen$ takes the security parameter λ as input and returns a symmetric key K_R; Enc takes the key K_R and a plaintext m as input and returns a ciphertext m'; Dec takes K_R and m' as input and returns the plaintext m.

Pseudo-Random Function. Define pseudo-random function (PRF) $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$, if for all PPT distinguishers D, there exists a negligible function $negl$ such that: $\left|\Pr\left[D^{F_k(\cdot)}\left(1^\lambda\right) = 1\right] - \Pr\left[D^{f_\lambda(\cdot)}\left(1^\lambda\right) = 1\right]\right| \leq negl(\lambda)$, where k is randomly chosen from $\mathcal{K}$ and $f_\lambda(\cdot)$ is a truly random function from $\mathcal{X}$ to $\mathcal{Y}$.

Trapdoor Permutation. A trapdoor permutation is a function that can be computed in one direction easily, but difficult in the inverse direction without the trapdoor. Formally, π is a trapdoor permutation if for any PPT adversary $\mathcal{A}$, $\Pr[y \xleftarrow{\$} \mathcal{M}, x \leftarrow \mathcal{A}\left(1^\lambda, pk, y\right) : \pi_{pk}(x) = y] \leq negl(\lambda)$ while $\pi_{pk}\left(\pi_{sk}^{-1}(x)\right) = x$ and $\pi_{sk}^{-1}\left(\pi_{pk}(x)\right) = x$. Here pk and sk are generated public key and secret key respectively, and $\pi_{pk}(\cdot)$ and $\pi_{sk}^{-1}(\cdot)$ can be efficiently calculated.

Multiset Hash Function. The multiset hash function maps a multiset to a fixed-size string. Define a triple of PPT algorithms $(\mathcal{H}, \equiv_{\mathcal{H}}, +_{\mathcal{H}})$ and it is a multiset hash function if for multiset M and N:

- $\mathcal{H}(M) \equiv_{\mathcal{H}} \mathcal{H}(M)$.
- $\mathcal{H}(M \cup N) \equiv_{\mathcal{H}} \mathcal{H}(M) +_{\mathcal{H}} \mathcal{H}(N)$.

In this chapter, we employ the MSet-Mu-Hash construction in [28]. It is defined as $\mathcal{H}(M) = \prod_{b \in B} H(b)^{M_b}$, where M is a multiset of elements of a countable set B and M_b is the number of times that b appears in M. $H(\cdot)$ is a poly-random function that maps a set to a finite field $GF(q)$. It is proved that $\mathcal{H}$ is multiset collision resistant under the discrete log assumption.

This chapter also employs four functions of RSA accumulator mentioned in Section 4.2, i.e., $\texttt{Setup}(1^\lambda)$, $\texttt{Accumulation}(X)$, $\texttt{MemWit}(x)$ and $\texttt{VerifyMem}(x, mw)$.

5.3 Problem Formulation

In this section, we present our model of verifiable encrypted search over numerical data and design goals.

5.3.1 Framework Architecture

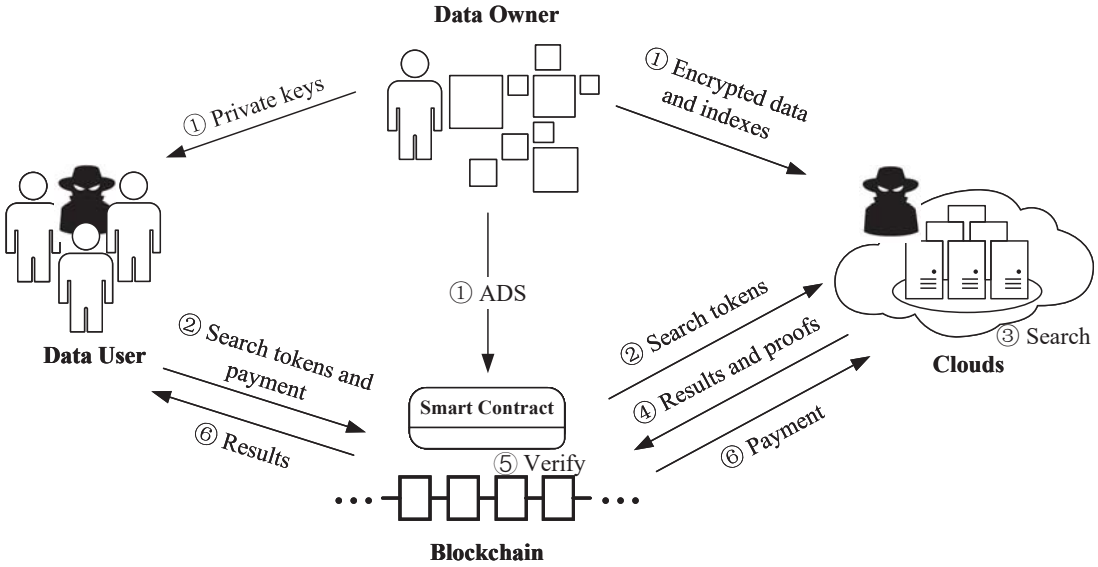


Fig. 5.1: Verifiable encrypted search using blockchain.

As shown in Figure 5.1, our search framework is comprised of four parties, i.e., data owners, data users, clouds and blockchain. The data owner outsources his encrypted data and established indexes to clouds for their storage and search services. He also gives his secret keys to authorized data users so that they can generate search tokens on their own. The blockchain can publicly verify the results returned by clouds via smart contract.

The workflow starts from the initialization of the data owner. In addition to the data, indexes and secret keys, the data owner also generates the authenticated data structure (ADS) and sends it to the blockchain. When the data user wants to search over the encrypted data, he calculates the search tokens and gives it to the blockchain as well as the payment for the cloud's search services. Then the cloud retrieves the search tokens and executes the search to get the results. It also returns the proofs for further result verification on blockchain. The verification is performed by the smart contract using the received search tokens, results and proofs. If the verification passes, the payment will be transferred to the cloud, otherwise it will be refunded.

5.3.2 Threat Model

We regard the data owner and blockchain as fully trusted parties. The data owner faithfully builds the encrypted indexes and ADS, and delivers them with encrypted data. The blockchain guarantees the trusted storage and program execution via underlying consensus protocols. For data users, we model them as quasi-honest, which means they are honest about secret keys maintenance and search token generation. But they may become dishonest about the result verification after receiving search results and proofs from clouds. They can save the search fees if they deliberately deny the returned results regardless of their correctness. For the clouds, we assume two aspects of dishonest behaviors. First, the cloud may maliciously return incorrect or incomplete results due to commercial interests or security vulnerabilities. The other aspect is that they might attempt to learn the content of the outsourced data for further abuse.

5.3.3 Design Goals

Our design aims to cover the following goals: (1) We cross over from the encrypted keyword-file search to the encrypted search on dynamic numerical data. (2) The data user can verify the data freshness without the online participation of the data owner. (3) The updated data should not reveal any information about previously searched tokens, i.e., forward security. (4) The result verification process needs to be publicly performed so that the fairness between data users and clouds can be ensured.

5.4 Slicer Design

5.4.1 Technical Overview

We denote the numerical database as a list of key-value pair records, i.e., $\mathsf{DB} = \{(R, v)\}$, where R is the unique record ID and v is the corresponding numerical value. The range search over numerical data is usually comprised of two types, i.e., equality search and order search. The former one means searching for records that have a certain value while the latter represents the search for records whose value is smaller or greater than the given value. Formally, a query consists of a value v and a matching condition $mc = \{\text{“=”}, \text{“>”}, \text{“<”}\}$.

The equality search can be seen as a variant of traditional keyword-file search, where the value becomes the file and the record ID is the keyword. Therefore, some previous schemes like multiset hash functions [28] can be adopted to facilitate the result verification by computing a set hash for each value. Nevertheless, these schemes cannot be directly applied to the order search due to the excessive amount of values. Our intuition for solving the order search verification is to slice the entire value field under the order condition into a fixed number of slices like a slicer. Each slice can be

seen as a unique tuple and the total number is only the bit count of the value. The original value satisfies the order condition if and only if it contains the same slice. We manufacture this slicer via the design of Succinct Order-Revealing Encryption (SORE) scheme.

5.4.2 SORE Scheme

Design Rationale. First, we need the smallest ciphertext space to compose a lightweight ORE scheme, because the verification cost is high on the blockchain. To this end, we generate only one ciphertext for each bit rather than the divided block. In order to avoid the one-by-one comparison on ciphertexts, we tokenize the orders and convert the computation to the tuple matching. Specifically, we embed the order relations into the encryption so that the order can be regarded as a one-bit value. Based on the left/right framework [51], we devise the tuple to make sure there exists one and only one common tuple matched if the ciphertext satisfies the query condition. Therefore, the order comparison between two values is transformed to the exact match among tuples, which can be further exploited to build encrypted indexes.

SORE Construction. We present our construction based on positive integers for simplicity since all numerical values in practical can be transformed into this form through scaling. Given a b-bit integer v, let v_i represent the ith bit of the value, and $v_{|i-1}$ denote the bits from 1 to $i-1$, i.e., the entire prefix of v_i. We use $\bar{v_i}$ to denote the inverse value derived by the bitwise `NOT` operation of v_i. Let $F : \{0,1\}^\lambda \times \{0,1\}^{b+1} \rightarrow \{0,1\}^\lambda$ be a secure PRF. During the setup phase, given a security parameter λ, our scheme outputs a uniformly random PRF key k as the secret key. Let $\|$ denote the concatenation operation. We define the core part of SORE scheme $\Pi = \{\mathsf{SORE.Token}, \mathsf{SORE.Encrypt}, \mathsf{SORE.Compare}\}$ as follows:

- SORE.Token(k, v, oc): Given the queried value v and the order condition $oc \in \{\text{“>”}, \text{“<”}\}$, the algorithm generates query tokens to find all answers a satisfying v oc a. For each $i \in [1, b]$, it computes a tuple $tk_i \leftarrow v_{|i-1} \| v_i \| oc$. Then it shuffles all tuples and outputs corresponding PRF values as tokens $tk = \{F_k(tk_1), F_k(tk_2), \cdots, F_k(tk_b)\}$.

- SORE.Encrypt(k, v): For each bit $i \in [1, b]$, it computes a tuple $ct_i \leftarrow v_{|i-1} \| \bar{v}_i \| cmp(\bar{v}_i, v_i)$, where $cmp(\bar{v}_i, v_i) \in \{\text{“>”}, \text{“<”}\}$ denotes the comparison result between $\bar{v}_i$ and v_i. Then the algorithm randomly shuffles all tuples and outputs their PRF values as ciphertexts $ct = \{F_k(ct_1), F_k(ct_2), \cdots, F_k(ct_b)\}$.

- SORE.Compare(ct, tk): Given the ciphertexts $ct = \{F_k(ct_1), F_k(ct_2), \cdots, F_k(ct_b)\}$ and the query tokens $tk = \{F_k(tk_1), F_k(tk_2), \cdots, F_k(tk_b)\}$, the algorithm checks whether they have one and only one value in common. If the common value exists, output `True`. Otherwise, output `False`.

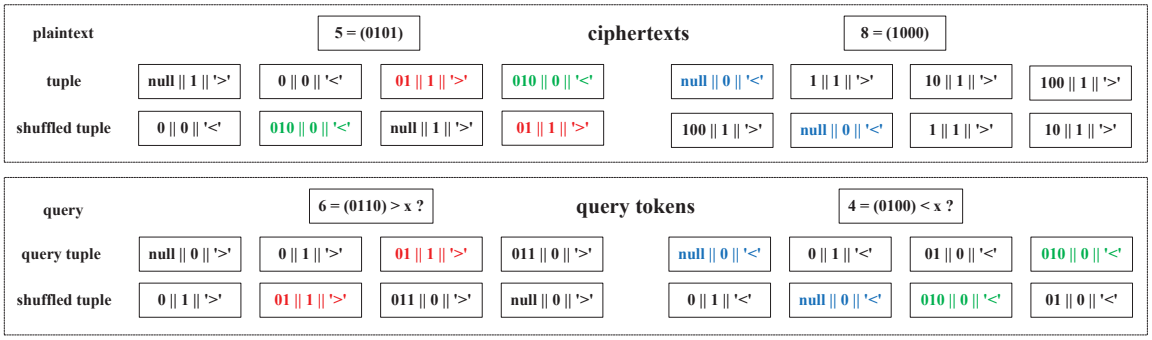


Fig. 5.2: An illustrative example of SORE.

We give an illustrative example of our SORE scheme in Figure 5.2. Suppose we have two plaintexts, i.e., 5 = (0101) and 8 = (1000), to be encrypted, and two query conditions, i.e., 6 = (0110) and 4 = (0100), to be executed. Among the

tuples generated by the corresponding algorithms, we mark the matched tuples for the two comparison results. Since the tuples are shuffled, the matched bit index can be concealed during each single query.

5.4.3 Building Encrypted Indexes and ADS

Algorithm 12: Build: Indexes and ADS Building

Input : PRF key K; encryption key K_R; key-value database DB; secure PRFs $\{F, G\}$; multiset hash function $\mathcal{H}$; random oracle function H_{prime}.
Output: Encrypted index I; ADS information (X, Ac).

foreach $(R, v) \in \mathsf{DB}$ **do**
  for $i = 1$ ***to*** b **do**
    $ct_i \leftarrow v_{|i-1} \| \bar{v}_i \| cmp(\bar{v}_i, v_i)$;
    Put (R, ct_i) into DB;
Initialize a dictionary I for indexes, T for trapdoor states and S for set hashes;
foreach $w \in \{v\} \cup \{ct_i\}$ **do**
  Randomly generate a trapdoor t_0;
  $T.put(w, (t_0, 0))$;
  $G_1 \leftarrow G(K, w\|1)$; $G_2 \leftarrow G(K, w\|2)$; $c \leftarrow 0$;
  $h \leftarrow \mathcal{H}(\phi)$;
  foreach $R \in \mathsf{DB}(w)$ **do**
    $l \leftarrow F(G_1, t_0\|c)$;
    $d \leftarrow F(G_2, t_0\|c) \oplus Enc(K_R, R)$;
    $I.put(l, d)$; $c++$;
    $h \leftarrow h +_{\mathcal{H}} \mathcal{H}(Enc(K_R, R))$;
  $S.put(t_0\|0\|G_1\|G_2, h)$;
Initialize a list X for primes;
foreach $(g, h) \in S$ **do**
  $x \leftarrow H_{prime}(g\|h)$; $X.add(x)$;
$\mathsf{Ac} \leftarrow \mathsf{Accumulation}(X)$;
Send (I, X, Ac) to the cloud;
Send Ac to the blockchain;
Send (K, K_R, T) to the data user;

Algorithm 12 presents the building process of the encrypted indexes and ADS. Following SORE.Encrypt, we generate the tuples (line 1 to 4) and produce the encrypted indexes (line 5 to 16) and ADS, i.e., RSA accumulator (line 17 to 20), together with the values. To enable the forward security, we employ the trapdoor permutation [19] to make the updated values unlinkable to previous searches until the newest search token is issued. Specifically, we first generate a random trapdoor and two tokens G_1 and G_2 using the PRF G. G_1 and G_2 can hide the true values and be further used to build the indexes via the PRF F. Note that we use the concatenation of the trapdoor and a self-incremental counter c to index the encrypted R via l and d. As for the RSA accumulator, we first get a random hash through the multiset hash function $\mathcal{H}$ on the qualified result set for each w, i.e., the original value v or tuple ct_i. Then we calculate a prime representative for the concatenation of the search token and corresponding set hash, and get the accumulation value Ac by accumulating all primes. Ac is sent to both the cloud and the blockchain while the prime list X will be uploaded only to the cloud for the generation of proof, known as verification object (VO). Moreover, the data user keeps the trapdoor states for further search requests.

5.4.4 Data Insertion

As shown in Algorithm 13, the forward-secure insertion protocol follows the similar procedure as the Build protocol. The main trick that achieves the forward security during the insertion lies in the trapdoor update when w has been searched before (line 12 to 16). Specifically, the data owner needs to use the trapdoor permutation π to get a new trapdoor based on the former one via $\pi_{sk}^{-1}(t)$, where sk is the secret key of π. The new trapdoor is saved to the state dictionary together with the update times j.

Algorithm 13: Insert: Forward-Secure Insertion

Input : PRF key K; encryption key K_R; trapdoor secret key sk; key-value pairs to insert DB^+; secure PRFs $\{F, G\}$; multiset hash function $\mathcal{H}$; random oracle function H_{prime}.

Output: Updated encrypted index I; updated ADS information (X, Ac).

foreach $(R, v) \in \mathsf{DB}^+$ **do**
  for $i = 1$ ***to*** b **do**
    $ct_i \leftarrow v_{|i-1}\|\bar{v}_i\|cmp\,(\bar{v}_i, v_i)$;
    Put (R, ct_i) into DB^+;
Initialize a list X^+ for primes to add;
foreach $w \in \{v\} \cup \{ct_i\}$ **do**
  $G_1 \leftarrow G\,(K, w\|1)$; $G_2 \leftarrow G\,(K, w\|2)$; $c \leftarrow 0$;
  if $T.find(w) = \perp$ **then**
    $h \leftarrow \mathcal{H}(\phi)$;
    Randomly generate a trapdoor t; $j \leftarrow 0$;
    $T.put(w, (t, j))$;
  else
    $t, j \leftarrow T.get(w)$;
    $h \leftarrow S.pop(t\|j\|G_1\|G_2)$;
    $t \leftarrow \pi_{sk}^{-1}(t)$; $j++$;
    $T.put(w, (t, j))$;
  foreach $R \in \mathsf{DB'}(w)$ **do**
    $l \leftarrow F(G_1, t\|c)$;
    $d \leftarrow F(G_2, t\|c) \oplus Enc(K_R, R)$;
    $I.put(l, d)$; $c++$;
    $h \leftarrow h +_{\mathcal{H}} \mathcal{H}(Enc(k_R, R))$;
  $S.put(t\|j\|G_1\|G_2, h)$;
  $x^+ \leftarrow H_{prime}(t\|j\|G_1\|G_2\|h)$; $X^+.add(x^+)$;
$X \leftarrow X \cup X^+$;
$\mathsf{Ac} \leftarrow \mathsf{Accumulation}(X)$;
Send (I, X, Ac) to the cloud;
Send Ac to the blockchain;
Send T to the data user;

5.4.5 Verifiable Search Protocol

Algorithm 14: Search: Search Token Generation

Input : PRF key K; encryption key K_R; trapdoor public key pk; query value v; matching condition mc; secure PRFs $\{F, G\}$.
Output: Search tokens sts.

User.Token
- **if** $mc \in \{$ “>”, “<” $\}$ **then**
  - **for** $i = 1$ ***to*** b **do**
    - $tk_i \leftarrow v_{|i-1} \| v_i \| mc$;
  - Randomly shuffle $\{tk_i\}$;
  - $W \leftarrow \{tk_i\}$;
- **else**
  - $W \leftarrow \{v\}$;
- Initialize a list sts for search tokens;
- **foreach** $w \in W$ **do**
  - **if** $T.find(w) \neq \perp$ **then**
    - $t_j, j \leftarrow T.get(w)$;
    - $G_1 \leftarrow G(K, w\|1)$; $G_2 \leftarrow G(K, w\|2)$;
    - $sts.add((t_j, j, G_1, G_2))$;
- Send sts and payment to the blockchain;

Algorithm 14 describes the search token generation executed by the data owner in the Search protocol. Following SORE.Token, the data owner first produces the token list $\{tk_i\}$. Along with v, he then generates the corresponding search tokens for each item, including the trapdoor, the update times, G_1 and G_2, if it exists in the trapdoor states. Finally, he sends the search tokens and the payment to the blockchain.

After retrieving the search tokens from the blockchain, the cloud starts the search as shown in Algorithm 15. The cloud will traverse from the newest indexes by using PRF on the concatenation of the newest trapdoor t_j and the counter c. After each traversal, it computes the previous trapdoor using $\pi_{pk}(t_i)$, where pk is the public key

Algorithm 15: Search: Cloud Search

Input : Search tokens sts; trapdoor public key pk; secure PRF $\{F\}$; multiset hash function $\mathcal{H}$, random oracle function H_{prime}.
Output: Encrypted matched results er; verification objects $\{vo\}$.

Cloud.Search
  foreach $(t_j, j, G_1, G_2) \in sts$ **do**
    for $i = j$ ***to*** 0 **do**
      for $c = 0$ ***until*** $I.find(l) = \perp$ **do**
        $l \leftarrow F(G_1, t_i \| c)$;
        $r \leftarrow F(G_2, t_i \| c) \oplus I.get(l)$;
        $er.add(r)$; $c++$;
      $t_{i-1} \leftarrow \pi_{pk}(t_i)$;
    $h \leftarrow \mathcal{H}(er)$; $x \leftarrow (H_{prime}(t_j \| j \| G_1 \| G_2 \| h)$;
    $vo \leftarrow$ `MemWit` (x);
    Send er and vo to the blockchain;

of the trapdoor permutation, and proceeds the next round. When all traversals end, the algorithm calculates the set hash on the result list and derives the prime number accordingly. The membership witness of the prime will be generated from `MemWit` and then sent to the blockchain with the results.

We present the result verification by the blockchain in Algorithm 16. It only needs to reproduce the prime number based on the search tokens and corresponding results. Then `VerifyMem` of the RSA accumulator will be invoked to validate the correctness of the VOs.

5.4.6 Extensions

Data Deletion and Update. Although the data deletion cannot be directly supported by our scheme, but it can be addressed by duplicating the original construction [19]. In another word, we can use one instance for all inserted data while the other

Algorithm 16: Search: Result Verification

Input : Search tokens sts; encrypted matched results er; verification objects $\{vo\}$; multiset hash function $\mathcal{H}$, random oracle function H_{prime}; encryption key K_R.
Output: Verification result vr.

```
1 Blockchain.Verify
2     vr ← True;
3     foreach (t_j, j, G_1, G_2, er, vo) do
4         h ← H(er); x ← (H_prime(t_j‖j‖G_1‖G_2‖h);
5         vr ← VerifyMem (x, vo);
6         if vr=False then
7             Refund the payment;
8     Proceed with the payment;
9     The data user decrypts all er using Dec(K_R, er);
```

one stores all deleted data. In this way, the final search result becomes the difference between the corresponding results from the two instances. As for the update on one record, it can be regarded as a combination of one deletion operation and one insertion operation. Note that we do not allow a repetitive insertion of the same record ID in both instances since the ID is unique.

Data with Multiple Attributes. Our design can be easily extended to data with multiple attributes a, i.e., $\mathsf{DB} = \{(R, \{(a, v)\})\}$, which is a more popular and practical data type. Specifically, we can incorporate the attribute name a into the token and the ciphertext, i.e., $tk_i \leftarrow a\|v_{|i-1}\|v_i\|oc$ and $ct_i \leftarrow a\|v_{|i-1}\|\bar{v}_i\|cmp(\bar{v}_i, v_i)$, for each value.

5.5 Design Analysis

We perform a formal analysis on the correctness and security of our SORE scheme and encrypted search protocol.

5.5.1 Correctness and Security on SORE scheme

Our SORE scheme is inspired by the ORE schemes in [27, 51, 90]. We devise a lightweight scheme to enable the efficient encrypted search and public verification while remaining comparable security. In this subsection, we present the correctness analysis of SORE and discuss its leakage.

Correctness Analysis. We prove the correctness by giving the proof of the following theorem:

Theorem 3. *Given the PRF key k, two values x, y, and the order condition* $\mathsf{oc} \in \{\text{“}>\text{”}, \text{“}<\text{”}\}$, *write* $tk \leftarrow \{F_k(tk_1), F_k(tk_2), \cdots, F_k(tk_b)\}$ *generated by* $\mathsf{SORE.Token}(k, x)$ *and* $ct \leftarrow \{F_k(ct_1), F_k(ct_2), \cdots, F_k(ct_b)\}$ *generated by* $\mathsf{SORE.Encrypt}(k, y)$. x oc y *stands if and only if* $\mathsf{SORE.Compare}(ct, tk) = \texttt{True}$.

Proof. Because secure PRF is applied to both sides, $\mathsf{SORE.Compare}(ct, tk)$ can be reduced to the comparison between $\{x_{|i-1}\|x_i\|\mathsf{oc}\}$ and $\{y_{|i-1}\|\bar{y_i}\|cmp(\bar{y_i}, y_i)\}$ before shuffle. We first argue that if $\{x_{|i-1}\|x_i\|\mathsf{oc}\}$ and $\{y_{|i-1}\|\bar{y_i}\|cmp(\bar{y_i}, y_i)\}$ have tuples in common, the amount of the tuples must be 1. We will give the proof by contradiction. Since the length of the tuple is determined by the bit index due to the prefix, the same tuple must share the same index. Suppose we already have an identical tuple at index m, i.e., $x_{|m-1}\|x_m\|\mathsf{oc}= y_{|m-1}\|\bar{y_m}\|cmp(\bar{y_m}, y_m)$. This means $x_{|m-1} = y_{|m-1}$ and $x_m = \bar{y_m}$. Then we assume there exists another common tuple at index n. If $n < m$, since $x_n = y_n$, then $x_n \neq \bar{y_n}$ must stand, which contradicts the assumption of the common tuple at n. If $n > m$, then $x_{|n-1} \neq y_{|n-1}$ because $x_m \neq y_m$. It also violates the previous assumption. Thus, the claim follows.

Next, we prove the correctness in two situations:

- Suppose x oc y stands. Let m be the smallest index where the bit value differs, i.e., $x_{|m-1} = y_{|m-1}$ and $x_m = \bar{y_m}$. Because m is the smallest differing bit index, the order between x and y coincides with that between x_m and y_m, which means $\mathsf{oc} = cmp(\bar{y_m}, y_m)$. Then $\mathsf{SORE.Compare}(ct, tk)$ outputs $\texttt{True}$ since the tuple at index m is the desired common one.

- Suppose that $\mathsf{SORE.Compare}(ct, tk) = \texttt{True}$, i.e., there exists one and only one common tuple, and let m be the bit index of the tuple. Now we have $x_{|m-1}\|x_m\|\mathsf{oc}= y_{|m-1}\|\bar{y_m}\|cmp(\bar{y_m}, y_m)$, which means $x_{|m-1} = y_{|m-1}$, $x_m = \bar{y_m}$, and $\mathsf{oc}= cmp(\bar{y_m}, y_m)$ all hold. Apparently, the order between x and y is determined by that between x_m and y_m since it is the first differing bit. Then x oc y follows.

□

Leakage Discussion. Solely adopting our SORE scheme leaks the index of the first differing bit among query tokens or among ciphertexts. Specifically, given a list of query tokens generated by SORE.Token, we can find out the leakage between any two values by counting how many common tuples exist. The leakage among the ciphertexts that is produced by SORE.Encrypt can be learned likewise. Nevertheless, the risk brought by the leakage among ciphertexts can be eliminated by Build and Insert protocol. It is because the indexes are derived through secure PRF and stored in a history-independent dictionary, which totally conceals the relationships among ciphertexts. As for each pairwise comparison between query tokens and ciphertexts, the SORE.Compare has no leakage owing to the semantic security of PRF and the shuffle operations. The formal security analysis of the encrypted search equipped with SORE is presented in the next subsection.

5.5.2 Security on Encrypted Search

In this subsection, following the security notion of SSE [23, 29, 44], we prove the security of our encrypted search protocol. Before presenting the security theorem, we first give the formal definitions of our four leakage functions. After the data owner initially builds the encrypted indexes and ADS, we have the following information leakage:

$$\mathcal{L}^{build}(\mathsf{DB}) = (\langle |l|, |d| \rangle_p, |x|_q),$$

where DB is the record-value pairs. $\langle |l|, |d| \rangle$ are bit lengths of the encrypted index I and p is the size of I. $|x|$ is the bit length of the prime number, q denotes the size of the prime list X. When the data user issues a search request to the cloud, the

leakage captured by the server is defined as:

$$\mathcal{L}^{search}(v, mc) = \left(\left\{t_j, j, G_1, G_2, \{\langle l, d, er\rangle_{c_i}\}_j, h, x, vo\right\}_n\right),$$

where v is the queried value and mc is the queried matching condition. This leakage is a n-size list of search tokens and corresponding results. t_j, j, G_1, G_2 form the search token and $\{\langle l, d, er\rangle_{c_i}\}_j$ are matched indexes and encrypted results in each loop. h is the multiset hash, x is the prime representative and vo is the verification object. The leakage function during the data insertion can be defined as:

$$\mathcal{L}^{insert}(\mathsf{DB}^+) = \left(\langle |l^+|, |d^+|\rangle_{p^+}, |x^+|_{q^+}\right),$$

where DB^+ is the inserted records. $\langle |l^+|, |d^+|\rangle_{p^+}$ are newly added indexes whose size is p^+. $|x^+|$ is the bit length of the added prime number and q^+ is the number of primes. Moreover, we have a leakage function to track repeated queries:

$$\mathcal{L}^{repeat}(Q) = \left(M_{r\times r}, \left\{r, \{\langle l, d, er\rangle_{c_i}\}_j, h, x\right\}\right),$$

where Q is r number of historical queried tokens and $M_{r\times r}$ is a symmetric bit matrix that records the repeat information. All elements in $M_{r\times r}$ are initially set to 0. If the i-th search token is identical to the j-th one, then $M_{i,j}$ and $M_{j,i}$ are equal to 1. Given the above leakages, we adopt the simulation proof technique and give the following security definition:

Definition 3. *Let* $\Omega = (\mathsf{KGen}, \mathsf{Build}, \mathsf{Search}, \mathsf{Insert})$ *be our encrypted search scheme, and let* $\mathcal{L}^{build}$, $\mathcal{L}^{search}$, $\mathcal{L}^{insert}$ *and* $\mathcal{L}^{repeat}$ *be the leakage functions. For a PPT adversary* $\mathcal{A}$ *and a PPT simulator* $\mathcal{S}$*, we define the games* $\mathbf{Real}_{\mathcal{A}}(\lambda)$ *and* $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ *as follows:*

$\mathbf{Real}_{\mathcal{A}}(\lambda)$*: The data owner generates a private key* K *using* $\mathsf{KGen}(1^\lambda)$*.* $\mathcal{A}$ *chooses a dataset* DB *and asks the data owner to build encrypted indexes and ADS via* Build*. Next,* $\mathcal{A}$ *repeatedly requests a polynomial number of verifiable queries or data insertions. To respond, the game runs* Search *or* Insert *with corresponding inputs. Eventually,* $\mathcal{A}$ *returns a bit that the game adopts as the output.*

Ideal$_{\mathcal{A},\mathcal{S}}(\lambda)$*:* $\mathcal{A}$ *selects a dataset* DB*, and* $\mathcal{S}$ *builds simulated indexes and ADS based on the given leakage* $\mathcal{L}^{build}$*. Next,* $\mathcal{A}$ *repeatedly requests a polynomial number of verifiable queries or data insertions. To respond to the queries,* $\mathcal{S}$ *generates the simulated search tokens and results based on* $\mathcal{L}^{search}$ *and* $\mathcal{L}^{repeat}$*. In response to insertion,* $\mathcal{S}$ *updates the indexes and ADS based on* $\mathcal{L}^{insert}$*. Finally,* $\mathcal{A}$ *returns a bit that the game adopts as the output.*

We say Ω *is adaptively secure with* ($\mathcal{L}^{build}$, $\mathcal{L}^{search}$, $\mathcal{L}^{insert}$, $\mathcal{L}^{repeat}$) *leakages if for all adversaries* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}$ *such that:* $\Pr[\mathbf{Real}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda) = 1] \leq \mathrm{negl}(\lambda)$*, where* $\mathrm{negl}(\lambda)$ *denotes a negligible function in* λ*.*

Theorem 4. Ω *is adaptively secure with* ($\mathcal{L}^{build}$, $\mathcal{L}^{search}$, $\mathcal{L}^{insert}$, $\mathcal{L}^{repeat}$) *in the random-oracle model if* F, G *are secure PRFs, and* (Enc, Dec) *is CPA-secure.*

Proof. We first define random oracles $\{\mathcal{O}_F, \mathcal{O}_G, \mathcal{O}_{mh}, \mathcal{O}_{prime}\}$ and then sketch the execution of the simulator $\mathcal{S}$. At the build-up phase, $\mathcal{S}$ generates the simulated indexes and ADS based on $\mathcal{L}^{build}$. Specifically, it includes p entries of $|l|$-bit and $|d|$-bit random string pairs, i.e., $\langle l', d' \rangle$, as the indexes, and q number of $|x|$-bit random prime numbers denoted as x'.

Given the first query (v, mc), if mc is "=", $\mathcal{S}$ simulates $G'_1 = \mathcal{O}_G(K' \| v \| 1)$ and $G'_2 = \mathcal{O}_G(K' \| v \| 2)$ as the tokens, where K' is a random string. Otherwise, $\mathcal{S}$ first generates b number of tuples $v_{|i-1} \| v_i \| mc$ and then randomly pick each one to produce $G'_1 = \mathcal{O}_G(K' \| v_{|i-1} \| v_i \| mc \| 1)$ and $G'_2 = \mathcal{O}_G(K' \| v_{|i-1} \| v_i \| mc \| 2)$. A random trapdoor t'_j, j, G'_1 and G'_2 form a simulated search token. For i from j to 1, the random oracle $\mathcal{O}_F$ is programmed so that $\mathcal{O}_F(G'_1 \| t_i \| c) = l'$ on c from 0 to c_i to find the matched indexes, where t_i randomly generated in each loop. For each matched entry, $\mathcal{S}$ operates $\mathcal{O}_F$ to satisfy $\mathcal{O}_F(G'_2 \| t_i \| c) \oplus l' = \mathcal{O}_R(K'_R \| \alpha) \oplus R$, where $\mathcal{O}_R$ is a random oracle, α is a random string and R is the record ID. Moreover, $\mathcal{O}_{mh}$ is programmed so that $\mathcal{O}_{mh}(\{er\}) = h'$, where h' is a random string. $\mathcal{O}_{prime}$ is programmed to meet $\mathcal{O}_{prime}(t'_j \| j \| G'_1 \| G'_2 \| h') = x'$. Then a simulated vo' is generated through the membership witness algorithm using x'. At last, for each matched entry r, $\mathcal{S}$ sets $M'_{r,r}$ to 1 and records corresponding information.

For the subsequent queries, $\mathcal{S}$ will generate the tokens and check whether each token appeared before through M'. If yes, $\mathcal{S}$ returns the repeated matching entry and generates vo' accordingly. Otherwise, $\mathcal{S}$ will simulate the query tokens and corresponding results in the same way as the first query process. Eventually, $\mathcal{S}$ updates M' and stores the repetition information.

To respond to each adaptive data insertion request, $\mathcal{S}$ simulates p^+ entries of random indexes and q^+ number of random primes, who have the same sizes as stated in $\mathcal{L}^{insert}$.

Due to the pseudo-randomness of PRFs and the semantic security of symmetric encryption, it is infeasible for $\mathcal{A}$ to distinguish between the real outputs and the

simulated ones. The definition of forward security in [19] requires the insertion should not reveal any information about the added keywords. Our $\mathcal{L}^{insert}$ only contains some random strings and numbers as well as their amounts, thus meeting forward security. □

Despite the security analysis above, our design may still be vulnerable to some leakage-abuse attacks, where our defined privacy leakage can be abused to infer plaintext information. For instance, by observing a series of relations between query tokens and result size, adversaries can learn sensitive information about the indexes and plaintexts. This is also known as volume attacks [39, 45, 49] based on the volume-pattern leakage, which will be revealed in our leakage profiles.

5.5.3 Correctness of Verifiable Search

We prove the correctness of verification in terms of soundness and completeness.

Definition 4. *We say a verifiable query algorithm is correct if for any PPT adversary $\mathcal{A}$, the following experiment has negligible possibility to succeed:*

- *$\mathcal{A}$ chooses a key-value dataset DB. The algorithm constructs the indexes and ADS based on DB, and gives the ADS state Ac to $\mathcal{A}$;*
- *To respond to a query Q, $\mathcal{A}$ outputs a result $\{rs\}_n$ and a proof $\{vo\}_n$ to the query user. $\mathcal{A}$ performs a successful attack if the proof passes the verification using Ac and R satisfies: $\{R|R \notin Q(\mathsf{DB}) \wedge R \in \{rs\}_n\} \neq \phi \vee \{R|R \in Q(\mathsf{DB}) \wedge R \notin \{rs\}_n\} \neq \phi$.*

Theorem 5. *Ω is correct if the multiset hash function and prime representation function are collision resistant, and the underlying RSA accumulator is secure.*

Proof. We give the proof by contradiction. The first case, i.e., $\{R|R \notin Q(\mathsf{DB}) \wedge R \in \{rs\}_n\} \neq \phi$, indicates that there exists a record R in the result that does not satisfy $Q(\mathsf{DB})$. The second one means that some records that conform to the query condition are not included in the result. Let rs' be the result containing the incorrect or incomplete records and the corresponding proof be vo'. Due to the security of the RSA accumulator [53], the membership witness of an element cannot be forged. It means that if vo' can pass the verification, the true rs shares the same multiset hash or the same prime representative with rs'. This violates the assumption of collision-resilience of the multiset hash function and prime representation function. □

5.6 Implementations and Evaluation

We first list state-of-the-art related studies on verifiable SSE in Table 5.1. To the best of our knowledge, our Slicer is the first system that supports all desired features including data dynamics, numerical comparison, data freshness, forward security and public verifiability.

Table 5.1: Comparison with State-of-the-Art Verifiable Searchable Encryption Schemes

Designs[a]		**Dynamics**[b]	**Numerical comparison**	**Freshness**[c]	**Forward**[d] **security**	**Public**[e] **verifiability**
Traditional designs	[24]	×	×	N/A	N/A	×
	[76] [20]	√	×	N/A	√	×
	[83]	√	√	×	×	×
	[34]	√	×	×	×	×
	[97]	√	×	√	×	×
	[59]	√	×	×	×	×
	[75]	×	×	N/A	N/A	√
	[95]	×	×	N/A	N/A	×
	[77]	√	×	×	×	√
Blockchain-based designs	[41] [40] [52]	√	×	√	√	√
	[21]	×	×	√	√	√
	Ours	√	√	√	√	√

[a] We exclude TEE-based solutions because they can achieve arbitrary functionalities through customized programs. But they cannot provide public verifiability due to the encapsulation of TEE.
[b] The dynamics covers operations including addition, update and deletion over the encrypted data.
[c] The data freshness can be verified by the data user without the online participation of the data owner. 'N/A' means the freshness property does not apply to the design because it is either a static-data scenario or a single-user scenario (the owner is the user).
[d] 'N/A' means the schemes without data addition inherently do not support forward security.
[e] The integrity of the search result needs to be publicly verified in case of malicious data users.

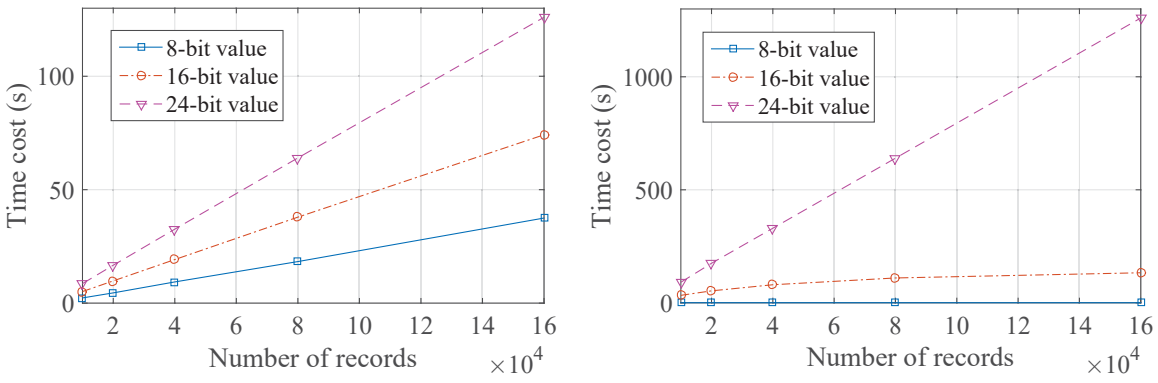
To demonstrate the practical efficiency of our design, we implement a prototype[1], including the data owner, data user and clouds in Python 3.8.0 and the blockchain in Solidity. We perform the evaluation on a machine with i9-9900K CPU, 32 GB memory

[1]Online at https://github.com/tripleday/Slicer.

and 1 TB SSD. For cryptographic primitives, we employ AES-128 for the symmetric encryption, HMAC-128 for the pseudo-random function, and RSA implementation for the trapdoor permutation. We evaluate the time cost and overhead size based on randomly simulated key-value records, where the value has 8, 16 and 24 bit settings.

5.6.1 Building Performance

Figure 5.3 presents the time cost of our Build protocol. We evaluate the time of index building and ADS building at three bit settings based on the records of 10K, 20K, 40K, 80K, 160K entries. As we can see from Figure 5.3a, the time cost of index building raises linearly in all cases as the amount increases. It only takes roughly 38s to build encrypted indexes 160K records of 8-bit values. As for the ADS building in Figure 5.3b, the time cost for 8-bit values is almost a constant value, i.e., around 0.5s for any amount of records. This is due to the limited value space under the 8-bit setting. Regarding the 16-bit and 24-bit settings, the ADS building time increases rapidly as the growing amount incurs larger value space.

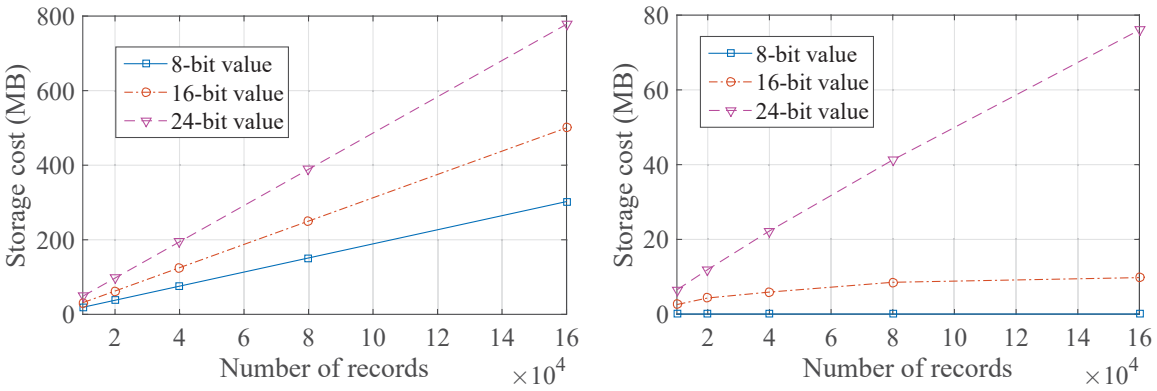


(a) Indexes (b) ADS

Fig. 5.3: Time cost of Build.

We show the storage cost of the indexes and the ADS during the building phase

in Figure 5.4a and Figure 5.4b respectively. The index storage is proportional to the amount of records since each record maps to a constant number of index entries. For the storage of ADS, i.e., the size of the prime list, upon 8-bit values, it keeps constant as 0.04MB due to the aforementioned value space. Under the other two settings, the storage grows linearly but still remains at a practical level.



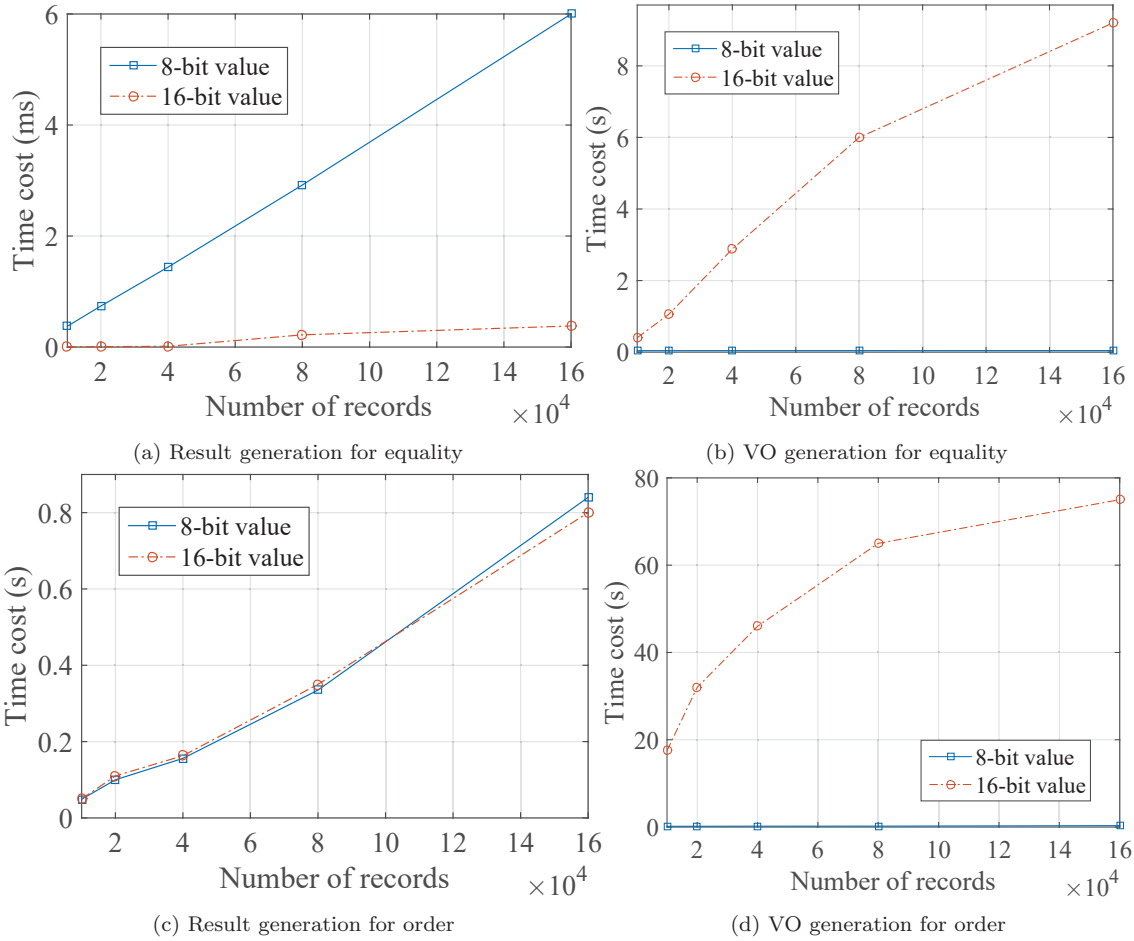
(a) Indexes (b) ADS on cloud

Fig. 5.4: Storage cost of Build.

5.6.2 Search Performance

To evaluate the performance of search, we select random numbers to execute the protocol and average the outcomes. Figure 5.5 depicts the time of cloud search, including the result generation and VO generation, for the equality search and order search. For the result generation time of equality search in Figure 5.5a, the time rises faster on the 8-bit values than the 16-bit values because the number of qualified results is larger. In contrast, the time costs for order search in Figure 5.5c under two settings both increase due to the similar number of results. Although the result generation for equality search costs more time than order search, its VO generation time in Figure 5.5b keeps steadily at about 0.04s. The VO generation for order search

costs around 0.32s when the amount of records reaches 160K. When the bit count extends from 8 to 16, the VO generation time grows significantly for both search types.



(a) Result generation for equality

(b) VO generation for equality

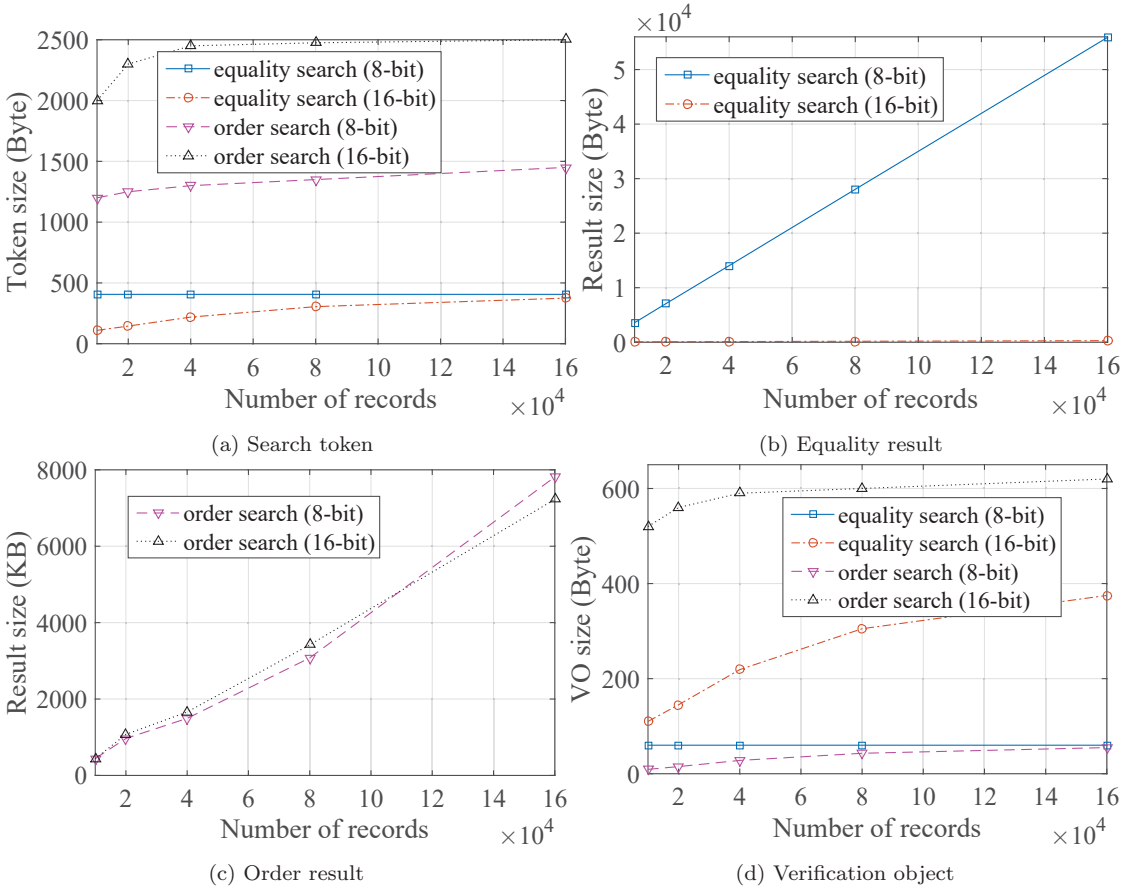
(c) Result generation for order

(d) VO generation for order

Fig. 5.5: Time cost of Search.

Figure 5.6 presents the overhead, i.e., search tokens, encrypted results and verification objects, produced by Search protocol. The amount of search tokens generated by the 8-bit setting stays relatively stable while the 16-bit setting produces more tokens since the records gradually fill the value space. As shown in Figure 5.6b and Figure 5.6c, the size of encrypted results under all settings is proportional to the

amount of records. As for the VO in Figure 5.6d, its size under the 8-bit setting is always smaller than 60 Bytes, whereas the size of 16-bit setting slowly increases and levels off due to the constant number of tuples.



(a) Search token

(b) Equality result

(c) Order result

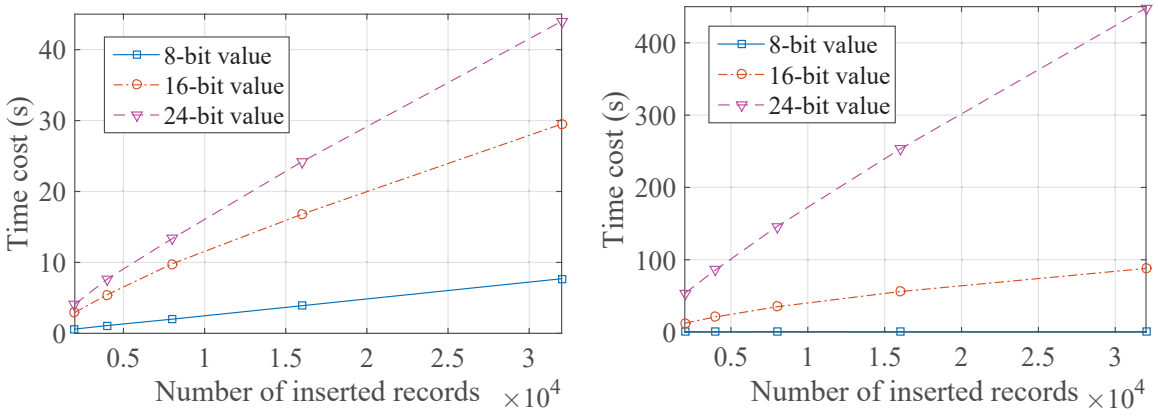
(d) Verification object

Fig. 5.6: Overhead generated by Search.

5.6.3 Insertion Time

We pre-load 160K amount of records and assess the insertion efficiency in terms of indexes and ADS. In Figure 5.7, we can find that as the number of inserted records increases, the time cost grows in similar proportions. We can see that when the bit count achieves up to 24, the ADS takes much more time to compute since the amount

of prime numbers becomes larger.



(a) Indexes (b) ADS

Fig. 5.7: Time cost of Insert.

5.6.4 Gas Consumption

Table 5.2: Gas cost of smart contract

Operations	**Gas cost**
Deployment	745,346 gas
Data insertion	29,144 gas
Result verification	94,531 gas

We list the gas cost of the smart contract conducted on Rinkeby testnet in Table 5.2. The data insertion in our design is very cheap in gas since it only needs to change one storage value of the ADS on smart contract. It only costs $29,144$ gas per time regardless of the amount of items to insert. Regarding the gas of result verification for an equality search, it costs around $94,531$ gas, i.e., approximately 0.28$ when ETH is at the price of 3000$. The gas appears practically low because the verification of the ADS can be finished in $O(\lambda)$.

5.7 Chapter Summary

Many traditional verifiable SSE schemes are limited to keyword-file search and leave the range search blank. In addition, the mutual distrusted scenario where data users may lie about the result verification necessitates the public verification of search results. In this chapter, we first propose a fair framework for encrypted search based on blockchain. We then design the SORE scheme to handle numerical search and enable the verifiable search using multiset hash and RSA accumulator. At last, we prove the security via formal analysis and show the efficiency through extensive experiments.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Blockchain can be utilized to supplement and improve the untrusted cloud storage due to its capability of trusted storage and computation. We propose novel methods to solve the verifiable search problem atop the blockchain over three types of data, i.e., native blockchain data, outsourced raw graph data and outsourced encrypted numerical data.

To conquer the authenticity problem over native blockchain data, we propose a Verifiable Query Layer (VQL) that can be deployed in the cloud to provide both efficient and verifiable data query services for blockchain systems. The middleware layer extracts data from the underlying blockchain system and efficiently reorganizes them in databases. To prevent falsified data from being stored in the middleware, a cryptographic fingerprint is calculated based on each constructed database. The database fingerprint will be first verified by miners and then written into the blockchain. Moreover, public users can verify the entire databases or several databases that interest them in the middleware layer. We implement VQL together with the verification

schemes and conduct extensive experiments based on Ethereum testnet. The evaluation results demonstrate the efficiency of our design.

For the outsourced raw graph data, we conduct pioneering research on authenticated queries on the blockchain-assisted cloud, where the data owner outsources original data to cloud servers and stores some metadata on the blockchain. The primary challenge is how to design an ADS that supports authenticated queries and can be easily maintained by the blockchain. To this end, we propose a novel ADS, named PAGB, based on the RSA accumulator and completeness set. It can also prevent the original data from being revealed to the public through blockchain or irrelevant queries. We further optimize our design to be more efficient in terms of communication and computation. The effectiveness and efficiency of PAGB are verified via theoretical analysis and extensive experiments.

To enable the verifiable search over encrypted numerical data, we take the first step to study the public verification problem atop the blockchain. We design a novel verifiable SSE scheme named Slicer based on a Succinct Order-Revealing Encryption (SORE) scheme to achieve range search on numerical data. The fairness of search in the mutual distrusted scenario, where data users may maliciously deny the results after the local result verification, can be perfectly guaranteed by the public verification on the smart contract. Moreover, we achieve the forward-security during the data insertion via the incorporation of trapdoor permutation. We illustrate the security and practicality of our design through rigorous analysis and extensive evaluations respectively.

Since the three contributions in this thesis are based on different system models and threat models, they cannot be simultaneously equipped in one system. Instead,

they can be seen as an incremental process in which application requirements of security and versatility are gradually raised. It is noted that although this thesis takes Ethereum or its testnets as examples to illustrate the feasibility of our designs, our work can be generalized to any permissionless or permissioned blockchain systems with smart contract functions. It does not make any difference what consensus mechanism the system adopts as well.

6.2 Future Work

Blockchain has brought novel insights and new challenges to the reliable search on untrusted clouds. Based on our work, we indicate three future research directions as follows.

More search types. Besides the verifiable query types over graph data that have been addressed in this thesis, other complicated but interesting ones can be investigated. For example, the shortest path query authentication in the graph is challenging since the computation process involves all nodes and edges. Meanwhile, some popular data types like spatial data and time-series data and their corresponding queries are well worth the exploration.

More privacy-preserving schemes. In addition to the symmetric encryption scheme in the thesis, data privacy can also be protected by the asymmetric encryption scheme or differential privacy technology. Since these schemes involve more complicated constructions, it may need some innovative ideas to integrate them with the reliable search on the top of blockchain.

More trusted techniques. Similar to the trusted environment provided by the smart contract on the blockchain, some other techniques can also provide trusted

computation or storage like trusted execution environment (TEE) or multi-party secure computation. It will be interesting and challenging to combine these methods to enhance the query verification.

Bibliography

[1] Blockchain.info. https://blockchain.info/.

[2] Coinbase: Toshi project. https://github.com/martindale/toshi.

[3] Etherscan. https://etherscan.io/.

[4] Merkle patricia tree. https://github.com/ethereum/wiki/wiki/Patricia-Tree.

[5] Neo4j. https://neo4j.com/.

[6] Provenance. https://www.provenance.org/.

[7] TITAN. http://titan.thinkaurelius.com/.

[8] Muneeb Ali, Jude C Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A global naming and storage system secured by blockchains. In *Proc. of USENIX Annual Technical Conference (ATC)*, pages 181–194, 2016.

[9] Lindsey Allen, Panagiotis Antonopoulos, Arvind Arasu, Johannes Gehrke, Joachim Hammer, James Hunter, Raghav Kaushik, Donald Kossmann, Jonathan Lee, Ravi Ramamurthy, et al. Veritas: Shared verifiable databases and tables in the cloud. In *9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[10] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W Hare, Jan Hidders, Victor E Lee, Bei Li, Leonid Libkin, Wim Martens, et al.

PG-Keys: Keys for property graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2423–2436, 2021.

[11] Panagiotis Antonopoulos, Raghav Kaushik, Hanuma Kodavalla, Sergio Rosales Aceves, Reilly Wong, Jason Anderson, and Jakub Szymaszek. SQL Ledger: Cryptographically verifiable data in Azure SQL database. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2437–2449, 2021.

[12] Muhammad U Arshad, Ashish Kundu, Elisa Bertino, Krishna Madhavan, and Arif Ghafoor. Security of graph data: hashing schemes and definitions. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pages 223–234, 2014.

[13] Gbadebo Ayoade, Vishal Karande, Latifur Khan, and Kevin Hamlen. Decentralized IoT data management using blockchain and trusted execution environment. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 15–22. IEEE, 2018.

[14] Sumeet Bajaj, Anrin Chakraborti, and Radu Sion. ConcurDB: Concurrent query authentication for outsourced databases. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[15] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 480–494. Springer, 1997.

[16] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, 2014.

[17] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'neill. Order-preserving symmetric encryption. In *Annual International Conference on the*

Theory and Applications of Cryptographic Techniques, pages 224–241. Springer, 2009.

[18] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

[19] Raphael Bost. $\Sigma o\phi o\varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1143–1154, 2016.

[20] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. *IACR Cryptol. ePrint Arch.*, 2016:62, 2016.

[21] Chengjun Cai, Jian Weng, Xingliang Yuan, and Cong Wang. Enabling reliable keyword search in encrypted decentralized storage with fairness. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[22] Philippe Camacho and Alejandro Hevia. Short transitive signatures for directed trees. In *Cryptographers' Track at the RSA Conference*, pages 35–50. Springer, 2012.

[23] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: data structures and implementation. In *NDSS*, volume 14, pages 23–26. Citeseer, 2014.

[24] Qi Chai and Guang Gong. Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In *2012 IEEE International Conference on Communications (ICC)*, pages 917–922. IEEE, 2012.

[25] Hongzhi Chen, Bowen Wu, Shiyuan Deng, Chenghuan Huang, Changji Li, Yichao Li, and James Cheng. High performance distributed OLAP on property graphs with grasper. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2705–2708, 2020.

[26] Qian Chen, Haibo Hu, and Jianliang Xu. Authenticated online data integration services. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 167–181. ACM, 2015.

[27] Nathan Chenette, Kevin Lewi, Stephen A Weis, and David J Wu. Practical order-revealing encryption with limited leakage. In *International conference on fast software encryption*, pages 474–493. Springer, 2016.

[28] Dwaine Clarke, Srinivas Devadas, Marten Van Dijk, Blaise Gassend, and G Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *International conference on the theory and application of cryptology and information security*, pages 188–207. Springer, 2003.

[29] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[30] Hermann de Meer, Henrich C Pöhls, Joachim Posegga, and Kai Samelin. Redactable signature schemes for trees with signer-controlled non-leaf-redactions. In *International Conference on E-Business and Telecommunications*, pages 155–171. Springer, 2012.

[31] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. Practical private range search revisited. In *Proceedings of the 2016 International Conference on Management of Data*, pages 185–198, 2016.

[32] Ali Dorri, Salil S Kanhere, Raja Jurdak, and Praveen Gauravaram. Blockchain for IoT security and privacy: The case study of a smart home. In *Proc. of IEEE PerCom*, pages 618–623, 2017.

[33] Andreas Erwig, Marc Fischlin, Martin Hald, Dominik Helm, Robert Kiel, Florian Kübler, Michael Kümmerlin, Jakob Laenge, and Felix Rohrbach. Redactable graph hashing, revisited. In *Australasian Conference on Information Security and Privacy*, pages 398–405. Springer, 2017.

[34] Xinrui Ge, Jia Yu, Hanlin Zhang, Chengyu Hu, Zengpeng Li, Zhan Qin, and Rong Hao. Towards achieving keyword search over dynamic encrypted cloud data with symmetric-key based verification. *IEEE Transactions on Dependable and Secure computing*, 2019.

[35] Rosario Gennaro, Shai Halevi, and Tal Rabin. Secure hash-and-sign signatures without the random oracle. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 123–139. Springer, 1999.

[36] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3–16, 2016.

[37] Michael T Goodrich, Roberto Tamassia, and Jasminka Hasić. An efficient dynamic and distributed cryptographic accumulator. In *International Conference on Information Security*, pages 372–388. Springer, 2002.

[38] Michael T Goodrich, Roberto Tamassia, and Nikos Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.

[39] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 361–378, 2019.

[40] Yu Guo, Chen Zhang, and Xiaohua Jia. Verifiable and forward-secure encrypted search using blockchain techniques. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2020.

[41] Shengshan Hu, Chengjun Cai, Qian Wang, Cong Wang, Xiangyang Luo, and Kui Ren. Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 792–800. IEEE, 2018.

[42] Yang Ji, Cheng Xu, Ce Zhang, and Jianliang Xu. Dcert: towards secure, efficient, and versatile blockchain light clients. In *Proceedings of the 23rd conference on 23rd ACM/IFIP International Middleware Conference*, pages 269–280, 2022.

[43] Harry Kalodner, Malte Möser, Kevin Lee, Steven Goldfeder, Martin Plattner, Alishah Chator, and Arvind Narayanan. BlockSci: Design and applications of a blockchain analysis platform. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2721–2738, 2020.

[44] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 965–976, 2012.

[45] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340, 2016.

[46] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving

smart contracts. In *IEEE Symposium on Security and Privacy (SP)*, pages 839–858, 2016.

[47] Ashish Kundu and Elisa Bertino. How to authenticate graphs without leaking. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 609–620, 2010.

[48] Ashish Kundu and Elisa Bertino. Privacy-preserving authentication of trees and graphs. *International journal of information security*, 12(6):467–494, 2013.

[49] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314. IEEE, 2018.

[50] Yoad Lewenberg, Yoram Bachrach, Yonatan Sompolinsky, Aviv Zohar, and Jeffrey S Rosenschein. Bitcoin mining pools: A cooperative game theoretic analysis. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 919–927. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

[51] Kevin Lewi and David J Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1167–1178, 2016.

[52] Han Li, Hongliang Zhou, Hejiao Huang, and Xiaohua Jia. Verifiable encrypted search with forward secure updates for blockchain-based system. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 206–217. Springer, 2020.

[53] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In *International Conference on Applied Cryptography and Network Security*, pages 253–269. Springer, 2007.

[54] Yang Li, Kai Zheng, Ying Yan, Qi Liu, and Xiaofang Zhou. EtherQL: A query layer for blockchain system. In *International Conference on Database Systems for Advanced Applications*, pages 556–567. Springer, 2017.

[55] Zhetao Li, Jiawen Kang, Rong Yu, Dongdong Ye, Qingyong Deng, and Yan Zhang. Consortium blockchain for secure energy trading in industrial internet of things. *IEEE Transactions on Industrial Informatics*, 2017.

[56] Bin Liu, Xiao Liang Yu, Shiping Chen, Xiwei Xu, and Liming Zhu. Blockchain based data integrity service framework for IoT data. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 468–475. IEEE, 2017.

[57] Weijie Liu, Peng Zhou, Zhe Zhao, Zhiruo Wang, Qi Ju, Haotang Deng, and Ping Wang. K-bert: Enabling language representation with knowledge graph. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 2901–2908, 2020.

[58] Wenhe Liu, Dong Gong, Mingkui Tan, Javen Qinfeng Shi, Yi Yang, and Alexander G Hauptmann. Learning distilled graph for large-scale social network data clustering. *IEEE Transactions on Knowledge and Data Engineering*, 32(7):1393–1404, 2019.

[59] Xueqiao Liu, Guomin Yang, Yi Mu, and Robert H Deng. Multi-user verifiable searchable symmetric encryption for cloud storage. *IEEE Transactions on Dependable and Secure Computing*, 17(6):1322–1332, 2018.

[60] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[61] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and

Alberto Granzotto. BigchainDB: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[62] Antonio Messina, Antonino Fiannaca, Laura La Paglia, Massimo La Rosa, and Alfonso Urso. BioGraph: a web application and a graph database for querying and analyzing bioinformatics resources. *BMC systems biology*, 12(5):75–89, 2018.

[63] Muhammad Muzammal, Qiang Qu, and Bulat Nasrulin. Renovating blockchain with distributed databases: An open source system. *Future Generation Computer Systems*, 90:105–117, 2019.

[64] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[65] Lan Nguyen. Accumulators from bilinear pairings and applications. In *Cryptographers' track at the RSA conference*, pages 275–292. Springer, 2005.

[66] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[67] HweeHwa Pang and K-L Tan. Authenticating query results in edge computing. In *Proceedings. 20th International Conference on Data Engineering*, pages 560–571. IEEE, 2004.

[68] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013.

[69] Subrata Paul, Chandan Koner, Robiul Islam Kabir, and Anirban Mitra. Issues of knowledge management in deep web and its graph-based analysis. In *Proceedings of the 3rd International Conference on Communication, Devices and Computing*, pages 213–223. Springer, 2022.

[70] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[71] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[72] Kui Ren, Yu Guo, Jiaqi Li, Xiaohua Jia, Cong Wang, Yajin Zhou, Sheng Wang, Ning Cao, and Feifei Li. HybrIDX: New hybrid index for volume-hiding range queries in data outsourcing services. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 23–33. IEEE, 2020.

[73] Kui Ren, Cong Wang, and Qian Wang. Security challenges for the public cloud. *IEEE Internet Computing*, 16(1):69–73, 2012.

[74] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security and Privacy (S&P)*, pages 459–474, 2014.

[75] Azam Soleimanian and Shahram Khazaei. Publicly verifiable searchable symmetric encryption based on efficient cryptographic components. *Designs, Codes and Cryptography*, 87(1):123–147, 2019.

[76] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, volume 71, pages 72–75, 2014.

[77] Wenhai Sun, Xuefeng Liu, Wenjing Lou, Y Thomas Hou, and Hui Li. Catch you if you lie to me: Efficient verifiable conjunctive keyword search over large dynamic

encrypted cloud data. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 2110–2118. IEEE, 2015.

[78] Hamed Tabrizchi and Marjan Kuchaki Rafsanjani. A survey on security challenges in cloud computing: issues, threats, and solutions. *The journal of supercomputing*, 76(12):9493–9532, 2020.

[79] Feng Tian. An agri-food supply chain traceability system for china based on rfid & blockchain technology. In *Proc. of IEEE Service Systems and Service Management (ICSSSM)*, pages 1–6, 2016.

[80] Haixin Wang, Cheng Xu, Ce Zhang, Jianliang Xu, Zhe Peng, and Jian Pei. vchain+: Optimizing verifiable blockchain boolean range queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1927–1940. IEEE, 2022.

[81] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

[82] Haotian Wu, Zhe Peng, Songtao Guo, Yuanyuan Yang, and Bin Xiao. VQL: Efficient and verifiable cloud query services for blockchain systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1393–1406, 2021.

[83] Songrui Wu, Qi Li, Guoliang Li, Dong Yuan, Xingliang Yuan, and Cong Wang. ServeDB: Secure, verifiable, and efficient range queries on outsourced database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 626–637. IEEE, 2019.

[84] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 71–82, 2015.

[85] Qi Xia, Emmanuel Boateng Sifah, Kwame Omono Asamoah, Jianbin Gao, Xiaojiang Du, and Mohsen Guizani. Medshare: Trust-less medical data sharing among cloud service providers via blockchain. *IEEE Access*, 5:14757–14767, 2017.

[86] Yuhang Xia and Chenglin Sun. Property graph database modeling and application of electronic medical record. In *2018 Eighth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC)*, pages 963–967. IEEE, 2018.

[87] Cheng Xu, Jianliang Xu, Haibo Hu, and Man Ho Au. When query authentication meets fine-grained access control: A zero-knowledge approach. In *Proceedings of the 2018 International Conference on Management of Data*, pages 147–162. ACM, 2018.

[88] Cheng Xu, Ce Zhang, and Jianliang Xu. vChain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 International Conference on Management of Data*, pages 141–158, 2019.

[89] Yuqin Xu, Shangli Zhao, Lanju Kong, Yongqing Zheng, Shidong Zhang, and Qingzhong Li. ECBC: A high performance educational certificate blockchain with efficient query. In *International Colloquium on Theoretical Aspects of Computing*, pages 288–304. Springer, 2017.

[90] Xingliang Yuan, Xinyu Wang, Cong Wang, Baochun Li, Xiaohua Jia, et al. Enabling encrypted rich queries in distributed key-value stores. *IEEE Transactions on Parallel and Distributed Systems*, 30(6):1283–1297, 2018.

[91] Bo Zhang, Boxiang Dong, and Wendy Hui Wang. Integrity authentication for SQL query evaluation on outsourced databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[92] Ce Zhang, Cheng Xu, Haixin Wang, Jianliang Xu, and Byron Choi. Authenticated keyword search in scalable hybrid-storage blockchains. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 996–1007. IEEE, 2021.

[93] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. GEMˆ2-tree: A gas-efficient structure for authenticated range queries in blockchain. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 842–853. IEEE, 2019.

[94] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 863–880. IEEE, 2017.

[95] Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. VABKS: Verifiable attribute-based keyword search over outsourced encrypted data. In *IEEE INFOCOM 2014-IEEE conference on computer communications*, pages 522–530. IEEE, 2014.

[96] Fei Zhu, Wei Wu, Yuexin Zhang, and Xiaofeng Chen. Privacy-preserving authentication for general directed graphs in industrial IoT. *Information Sciences*, 502:218–228, 2019.

[97] Jie Zhu, Qi Li, Cong Wang, Xingliang Yuan, Qian Wang, and Kui Ren. Enabling generic, verifiable, and secure data search in cloud services. *IEEE Transactions on Parallel and Distributed Systems*, 29(8):1721–1735, 2018.