The following publication H. Wu, Y. Tang, Z. Shen, J. Tao, C. Lin and Z. Peng, "TELEX: Two-Level Learned Index for Rich Queries on Enclave-Based Blockchain Systems," in IEEE Transactions on Knowledge and Data Engineering, vol. 37, no. 7, pp. 4299-4313, July 2025 is available at https://doi.org/10.1109/TKDE.2025.3564905.

TELEX: Two-Level Learned Index for Rich Queries on Enclave-based Blockchain Systems

Haotian Wu, Yuzhe Tang, Zhaoyan Shen, Jun Tao, Chenhao Lin, and Zhe Peng

***Abstract*—Blockchain has become a popular paradigm for secure and immutable data storage. Despite its numerous applications across various fields, concerns regarding the user privacy and result integrity during data queries persist. Additionally, the need for rich query functionalities to harness the full potential of blockchain data remains an area ripe for exploration. In order to address these challenges, our paper first utilizes a framework based on the Trusted Execution Environment (TEE) and oblivious RAM technique to achieve both privacy and data integrity. To enhance the query efficiency over the entire blockchain, we then devise a two-level learned indexing methodology named TELEX within the TEE for both integer and string keys. We also propose different query processing algorithms for versatile query types, including exact queries, aggregate queries, Boolean queries, and range queries. By implementing the prototype and conducting extensive evaluation, we demonstrate the feasibility and remarkable improvement in efficiency compared to existing solutions.**

***Index Terms*—Blockchain, learned index, rich queries, trusted execution environment.**

I. Introduction

In a world where centralized authorities are deemed less trustworthy or even risky, blockchains have emerged recently as a decentralized infrastructure with data transparency, immutability, and other desirable features for societal critical applications. Blockchains already enjoy wide adoption in the highly-popular domain of decentralized finance (DeFi) [1] and can potentially revolutionize many more (e.g., healthcare [2] and supply chain management [3]). In these applications, the capabilities of understanding blockchain data and extracting insight are essential. In practice, this demand has spawned a good number of blockchain remote procedure call (RPC) services or query services, such as blockchain.info for Bitcoin [4], and etherscan.io, infura.io for querying Ethereum data [5], etc. These services index blockchain data and respond to users' queries. However, the existing query services are operated by third-party entities and assume centralized trust from users. For instance, etherscan users need to fully trust the entity operating etherscan.io, which is a big and risky assumption made in the real world [6]. As an alternative, users can interact with a full node that maintains the entire copy of blockchain data. Nevertheless, there are also trust issues with this query process in terms of query privacy and result integrity. Users can address the result integrity issue by sending queries to multiple service providers or full nodes and accepting answers with majority voting. However, it still suffers from the exposure of clients' privacy.

The privacy exposure is a critical concern that is often overlooked in many instances. As users send their queries to full nodes, they may reveal valuable information that could be exploited by malicious nodes. This not only undermines the confidentiality of user intentions or interests, but also opens up avenues for targeted attacks that can lead to financial loss or reputation damage. For instance, when a user queries his own transaction data, the node can acquire the user's address, allowing adversaries to link multiple transactions and potentially identify his actual identity. Some existing private information retrieval (PIR) techniques can address this privacy issue on the plaintext database like blockchain. Nevertheless, some of them [7] require non-colluding replicated servers, which is challenging to realize in the decentralized blockchain network. The homomorphic encryption is also exploited to implement the PIR protocol [8], but it is impractical for the blockchain system owing to the heavy calculation.

Furthermore, it is also vital to ensure the result integrity when querying blockchain data. In scenarios where nodes may act maliciously or get compromised, the answer provided in response to queries may be incorrect or incomplete. The lack of data integrity can lead to biased conclusions or improper decisions based on incorrect data. To tackle this problem, traditional methods like authenticated data structures (ADS) are widely adopted to provide authenticated queries [9]. They enable the query user to verify the integrity of query results using the proof returned by the server. However, most ADSs do not inherently support the requirement of private queries since they usually need the queried keyword and query results for proof generation.

Lastly, it is necessary to efficiently support a diverse range of query types for generic blockchain systems. This ability to perform timely complex queries on various blockchains can maximize the utility of blockchain applications. Some prior works study the privacy-preserving queries on Bitcoin [10], Ethereum [11] and Zcash [12] based on the Trusted Execution Environment (TEE). They can support some basic queries regarding transactions on the respective system. However,

Manuscript received June 28, 2024. *(Corresponding author: Zhe Peng.)*

Haotian Wu is with the Department of Industrial and Systems Engineering, The Hong Kong Polytechnic University, Hong Kong, China (e-mail: haotian.wu@connect.polyu.hk).

Yuzhe Tang is with the Department of Electrical Engineering and Computer Science, Syracuse University, USA (e-mail: ytang100@syr.edu).

Zhaoyan Shen is with the School of Computer Science and Technology, Shandong University, China (e-mail: shenzhaoyan@sdu.edu.cn).

Jun Tao is with the School of Cyber Science and Engineering, Southeast University, China (e-mail: juntao@seu.edu.cn).

Chenhao Lin is with the Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, China (E-mail: linchenhao@xjtu.edu.cn).

Zhe Peng is with the Department of Industrial and Systems Engineering, The Hong Kong Polytechnic University, and The Hong Kong Polytechnic University Shenzhen Research Institute (e-mail: jeffrey-zhe.peng@polyu.edu.hk).

some more complicated yet popular query types, e.g., range queries and aggregate queries, on generic data have not been investigated in depth.

This paper studies the following research problem: *How to efficiently achieve the privacy and integrity of rich queries on the entire history of a blockchain*. This question has not been well addressed in the existing literature. As mentioned above, existing constructions using only cryptographic protocols incur high overhead. Using TEE or enclave is promising to realize performance efficiency with a small trusted computing base. The only existing work using TEE to build a trustworthy blockchain query service [11] supports only limited query capabilities (i.e., keyword search and Boolean queries) on small blockchain data (i.e., on a few pre-specified blocks). Our research goal is distinct in two senses: 1) We aim to support more and richer queries, including range queries and aggregate queries. 2) We aim at querying "bigger" blockchain data, that is, on the entire blockchain data, not just a few pre-specified blocks.

Supporting rich queries on blockchain big data poses new systems-design challenges. Although Intel SGXv2 has expanded its enclave memory from SGXv1's 128MB to GB-level on the server CPUs, this size is still relatively small compared to the TB-level of infinitely growing blockchain data. With limited memory, it is also demanding to maintain indexes over the entire blockchain data for various query types. To address these challenges, we propose a two-level index architecture inside the enclave called TELEX. On the bottom level, TELEX builds a number of intra-partition indexes, each indexing the block data inside one partition by both string and integer keys. The intra-partition index exploits a novel hash scheme named RSHash for efficient data retrieval within each partition. It realizes both time and space efficiency by utilizing the learned index that captures the intrinsic characteristics of the partition data. On the top level, TELEX utilizes inter-partition indexes to filter the qualified partitions containing the results of the queried key for different queries. In general, our contributions in this paper can be concluded as follows:

- We present a two-level learned indexing method, i.e., *TELEX*, for TEE-enhanced blockchain systems. It can improve the query efficiency for blockchain big data.
- We design a novel learned hashing scheme *RSHash* to construct the second level of TELEX. It can achieve the lowest space cost while reducing the cost of query time.
- We propose effective query processing algorithms to facilitate versatile blockchain queries within TEE, including exact queries, range queries, aggregate queries, and Boolean queries.
- We implement the prototype of a TELEX-enabled blockchain query system and perform extensive evaluations. The experiment results demonstrate the feasibility and superior efficiency of our design.

The remainder of this paper is organized as follows. We first introduce background knowledge in Section II. We then formulate the problem to study in Section III and elaborate on the TELEX design in Section IV. The prototype implementation and evaluation results are presented in Section V. Finally, we review related work in Section VI and conclude the paper in Section VII.

II. Preliminaries

A. Blockchain

Blockchain is a distributed network where each node keeps a digital ledger that records transaction information. The typical structure of a blockchain ledger is a block sequence with each new block containing a set of transactions appended to the tail. Each block is linked to the previous one through a cryptographic hash, which is a digital fingerprint derived from the data of the entire block. The hash resides in the block header, where other block properties, such as the timestamp, the block height and the authenticated metadata of all transactions, are also included. The block body, on the other hand, contains the detailed content of actual transactions. Apart from the currency transfer data in classical cryptocurrencies, e.g., Bitcoin and Ethereum, the transaction data in the block body can also be other types of data in different blockchain-based applications. For instance, when the blockchain is used to track pandemics, the stored data can be users' contact records like identities and locations [13]. Within a public blockchain network, the nodes maintaining a complete record of the ledger are called full nodes. Some full nodes will be regarded as miners if they contribute their resources to create new blocks and validate transactions. Once a new block is generated, its validity and data correctness will be verified via consensus schemes [14] and recognized by all nodes if the verification passes. Therefore, the blockchain data is consistent across all nodes and remains immutable since data tampering on minority nodes will not be accepted by others.

B. Trusted Execution Environment

A Trusted Execution Environment (TEE), commonly known as an enclave, such as Intel's Software Guard Extensions (SGX), provides a layer of trust rooted in hardware [15]. It separates a portion of the memory named enclave page cache (EPC) and protects the computation inside from the external operating system and other softwares. Access to or modification of the code and data within an enclave is restricted to authorized users only. The user must build a secure channel with the enclave and perform remote attestation to verify that the internal code is set as expected. TEE can ensure the integrity and confidentiality of data and program executions even when the underlying server is not fully trusted. There are two generations of Intel SGX differing in scalability, i.e., SGXv1 for clients expanding to SGXv2 for servers. SGXv2 supports GB-level EPC size, which is much larger than SGXv1's 128MB. Although the limitation of EPC can be circumvented by swapping memory pages, it is still advisable to comply with the limitation because the encryption/decryption and integrity validation during swapping incur prohibitively high costs [16].

C. Monotone Minimal Perfect Hash Function

A Monotone Minimal Perfect Hash Function (MMPHF) is a specialized type of hash function that provides a unique

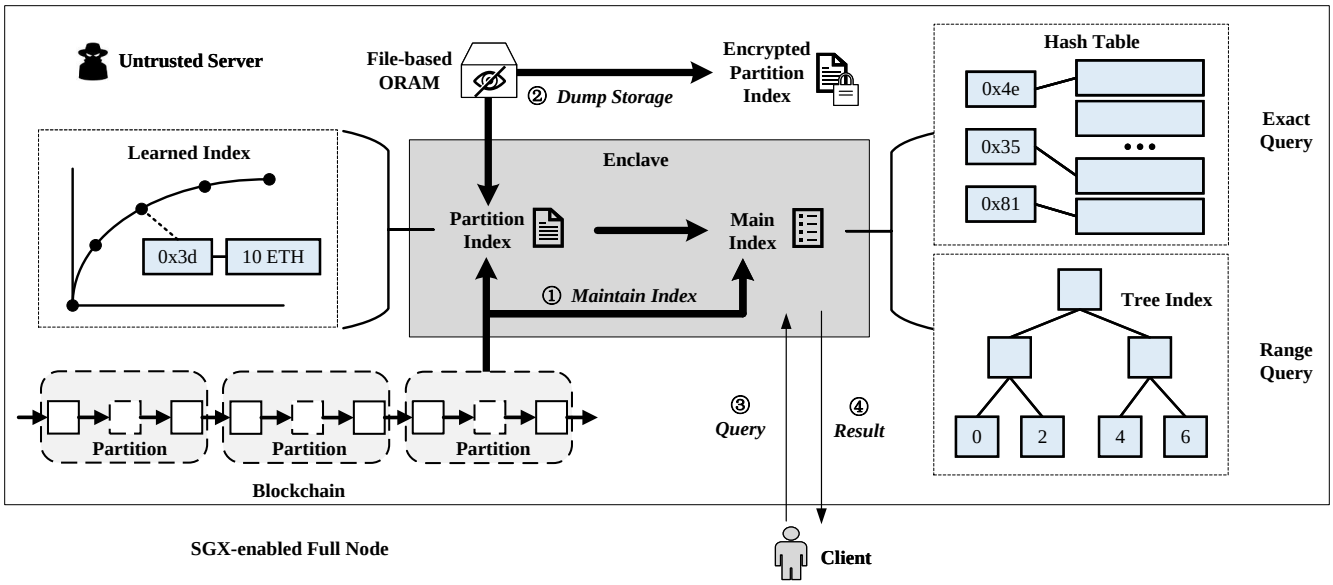


Fig. 1. System architecture.

mapping for a set of keys to their respective ranks within the set [17]. Concretely, given a set S of n keys, each originating from a universe $[u] = \{0, \ldots, u-1\}$, an MMPHF uniquely maps each key to its rank that belongs to $[n]$. If the input element is not in S, it will output an arbitrary answer. This function is characterized by three attributes: it is *perfect* because it guarantees one-to-one correspondence without any collisions; it is *minimal* since its output range is the smallest possible range, i.e., exactly the size of the set; and it is *monotone* because the order of any two keys' outputs follows their inherent order in the universe. The MMPHF exploits the natural order of keys to reduce the space cost down to $O(\log\log\log u)$ bits per key [18], which has been proven to be optimal in a recent work [19]. This remarkable advantage of compact space makes it an excellent choice for many practical applications, such as key-value store [20], data encryption [21], information retrieval [22], and pattern matching [23].

III. Problem Formulation

We consider a query user who interacts with a full node to search the public blockchain database. Our main goal is to guarantee the privacy of the user's query throughout the entire process, thereby preventing any information leakage regarding the user's interests.

A. System Architecture

Figure 1 illustrates the system architecture for privacy-preserving queries over blockchain data. It is composed of two primary entities, i.e., an SGX-enabled full node as the *server* and a user *client*. The server contains two main parts: the trusted enclave marked in dark and the untrusted host in the rest part. As a full node in the blockchain network, the server needs to maintain a complete copy of blockchain data. The data can only be stored outside the enclave because its increasing size easily exceeds the limited size of the enclave. Considering the maintenance of the entire blockchain data incurs substantial storage and computational burden, most users choose to access desired information via the full node to avoid the cost. In return, users can pay the server blockchain coins for the query service. The payments can be enforced by the smart contract as in [24]. To address the user's privacy concerns about his query content, the server will resort to the enclave for confidentiality guarantees. Specifically, based on the blockchain data outside, the enclave will periodically perform the data indexing and process the user's query in a private manner (detailed algorithms are presented in Section IV). Before sending a query, the client needs to initially build a secure channel with the enclave via the Diffie-Hellman key exchange protocol and then verify the integrity of protocol codes via remote attestation [25]. Finally, the query results will be returned to the user through the same channel.

B. Threat Model

We assume the client and the enclave are trusted, which means they are honest with predefined protocols and the channel between them will not leak the communication content. We note that some side-channel attacks towards TEE [26], [27] may compromise its security and confidentiality. However, we exclude these concerns in this paper because they can be well addressed by orthogonal works [28], [29]. As for the underlying blockchain network, we consider it to function properly all the time, ensuring its inherent characteristics of liveness, integrity and immutability. We exclude some attacks targeting the blockchain infrastructure itself, e.g., selfish mining [30] and bribery attacks [31], since they are beyond our study. The enclave will also be provided with the correct header information of the newest block. In this work, we focus on the blockchains where data exists in plaintext form, thus the privacy of data cannot be guaranteed as long as it is accessed. Unfortunately, the out-of-enclave part of the server in our model is regarded as untrusted when providing query services

for users. It can actively deduce from the user's query patterns, e.g., the keyword length or the result volume, to obtain his private information, such as the item he is interested in or even his account address. In addition, the server may also deviate from the original query protocols for his benefits, e.g., returning incomplete or incorrect answers to save the search cost.

C. Design Goals

In this work, we aim to realize the following goals that are crucial for privacy-preserving rich queries on blockchain data:

- *User Privacy.* We need to protect the privacy of the user's query from the full node, ensuring no information regarding the query or its outcomes can be revealed.
- *Result Integrity.* The query result should satisfy the correctness and completeness, i.e., it must contain all qualified answers in the authentic blockchain database.
- *Rich Queries.* We seek to support a variety of queries, including exact queries, range queries, Boolean queries and aggregate queries (max, min, sum, count), on plaintext blockchains.
- *Efficiency.* The time cost should be acceptable and better than simply traversing the entire blockchain.

IV. TELEX Design

In this section, we give the detailed design of TELEX, a novel indexing method within the enclave and corresponding query processing algorithms.

A. Design Rationale

To achieve the goal of private queries on untrusted servers, similar to other works like [10], [11], we leverage the ORAM method atop the enclave to conceal the access patterns during the query processing. In the ORAM, the algorithm usually takes a constant-size data chunk as the minimal unit, therefore we slice the blockchain into a number of partitions to populate these chunks one-to-one. In [11], they provide keyword queries and Boolean queries over Ethereum data within a specified block (or partition) range. We take one more step to get rid of the need of a pre-specified block range, preventing its potential privacy leakage and providing more flexible queries. However, simply searching the entire blockchain is costly, especially in the enclave of limited memory. To this end, we adopt a two-level indexing approach that positions the possible data chunk at the inter-partition index and gets the desired item within the chunk at the intra-partition index.

As shown in Figure 1, the main index is the inter-partition index and each partition index is the intra-partition index. The main index is always maintained inside the enclave, while a partition index is built only when the partition is newly produced. After building, it will never be modified since the blockchain data is immutable. The partition index will be encrypted and stored outside the enclave as a file via the ORAM. It will be reloaded into the enclave again and decrypted during subsequent queries. However, the data transmission between the enclave and the outside server is time-consuming since the encryption and decryption are computationally expensive. Hence, we aim to reduce the size of the generated partition index, which allows us to include more blocks under the same ORAM settings. We observe that the keys within all partitions occupy a significant amount of space and remain constant all the time. Consequently, we devise a novel hashing scheme based on the learned index to largely reduce this space usage while simultaneously supporting faster indexing speeds. Besides, we craft the partition index to support queries on two primary key types, i.e., string keys and integer keys. It is worth noting that partition indexes in our paper not only have the indexing structures, but also contain all corresponding payloads. This can prevent the leakage caused by the payload retrieval after searching the indexes.

B. Basic Workflows

To give a high-level understanding, we begin by describing the basic workflow of providing privacy-preserving rich query services on a full node. Since the enclave has limited memory space, the entire blockchain data is too massive for it to load and index. Therefore, we divide it into a series of partitions, each containing a constant number of blocks. The blocks in each partition will be unified for index building and further queries. The generated index and related payloads in each partition, also known as the partition index, can be encrypted using the symmetric encryption scheme and stored in the data chunk during the ORAM access. The number of blocks in the partition is jointly determined by the ORAM data chunk size and the block data size of the blockchain. Specifically, it should be as large as possible to make the most use of space, but it must ensure that even the largest possible partition index can also be fit into a single data chunk. As the miners keep producing blocks, the enclave will periodically check if the newest partition has been filled with valid blocks. If yes, all blocks in this partition will be loaded into the enclave and verified for data integrity. After the verification, the enclave extracts the data and constructs the indexes for various query types.

Algorithm 1 depicts the routine program of processing a new partition performed by the enclave. We denote the blockchain as $\{b_i\}$, where b_i is the i-th block in the blockchain. Similarly, p_j represents the j-th partition. To achieve the design goal of result integrity, we need to make sure the indexes for query response are built from authentic blockchain data. Therefore, it is critical to verify the integrity of the loaded blocks in each partition. The integrity check of blocks involves two aspects, i.e., the integrity of the block data and the correctness of the chain connections. The first requirement needs to guarantee the blockchain data in the block body is consistent with its ADS, e.g., Merkle Tree root, in the block header. Besides, the hash value of the block should also be correctly calculated based on the ADS and other block properties. The other requirement is to ensure the previous block's hash value is exactly used in the next one's block header. Meanwhile, the hash value of the newest block should be identical to the correct one acknowledged by the entire blockchain network outside. Hence, for each block in the

Algorithm 1 Enclave Routine Program

Input: Blockchain database $\{b_i\}$; a newly filled partition p_j; symmetric encryption key $eKey$.
Output: Main index MI; encrypted partition index ePI_j.
Enclave:

1: **for each** $b_i \in p_j$ **do**
2: Check the correctness of b_i's header against its body data and b_{i-1}'s hash;
3: **end for**
4: Get the authentic hash value h_{auth} of the highest block from trusted sources;
5: Check whether h_{auth} equals the highest block's hash value in p_j;
6: Traverse all blocks in p_j, accordingly build partition index PI_j and update main index MI;
7: Pad PI_j;
8: ePI_j =Enc$(eKey, PI_j)$;
9: Store ePI_j on the server via ORAM;

partition p_j, the enclave will check the correctness of the header calculation based on its own transaction data and the previous block's hash value. To check the correctness of the newest block, as done in recent works [10], [11], we assume the enclave can get the true hash value of the latest block h_{auth} from authentic sources, e.g., several large mining pools. As long as the hash value of the highest block in p_j equals h_{auth}, all the blocks and chain connections can be considered correct.

Then the enclave will construct the corresponding partition index PI_j and update the main index MI by traversing the data in all blocks, which will be elaborated on in the next subsection. The partition index PI_j will be padded and encrypted via the symmetric encryption scheme using the secret key, which is denoted as $eKey$. Note that every partition key has its own $eKey$ and it will be randomly updated for each new write/read round. With this mechanism, when the server returns invalid or outdated partition data, the enclave can easily verify it using the newest $eKey$. Lastly, we dump the encrypted index ePI_j to the server through the ORAM algorithm. Different from the partition index, the main index MI is always maintained in the enclave.

C. Two-Level Learned Index Construction

In this subsection, we present detailed algorithms for indexing blockchain data. The blockchain data hereby is not limited to traditional transaction data, it can also be real-world data such as names and positions in logistics blockchains or files uploaded by users. Specifically, we build the index on each queried attribute of the blockchain data items. Each entry of the index can be regarded as a key-value pair $\langle k, v\rangle$, where the key k is a unique identifier and the value v is the corresponding payload. The key can be in integer or string type while the payload can be a string, a number, or a list of them (e.g., the posting list in the inverted index).

Taking the traditional transaction data as an example, an Ethereum transaction can be seen as a tuple, i.e., $tx = \langle txid, from, to, amount\rangle$, where $txid$ is a hash value as the transaction identifier, $from$ and to are the account addresses of the sender and the receiver, and $amount$ is the number of currency. Our design can realize the following types of queries: 1) If a user wants to get the transaction value via the transaction ID, we can directly take $txid$ and $amount$ as the key and payload of the index entry. Then an exact query with a string key can realize the search. Moreover, based on the index with integer values, we can also provide aggregate queries, including max/min, sum, count and average queries. 2) If the user seeks to acquire all receivers that an account has transferred money to, we can set the index key and payload to $from$ and a posting list of to respectively. When the payload of the index entry is a list, we can additionally support Boolean queries that take several keys and Boolean operators as the input. 3) If the user desires to obtain the IDs of all transactions where the currency values fall within a specified range, then the key and payload in the index can be assigned $amount$ and its transaction ID list $\{txid\}$. Later, a range query can be conducted on these integer keys.

For clear demonstration, we neglect the detailed form of the payload v and focus on the algorithm design on two types of index keys, i.e., strings and numbers, in subsequent descriptions. We also skip the generation process of the payloads, which can be easily realized by direct assignment or inverted index building when traversing the data. Assume that we already have a constructed key-value index denoted as $\{\langle k, v\rangle\}$. Next, according to the data types of k, we present the concrete index construction algorithms on integer keys and string keys in turn.

Learned Partition Index on Integer Keys. An intuitive way to position the desired payload within the partition is to do a binary search. This method needs two arrays to store the sorted keys and corresponding payloads. The matched index in the key array can be directly used to get the corresponding answer in the payload array. However, this method caters to the search on arbitrary data at the cost of extra time. Because the blockchain data within the partition is immutable, we can expedite the search and reduce the space cost by learning the intrinsic distribution of data in advance. In addition, as we describe in the design rationale, we also aim to reduce the space of key storage to make the best use of data transfer. To this end, as illustrated in Figure 2, we propose RSHash, a novel monotone minimal perfect hash function (MMPHF) based on the learned index RadixSpline (RS) [32] to efficiently hash the sorted keys to their respective positions. In this way, the key array is no longer stored and only the structure of RSHash remains instead. The consequence of the removal of keys is that our RSHash does not support the query of a non-existent key. This is acceptable in the exact query since the main index will do the filtering.

RS leverages the radix table to quickly position the two spline points where the key falls, and does a linear interpolation between them to approximate the index. Different from the conventional usage of RS that conducts a binary search within the error-bounded range after the interpolation, we abort the costly binary search to accelerate the process further. It can be seen from the right of the figure that we regard each

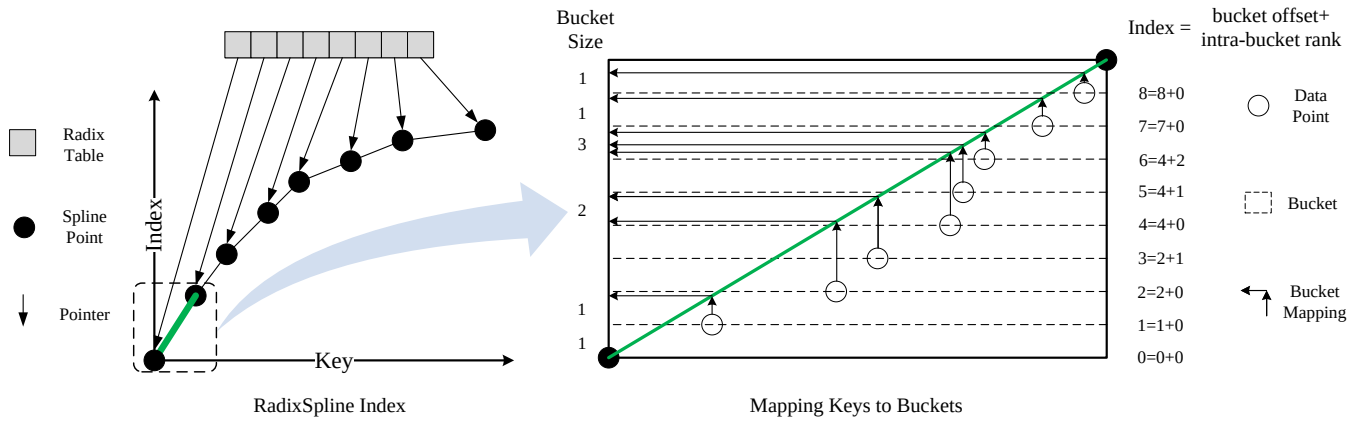


Fig. 2. Illustration of RSHash.

index as a bucket and map each key to a bucket via RS. As noted beside the left axis, some buckets are empty while some contain more than one key. We can observe that, the index of a key can be seen as a sum of how many keys are ahead and its rank in its bucket. The first term is called the *bucket offset* and the second is named *intra-bucket rank*. Next, we resort to two data structures to record these two terms.

Algorithm 2 presents the detailed steps of the partition index building. We first utilize a bit vector to store and answer the information of all buckets. Note that such a succinct data structure only stores a sequence of 0 and 1 with very efficient space and time, which can support two important queries, i.e., rank and select queries, in constant time. Concretely, given a bit vector of size n and a bit $b \in \{0, 1\}$, the $rank_b(pos)$ query returns how many b-bits exist in front of the position pos. Conversely, the $select_b(i)$ query tells the position of the i-th b-bit in the vector. In our design, we represent every bucket size (including zero of the empty one) in unary coding and append them to the bit vector. In the unary coding, we choose to write a number n as n zeros followed by a one, i.e., $unary(n) = 0^n1$ (appears in Line 13 and 16). With this bit vector, given the i-th bucket, we can get its bucket offset by $rank_0(select_1(i))$ and its bucket size via $rank_0(select_1(i+1)) - rank_0(select_1(i))$.

As RS can guarantee an error bound, denoted as e, during the interpolation, the size of each bucket will be no more than e. Thus, the possible intra-bucket rank of a key is strictly constrained within e. In our system, these restricted ranks are stored in Bumped Ribbon Retrieval (BuRR) [33]. This data structure can efficiently map a set of keys to a set of r-bit values with less than 1% space overheads in practice. Specifically, it first calculates a matrix based on the set of keys during initialization, and then answers the corresponding r-bit value for each key using the matrix. In our design, r equals $\log e$ and we only record the ranks within the buckets containing more than one key as no rank is needed in a one-key bucket.

As shown in Algorithm 2, we first initialize a RadixSpline model RS and train it to fit all keys (Line 2-3). Then the algorithm will use the trained RS to predict a bucket with index pos for each key (Line 9). When a new bucket is produced, the size of all the buckets from the last bucket, including the empty buckets along the way, will be recorded in the bit vector BV in the unary coding (Line 11-16). If the size of the last bucket is larger than 1, we need to leverage the retrieval data structure RDS to store the intra-bucket rank j of all keys within it (Line 17-21). The output index includes the RadixSpline index RS, the bit vector BV, the retrieval data structure RDS and the payload list $vList$. The radix table lookup and the linear interpolation in RS both finish in constant time. The $rank$ and $select$ queries on the bit vector are also constant-time functions as mentioned before. BuRR needs $O(1+rW/\log n)$ time to retrieve the ranks, where $W = O(\log n)$ is a parameter used in the ribbon retrieval. In summary, the complexity of the query time of our RSHash is $O(1)$.

Learned Partition Index on String Keys. To process the string keys, we need to transform them into integers so that the RSHash method can be directly applied. Therefore, we also focus on the situations where only a seen string key will be queried. Instead of intuitively converting each character into a 7-bit ASCII number and treating a string as a number in base-128, we seek to map the string to a number much more efficiently. An alphabet reduction approach was proposed in [17] to complete the transformation. It will sort the strings first and only focus on the branching characters, i.e., the first different characters, of every two adjacent strings. We borrow their example of a set of strings $\{\mathsf{shoppers}, \mathsf{shopping}, \mathsf{shops}\}$ for illustration. Apparently, the longest common prefix (LCP) is shop and they only need to transform the rest parts, i.e., $\{\mathsf{pers}, \mathsf{ping}, \mathsf{s}\}$. The branching character set Σ in alphabetical order will be $\{\mathsf{e}, \mathsf{i}, \mathsf{p}, \mathsf{s}\}$ and its size $\sigma = |\Sigma| = 4$. Then each character will be represented by its index in Σ if present, or by 0 if absent. For instance, ping is mapped to $2\sigma^3 + 1\sigma^2 + 0\sigma^1 + 0\sigma^0 = 144$. In this way, each string will be mapped to a number less than 4^4, which is much smaller than the upper limit of 128^8 (8 is the longest length) in the intuitive method. A lower upper limit can help cover more strings and reduce the calculation. This approach works in the situation where no updates on the strings occur such as in our blockchain scenario.

Inspired by this, we adopt a more aggressive strategy, as shown in Algorithm 3, to further lower the upper limit in two

Algorithm 2 Partition Index Building on Integer Keys

Input: Key-value index $\{\langle k, v\rangle\}$.
Output: Partition index PI.
Enclave:

1: Sort $\{\langle k, v\rangle\}$ in ascending order of k;
2: Initialize a RadixSpline RS;
3: RS.learn($\{k\}$);
4: Initialize a bit vector BV;
5: Initialize a retrieval data structure RDS;
6: $vList \leftarrow \phi$; ▷ payload list
7: $lastPos \leftarrow -1$; $lasti \leftarrow 0$;
8: **for each** $\langle k_i, v_i\rangle \in \{\langle k, v\rangle\}$ **do**
9: $pos \leftarrow \lfloor RS.\text{Predict}(k_i)\rfloor$; ▷ predicted bucket
10: **if** $pos > lastPos$ **then**
11: $lastSize \leftarrow i - lasti$;
12: **if** $lastSize > 0$ **then** ▷ last non-empty bucket
13: BV.append(unary($lastSize$));
14: **end if**
15: $emptys \leftarrow pos - lastPos - 1$;
16: BV.append(unary$(0) * emptys$); ▷ empty buckets
17: **if** $lastSize > 1$ **then**
18: **for each** $j \in [0, lastSize - 1]$ **do**
19: $RDS[k_{lasti+j}] \leftarrow j$;
20: **end for**
21: **end if**
22: $lasti \leftarrow i$;
23: **end if**
24: $lastPos \leftarrow pos$;
25: $vList$.append(v_i);
26: **end for**
27: $PI \leftarrow (RS, BV, RDS, vList)$;

Algorithm 3 Partition Index Building on String Keys

Input: Key-value index $\{\langle k, v\rangle\}$.
Output: Partition index PI.
Enclave:

1: Remove the LCP of $\{k\}$;
2: Initialize a list of ordered sets $\{\Sigma_j\}$;
3: **for each** $k_i \in \{k\}$ **do**
4: $bPos, bChars \leftarrow$ Branching(k_i, k_{i+1});
5: Add $bChars$ to Σ_{bPos} in alphabetical order;
6: **end for**
7: $jList \leftarrow \{j \mid \Sigma_j! = \phi\}$;
8: $\sigma \leftarrow max(\{|\Sigma_j|\})$;
9: **for each** $k_i \in \{k\}$ **do** ▷ String2Integer
10: $k_i' \leftarrow 0$;
11: **for** $char_j \in k_i$ **and** $j \in jList$ **do**
12: $k_i' \leftarrow k_i' * \sigma + \text{index}_j(char_j)$;
13: **end for**
14: **end for**
15: Invoke Algorithm 2;

Algorithm 4 Main Index Building for Exact Query

Input: Key-value index $\{\langle k, v\rangle\}$; partition number pn; the main index MI.
Output: Updated main index MI.
Enclave:

1: **for each** $\langle k, v\rangle \in \{\langle k, v\rangle\}$ **do**
2: $bm_k = MI[k]$;
3: $bm_k[pn] = 1$;
4: $MI[k] = bm_k$;
5: **end for**

terms. First, we only focus on the positions where branching characters are located. Still in the above example, pers and ping produce the branching characters e and i at position 1, while ping and s have the different characters p and s at position 0. Such information of branching position and characters can be obtained via the function $Branching$ at Line 4. Then we only do the mapping at the positions 0 and 1, and neglect the characters behind (Line 7). The other point is that we maintain an individual Σ for each position, i.e., $\Sigma_0 = \{\mathsf{p}, \mathsf{s}\}$ and $\Sigma_1 = \{\mathsf{e}, \mathsf{i}\}$, and assign the maximum size to σ, i.e., $\sigma = max(\{|\Sigma_i|\})$ where $i \in \{0, 1\}$. During the mapping, we use the index where the character is located in its location's Σ if present, or 0 if absent. For example, ping will be mapped to $0\sigma^1 + 1\sigma^0 = 1$, and pers and s are 0 and 2 respectively (Line 11-13). The upper limit is reduced to 2^2, which is extremely compact compared to the intuitive way. After the conversion to numbers, the string keys will proceed with the construction of the partition index exactly as Algorithm 2 does. It is noted that the extra information produced by the alphabet reduction, including the LCP, σ, the list of Σ and the corresponding $jList$, will also be stored in the partition index.

Main Index for Exact Query. Recall that before loading the partitions, we first need the main index to filter the qualified ones. To support the exact query, we construct the main index upon a traditional HashTable regardless of whether it is a string key or an integer key. It will store the relationships between the key and the partitions where its payloads reside. To further reduce the space cost of the main index, we leverage the bitmap to represent the partition list. Suppose we have a total of n partitions and the partition number $pn = 1, 2, \cdots, n$. Then a key k's bitmap is $bm_k = b_{k,1}b_{k,2}\cdots b_{k,n}$, where $b_{k,i} \in \{0, 1\}$ represents whether k exists in the i-th partition. Here 1 means presence, while 0 denotes absence. We outline the process of constructing the main index in Algorithm 4. It traverses all key-value pairs and sets the pn-th bit of each key's bitmap to 1.

Main Index for Range Query. In this paper, we only focus the range query on integer keys. Different from the exact query, the main index for the range query should be able to filter the partitions satisfying the order condition. Algorithm 5 presents the process of the main index building. To realize this requirement, we use two B+ trees, i.e., $minTree$ and $maxTree$, to record each partition's minimum and maximum value of k respectively (Line 2-4). For instance, when a user wants to find all keys smaller than k, $minTree$ can help rule out the partitions whose minimum value is no less than k. In addition, $maxTree$ can ensure that all items in the partitions whose maximum value is smaller that k can be

Algorithm 5 Main Index Building on Range Query

Input: Key-value index $\{\langle k, v\rangle\}$; partition number pn; the main index MI.
Output: Updated main index MI.
Enclave:
1: $minTree, maxTree \leftarrow MI$;
2: $k_{min} \leftarrow min(\{k\})$; $k_{max} \leftarrow max(\{k\})$;
3: $minTree$.insert(k_{min}, pn);
4: $maxTree$.insert(k_{max}, pn);
5: $MI \leftarrow minTree, maxTree$;

Algorithm 6 Exact Search on Partition Index

Input: Query key k; partition index PI.
Output: Query result res.
Enclave:
1: $RS, BV, RDS, vList \leftarrow PI$;
2: $pos \leftarrow \lfloor RS$.Predict($k$)$\rfloor$;
3: $offset, size \leftarrow BV$.at($pos$);
4: **if** $size = 1$ **then**
5: $res \leftarrow vList[offset]$;
6: **else**
7: $res \leftarrow vList[offset + RDS[k]]$;
8: **end if**

included without searching the index. The partition number is associated with its minimum and maximum k and inserted into the corresponding trees.

D. Rich Queries Processing

Exact Search on Partition Index. Before presenting the processing algorithms for different types of queries, we first demonstrate how to perform an exact search on the partition index built by our RSHash. Algorithm 6 describes the process of returning the corresponding v given an integer key k. It first gets the predicted bucket via RS, then obtains its bucket offset and size through BV. If the size is larger than 1, the key index can be obtained by adding up the offset and the intra-bucket rank stored in RDS. Otherwise, the key index is the offset. Note that this search process only applies to integers.

Exact Query and Aggregate Query. In Algorithm 7, we illustrate the processing of exact queries and aggregate queries given a keyword in the integer or string form. The main index is utilized to get a list of filtered partitions using the keyword. It is noted that we do not have to deliberately pad the partition list to prevent the leakage of the partition amount. It is because the ORAM access will load a constant number of data chunks each time and extract all chunks that contain the involved partition in the list. This method can naturally obfuscate the actual number of partitions. Before searching the decrypted partition index, the string keyword needs to be converted to an integer via the $String2Integer$ function (Line 7), which refers to Line 10-13 in Algorithm 3. If it is an exact query, the result list of payloads will be padded and returned. When it is an aggregate query, the list of payloads will do the aggregation first.

Algorithm 7 Exact Query and Aggregate Query

Input: Query type $type$; query keyword w; main index MI; encrypted partition indexes $\{ePI\}$; symmetric encryption key $eKey$.
Output: Query result res.
Client:
1: Send $type, w$ to the enclave;
Enclave:
2: $pnList \leftarrow MI[w]$;
3: $resList \leftarrow \phi$;
4: **for each** $pn \in pnList$ **do**
5: Load ePI_{pn} using ORAM;
6: **if** w **is** string **then**
7: $k \leftarrow$String2Integer(w);
8: **else**
9: $k \leftarrow w$;
10: **end if**
11: $PI_{pn} \leftarrow$Dec($eKey, ePI_{pn}$);
12: $v \leftarrow$PartitionIndexExactSearch(k, PI_{pn});
13: $resList$.append(v);
14: **end for**
15: **if** $type$ **is** exact query **then**
16: Pad $resList$ to res;
17: **end if**
18: **if** $type$ **is** aggregate query **then**
19: $res =$Aggregate($resList$);
20: **end if**
21: return res to the client;

Boolean Query. We demonstrate the Boolean query on the index of string keys as well. It is similar to the exact query in Algorithm 7 except that its input is composed of several keywords and Boolean operators. Hence, instead of a single keyword in the main index at Line 2, we will generate a partition list contributed by all keywords. Note that we consider a negation operator ('NOT') on a keyword as a general keyword. Meanwhile, the other two operators ('AND' and 'OR') will be finally computed after the payload lists of all keywords are obtained.

Range Query. Algorithm 8 shows the query processing algorithm for range queries. To issue a range query request, the user needs to include the query key k and the order condition $c \in \{>, <\}$ in the query. Taking the case of $c =$">" as an example, the keys larger than k should be acquired in the partitions whose maximum key is bigger than k. Meanwhile, keys in the partitions whose minimum key is bigger than k are all qualified without the need of further index searching. Accordingly, the list of partitions that need searching, denoted as $pnList$, can be obtained from $maxTree$, while the list of entirely qualified partitions, i.e., $pnList_q$, comes from $minTree$. Eventually, the partitions in $pnList$ are loaded and decrypted, and those not in $pnList_q$ will be searched on the index (Line 15-20).

As shown in Algorithm 9, the range search on the partition index differs slightly from the exact search because the searched key may not exist. We have to additionally

Algorithm 8 Range Query

Input: Query key k; query order condition oc; main index MI; encrypted partition indexes $\{ePI\}$; symmetric encryption key $eKey$.
Output: Query result res.
Client:
1: Send k, oc to the enclave;
Enclave:
2: $minTree, maxTree \leftarrow MI$;
3: **if** oc is "$>$" **then**
4: $pnList \leftarrow maxTree(>, v)$;
5: $pnList_q \leftarrow minTree(>, v)$;
6: **else**
7: $pnList \leftarrow minTree(<, v)$;
8: $pnList_q \leftarrow maxTree(<, v)$;
9: **end if**
10: $res \leftarrow \phi$;
11: **for each** $pn \in pnList'$ **do**
12: Load ePI_{pn} using ORAM;
13: $PI_{pn} \leftarrow$Dec$(eKey, ePI_{pn})$;
14: $RS, BV, RDS, vList \leftarrow PI_{pn}$;
15: **if** $pn \in pnList_q$ **then** ▷ all qualified
16: res.append$(vList)$;
17: **else**
18: $iList \leftarrow$PartitionIndexRangeSearch(k, oc, PI_{pn});
19: res.append$(vList[iList])$;
20: **end if**
21: **end for**
22: Pad the result res;
23: return res to the client;

include the key list $kList$ in the partition index so that the key comparison can be realized. After the bucket is predicted (Line 3-4), the algorithm will find the border index i between $offset-1$ and $offset+size$ according to the order condition oc and the key k (Line 5-10). Finally, the list of qualified payloads will be truncated and returned.

E. Discussions

Padding of Query Results. The volume-size attack is a critical issue when supporting private queries whose returned result has a variable size. Attackers can infer the queried keyword by observing the volume of its query result. One straight solution is to pad all query results to a fixed size so that the extra volume information is eliminated. Despite being feasible, it will bring too much overhead especially for the queries with a small-size result. Hence, as other works do [11], [34], we employ a more efficient padding scheme, which pads the result up to the nearest multiple of a parameter r.

Queries within a Specified Block/Partition Range. One of our goals in this paper is to improve the query efficiency when it has to search the entire blockchain. However, it is still possible to be compatible with the query accompanied by a pre-specified partition/block range. A partition range limit can be easily satisfied by taking an intersection of the limited range and the partition list from the main index. The block range limit further requires to exclude the unqualified blocks within a partition. To achieve this, we can add the block height to the payload v in each index entry. Then it can be used to check whether the query result comes from the desired block range.

Algorithm 9 Range Search on Partition Index

Input: Query key k; order condition oc; partition index PI.
Output: Query result res.
Enclave:
1: $RS, BV, RDS, kList, vList \leftarrow PI$;
2: $s \leftarrow |kList|$;
3: $pos \leftarrow \lfloor RS.\text{Predict}(k) \rfloor$;
4: $offset, size \leftarrow BV.\text{at}(pos)$;
5: **if** oc is "$>$" **then**
6: Get the smallest $i \in [-1, size]$ that $kList[offset + i] > v$;
7: $res \leftarrow vList[offset + i, s)$;
8: **else**
9: Get the largest $i \in [-1, size]$ that $kList[offset+i] < v$;
10: $res \leftarrow vList[0, offset + i]$;
11: **end if**

Smart Contract-Related Queries. Apart from the traditional transfer transaction, which we use as an example to show the indexing, our system is also applicable to smart contract-related transactions. The internal transaction, which occurs within or between smart contracts, can be regarded as a transaction that has attributes different from the normal ones. Thus, our indexing methods can be conducted on these attributes as well. Furthermore, to query the historical state of a specific variable in a smart contract, we can collect the timestamp and its value each time the variable changes. In this way, the state value can be retrieved by performing a range query on the timestamp.

V. Implementation and Evaluation

We implement the learned index and SGX-based search using C++17 and Intel SGX SDK, and make the implementation of RSHash available on GitHub[1]. The ORAM is instantiated with a file-based Path ORAM [35] where every node has 4 data chunks and each chunk is sized at 640KB. The evaluation is performed on a cloud server enabled with Intel SGX, equipped with a duo-core 2.7GHz Intel Xeon Platinum 8374B CPU, 16GB of RAM, and running the 64-bit version of Ubuntu Server 20.04 LTS.

In our evaluation, we first employ four traditional datasets, including two real-world datasets and two synthetic datasets, for integer and string data types respectively. For integer keys, similar to previous works in the same field [36], [37], we select four 64-bit unsigned integer datasets: books (book popularity data on Amazon), fb (a portion of Facebook user IDs), normal (the normal distribution) and uniform (the uniform distribution). With respect to string keys, we leverage four sampled string datasets: dblp (bibliographic information in the

[1] https://github.com/tripleday/RSHash

TABLE I
STATISTICS OF DATASETS

No.	Dataset name	Key type	# of elements
1	books	Integer	31,985,432
2	fb	Integer	9,781,048
3	normal	Integer	10,000,000
4	uniform	Integer	10,000,000
5	dblp	String	371,192
6	english	String	207,227
7	AN	String	1,000,000
8	Hex	String	1,000,000
9	ETH value	Integer	2,619,334
10	ETH addr	String	308,641

field of computer science from dblp), english (English texts selected from collections of Gutenberg Project), AN (strings randomly generated from alphabets and numbers) and Hex (strings randomly generated from hexadecimal characters). AN can simulate random identifiers like account addresses, while Hex can mimic random hexadecimal outputs like hash values. We also add the real Ethereum transaction data of the first two million blocks to show its practicality. Specifically, in dataset ETH value, we collect the transaction values (in $1e12$ Wei) of each transaction. Meanwhile, we extract the sender addresses of all transactions to form the dataset ETH addr. The amount details of all datasets are listed in Table I.

A. Performance of Learned Index

We conduct a comparative evaluation of various search methods, including B+ tree [38], HashTable, Binary Search (BS), Adaptive Radix Trie (ART) [39], PGM [40], RadixSpline (RS) [32], LeMonHash [17] and our RSHash. Among them, ART is only used on string keys while PGM and RadixSpline are evaluated only on integer keys. The remaining methods can be applied to both key types. Since we want to evaluate the learned effect of all methods, except for HashTable which inserts keys one by one, other methods take a static and sorted array of keys as batch input. Given a queried key, they will return its index in the array. To focus on the generic performance, the evaluation is conducted in the normal memory on the server rather than the enclave.

Integer Keys. Figure 3 shows the evaluation result of integer keys in terms of construction time, query time and space cost. As shown in Figure 3a, HashTable exhibits the longest construction time on the books dataset, reaching approximately 346*ns* per key. In contrast, RSHash needs merely 115*ns* on the same dataset, which is notably about 40% lower than the 197*ns* initialization time required by the other MMPHF LeMonHash. Two traditional learned indexes, i.e., PGM and RadixSpline, take very small time for the construction because their indexing mainly aims to narrow down the search range of the queried key instead of getting the precise location. The construction of B+ tree for each key takes about 20*ns* on all datasets. Besides, RSHash also maintains a consistently moderate construction time across all datasets thanks to its flattened index building. As an optimal baseline, BS needs negligible construction time since it has no index building.

Figure 3b presents the average time of querying a single key. We can see that RSHash outperforms all other methods on all datasets. Taking the books dataset as an example, RSHash records a query time of 334*ns* which is significantly lower than B+ tree's 669*ns* and HashTable's 666*ns*. Both RadixSpline and RSHash have very small query delays over all datasets, which can be attributed to their effective underlying positioning approach. The learned index methods (the last four ones) all show superior performance to generic search methods (the first three ones) because the data distribution is fitted beforehand. Moreover, the performance of various methods shows a similar relative ranking across different datasets, indicating that the size and the distribution of the dataset have limited impacts.

Figure 3c depicts the average space cost of a key, where RSHash again excels, requiring a mere 3.2 bits per key on the books dataset, compared to B+ tree's 153 bits per key and HashTable's 356 bits per key. B+ tree and HashTable incur huge space overhead because they will additionally maintain complicated data structures like trees and buckets. LeMonHash costs 3.27 bits for one key, which is a little bit larger than RSHash but significantly smaller than other competitors. This is because these two designs no longer need to store the keys after the construction phase, while others require the stored keys for comparison purposes during the data query. In all cases, the PGM, RadixSpline and BS require a similar space cost of about 64 bits per key.

String Keys. Figure 4 illustrates the performance of various methods for string keys on different datasets. In Figure 4a, the construction time of each method is depicted. On the ETH addr dataset, LeMonHash demonstrates a relatively long construction time of around 672*ns* per key. The reason behind its long time is that it needs to maintain the complicated tree structures during the building. In contrast, B+ tree, ART and RSHash exhibit significantly lower time costs of 143*ns*, 161*ns* and 214*ns* respectively. HashTable takes a moderate time of 277*ns* while the simplest BS needs no time for index building. Our RSHash achieves a building time about 68% lower than that of LeMonHash, indicating that RSHash can build the buckets much more efficiently.

Figure 4b illustrates the average query time of each method, where RSHash continues to have superior performance. Still on the ETH addr dataset, B+ tree records the most query time of 1,177*ns* per key, which is more than 4 times our RSHash's 277*ns*. The traditional HashTable and ART need 483*ns* and 608*ns* respectively to query a key, which is a little smaller than the baseline BS's 703*ns*. Meanwhile, LeMonHash also gives a relatively shorter query time of 373*ns*. However, we can observe that LeMonHash always takes more query time than our RSHash, especially up to two times on the dataset Hex. This is mainly because RSHash uses a more efficient bucket positioning approach and a more compact alphabet reduction algorithm.

Figure 4c provides a comparison of how efficiently each method utilizes memory space. The space cost of all traditional methods remains largely consistent across the first four

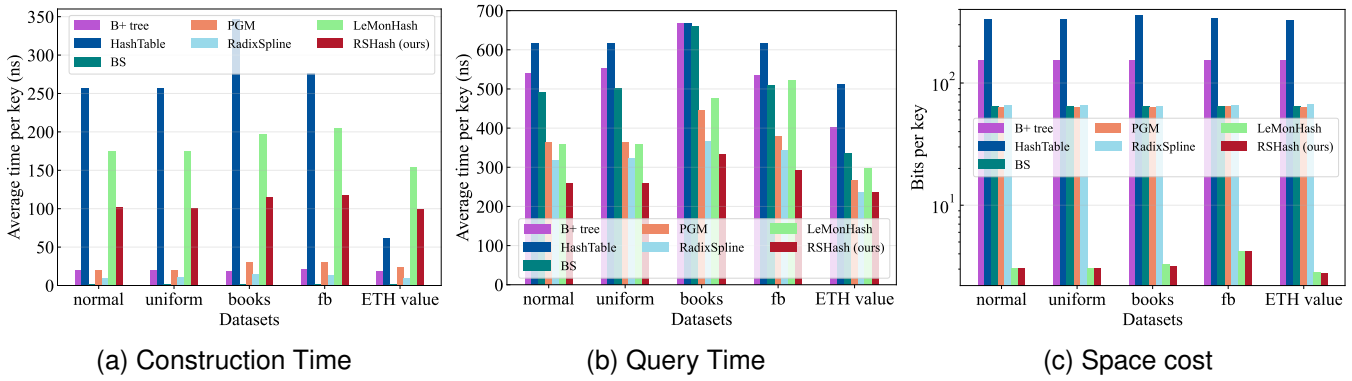


Fig. 3. Performance of Integer Keys on Different Datasets.

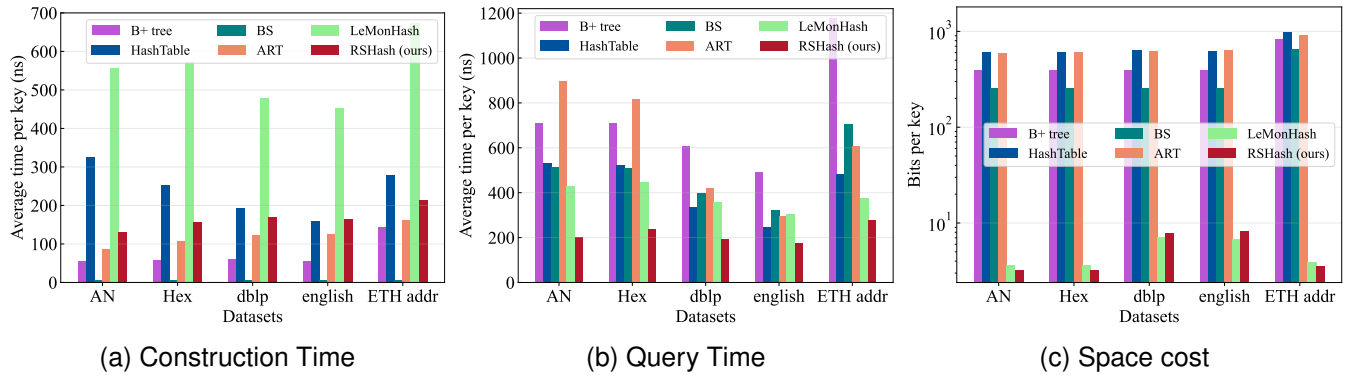


Fig. 4. Performance of String Keys on Different Datasets.

datasets. On the ETH addr dataset, the B+ tree, HashTable and ART require the most space, i.e., around 828 bits, 969 bits and 921 bits per key respectively, due to their extra data structures. The baseline BS approach needs a smaller space cost of 640 bits per key, which is all used to store the string array. LeMonHash and RSHash have the lowest space costs of 3.42 bits and 3.19 bits per key respectively, indicating that our RSHash is memory-efficient and advantageous for systems requiring small space.

The above evaluation illustrates that RSHash achieves minimal query latency and reduced construction time with a notably small space overhead. It is a highly efficient indexing method that is applicable to both integer and string keys across diverse datasets.

B. Performance of TELEX

We evaluate the performance of rich queries, including exact queries, range queries, Boolean queries and aggregate queries, using our proposed TELEX and other competitors. We use the real Ethereum datasets, including the ETH value for integers and ETH addr for strings, to populate the partitions according to the block numbers.

Exact Query. We evaluate the performance of TELEX against the approach, which we name as TB (traversal + BS), that prior work [11] would adopt when dealing with the search over the entire blockchain data. It traverses all partition data and uses the binary search way to get the result. We also include another two-level indexing method named HB (HashTable + BS), which employs a HashTable as the main index and uses the binary search method on the partition index. The reason why we adopt the BS manner on the partition index is that BS has the optimal space costs since it needs no index. Both integer keys and string keys are evaluated for the exact query. In the integer evaluation, TB and HB take 5,000 Ethereum blocks as a partition while TELEX increases it to 10,000. It is because the space saved by removing the keys in TELEX can accommodate more payloads. Similarly, in the string evaluation, the TB and HB regard 1,000 blocks as a partition and TELEX expands it to 5,000. The average query time is tested on the top 10 frequent keys.

Figure 5 shows the total construction time and average query time of these three methods on integers. In Figure 5a, we can see that the building time of all methods is almost linear to the amount of Ethereum blocks. When there are 1 million blocks, the HB, TB and TELEX take around 55.8s, 54.7s and 26.9s for construction, respectively. After the number of blocks reaches 2 million, their time rises to about 138.1s, 131.2s and 72s accordingly. Under all circumstances, TELEX costs over 45% less time than other methods since it has fewer partitions to process. HB always requires a little more time than TB due to the building of the main index.

On the other hand, as depicted in Figure 5b, TELEX achieves excellent performance in query time when compared with TB and HB. We can observe that the average time per

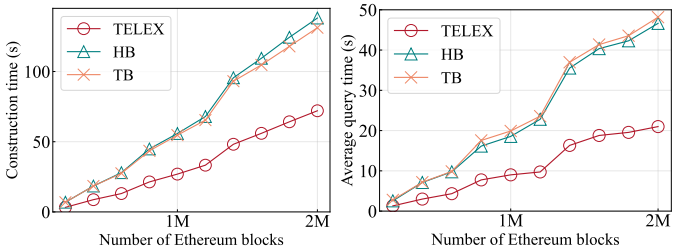


Fig. 5. Time Cost of Exact Query on Integers.

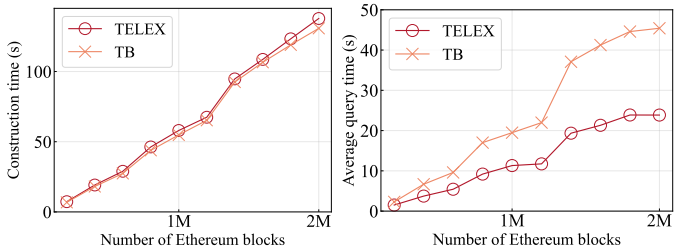


Fig. 7. Time Cost of Range Query.

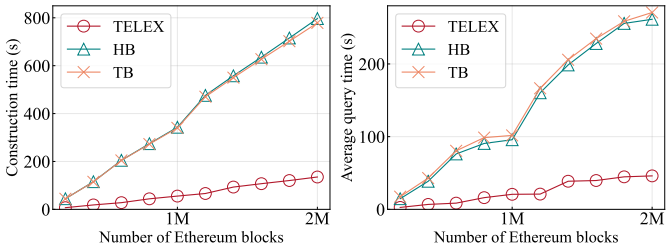


Fig. 6. Time Cost of Exact Query on Strings.

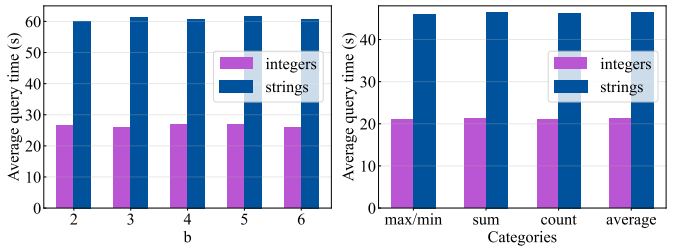


Fig. 8. Query Time of Other Queries.

query of TELEX remains consistently small, from 9s at 1 million blocks to 21s at 2 million blocks. On the contrary, TB requires about 19.9s and 48.2s when the block number is 1 million and 2 million respectively. Meanwhile, compared to TB, HB has a slightly shorter query time of 18.6s and 46.6s at the corresponding block number. We can conclude that the main index can help reduce some query time, but the reduction is limited when querying the most frequent keys.

Figure 6a presents the construction time of the methods on strings. It takes about 55.4s, 340.2s and 342.9s for TELEX, TB and HB respectively when there are 1 million blocks. The corresponding time increases to 135.1s, 778.7s and 796.8s when the block amount comes to 2 million. The significant reduction of RSHash's construction time is mainly due to the large space saved by removing long string keys. The saved space can help accommodate more payloads and decrease the number of partitions.

The query time on string keys is shown in Figure 6b. When there are 1 million blocks to search on, TELEX, HB and TB require about 20.7s, 95.7s and 101.7s on average for each query, respectively. After expanding to 2 million blocks, their time grows to 46s, 261.5s and 270.8s accordingly. The huge advantage in query time of RSHash is attributed to our efficient alphabet reduction algorithm, which avoids the time-consuming string comparison. As shown above, TELEX gains a great promotion in both construction time and query time thanks to the key removal in the partition indexes.

Range Query. We present the time costs of range queries on the ETH value in Figure 7. Only TB is used to compare with our TELEX since HB cannot handle the key comparison. Figure 7a illustrates that our TELEX requires slightly more time to build indexes compared to TB. With 2 million blocks, TB needs 130.8s to finish the construction while TELEX takes 137.8s. The additional time is primarily attributed to the main index building on all partitions. As we can see from Figure 7b, the main index in TELEX contributes to a shorter query time in return. Specifically, searching on 2 million blocks only costs TELEX 23.9s whereas TB needs 45.4s. The main index can help TELEX skip the partitions that do not contain the desired answers.

Boolean Query. We present the query time of Boolean queries on integers and strings using TELEX in Figure 8a. The number of keywords involved in the Boolean query expression is denoted as b and the queried keys are the top b frequent. We let b increase from 2 to 6 for two key types under the scenario of 2 million blocks. We can see that the query time of each type remains roughly close in all settings, i.e., 26s for integers and 61s for strings. This is because all queries contain the most frequent keywords, thus requiring most partitions to be loaded and searched, which reduces the filtering effectiveness of the main index.

Aggregate Query. We also evaluate the aggregate query, including max/min, sum, count, and average query, on given integer keys and string keys using the setting of 2 million blocks in Figure 8b. We can observe that all four types of queries also have very close query time on each key type, i.e., 21s for integers and 46s for strings. The reason is that the time cost of an aggregate query in our system is mainly determined by the loading of the encrypted partition indexes rather than negligible aggregate operations.

Non-private Query. We also evaluate the performance under the setting where the server is no longer interested in the client's query privacy but may still deviate from protocols. To this end, we develop the non-private version of our TELEX,

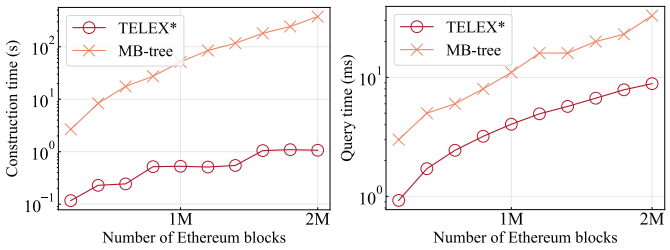


Fig. 9. Time Cost of Non-private Query on Integer Keys.

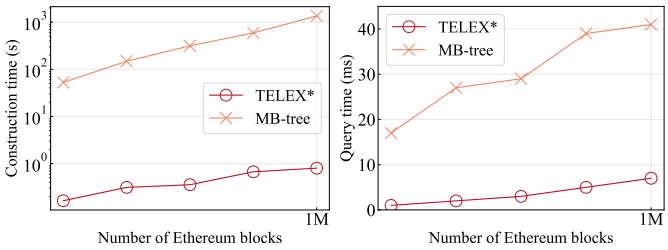


Fig. 10. Time Cost of Non-private Query on String Keys.

named TELEX*, which removes the ORAM component and related padding parts since it no longer needs to preserve privacy. However, it still reserves the index building and processing within the enclave to prevent the wrong results returning from the outside server. We contrast TELEX* with one of the state-of-the-art works focusing on the query authentication based on a MB-tree within the enclave [41], [42] (denoted as MB-tree).

Figure 9 shows the construction time and query time of the methods on integer keys. In Figure 9a, we can see that the construction time of MB-tree is always higher than TELEX*, e.g., 51.6s and 0.5s respectively at 1 million blocks and rise to 375.3s and 1.1s when block number comes to 2 million. This is mainly because MB-tree spends more time on the structure adjustment and hash calculation as the tree grows higher. The query time on integer keys is depicted in Figure 9b. Both methods' query time increases linearly as the number of blocks grows. When the block number reaches 2 million, TELEX* and MB-tree take 8.9ms and 33ms respectively to query the most frequent key. The reason behind MB-tree's higher query time is that it needs to generate the verification objects along the path for authentication purposes.

On the other hand, the performance of on string keys is presented in Figure 10. As shown in Figure 10a, similar to the integer case, MB-tree still takes much more time for construction than TELEX*. When there are 1 million blocks, it costs MB-tree and TELEX* 1352s and 0.8s respectively to finish the index building. This gap is larger than that in the integer case, indicating that the processing of strings is costlier than integers. Meanwhile, we can see from Figure 10b that the query time of the TELEX* is also much shorter than that of MB-tree. Taking 1 million blocks as an example, TELEX* needs only 7ms while MB-tree requires 41ms to query the most common string. Based on the observations above, we can conclude that our TELEX design is also a good choice for non-private query services.

VI. Related Work

A. Privacy-Preserving Queries over Plaintexts

The privacy protection when querying plaintexts is aimed to prevent service providers from learning sensitive information about user queries. There are three main approaches to realize the goal. The first category involves the use of anonymous communication protocols to obscure the identity of the query sender. For instance, the Tor network [43] is a well-known protocol that facilitates anonymity by routing the user's communication through a series of volunteering servers to conceal the user's identity and location. However, such protocols heavily depend on the altruism of intermediate nodes. In addition, the identity of the user is still necessary for subsequent processes in the blockchain applications, such as payment or data uploading. The second category of solutions addresses privacy by obfuscating the user's actual query with a series of dummy or fake queries [44]. This approach can be effective in preventing the identification of specific queries. However, it may introduce significant network overhead due to the transmission of redundant data.

The third category is a big family of Private Information Retrieval (PIR) protocols, which allow users to retrieve plaintexts from a server without leaking which information is being accessed. PIR can be implemented by a variety of techniques, including non-colluding replicated servers [45], function secret sharing [7], homomorphic encryption [8] and trusted hardware [15].The first approach proposes the family of classical PIR protocols based on mutually distrustful and non-colluding servers. The subsequent introduction of the function secret sharing significantly reduces the communication overhead of the protocols. However, the requirement of no collusion in these methods is inherently challenging in a decentralized blockchain scenario. While homomorphic encryption helps realize the computational PIR protocol on a single server, it is still impractical to apply to blockchain data due to heavy overhead. Trusted hardware, such as Intel SGX, offers a promising tool for PIR, by processing plaintexts and indexes within a secure environment. However, existing designs cannot be directly deployed to blockchain systems for privacy-preserving queries.

B. TEE for Blockchain Privacy

Trusted Execution Environment (TEE) has been widely utilized to enhance the privacy protection in blockchain systems. Ekiden [46] provides a platform for efficient confidentiality-preserving smart contracts with high scalability based on trusted enclaves. Yan *et al.* [47] investigate the problem of transaction confidentiality in financial blockchains and design a system named CONFIDE by leveraging TEE to address the privacy concern. Focusing on the private queries over blockchain data, two recent works explore to equip the

lightweight clients of Bitcoin [10] and Zcash [12] with TEE. They both employ Oblivious RAM (ORAM) techniques to randomize access patterns, thereby protecting the user's query privacy. However, how to efficiently support rich queries on blockchain data remains a topic that has not been resolved. To address this problem, a prior work [48] provides keyword query and Boolean query on Ethereum blockchain data in a private manner by employing ORAM as well. Their follow-up work [11] further supports queries on privacy coins and enforces fair service payments via smart contracts. In their designs, they require the issued query to be accompanied by a block range to specify the answer locations. But in our work, we weaken this requirement and alleviate the inefficiency when all blocks need to be traversed. In addition, we provide detailed index construction and query processing algorithms for other common yet complicated queries, e.g., range queries and aggregate queries. The on-chain data can be relaxed to any plaintext data in generic blockchain systems.

C. Learned Indexes

Learned indexes offer a fresh perspective on data indexing by leveraging machine learning algorithms to create a model that captures the inherent characteristics of data. Kraska *et al.* [49] initially introduce the concept of learned indexes and propose a multi-stage model named Recursive Model Index (RMI). It builds a tree of simple machine learning models to approximate the cumulative distribution function (CDF) of data. FITing-tree [50] explores the use of the linear model to fit the underlying data and constructs the index in a bottom-up fashion. Adopting a similar structure, the Piecewise Geometric Model index (PGM-index) [40] employs an error-bounded piecewise linear model for each level of the tree. It can support data updates with efficient time and space costs. PGM-index is also utilized to improve the performance of traditional hashing schemes, e.g., LeMonHash [17] as the monotone minimal perfect hash function. It uses the learned index to rapidly locate the hashing bucket and resorts to a retrieval data structure named BuRR [33] to get the rank within the bucket. Another notable learned index is the RadixSpline index [32], which combines a linear error-bounded spline with radix trees. Different from the tree structure in [40], [50], RS builds the index in a single pass to determine the spline points. The spline points are stored in a radix table and used to perform a linear interpolation for position estimation. It can achieve high performance in applications where data updates are trivial like blockchain systems.

VII. Conclusion

In this paper, we aim to tackle the problem of privacy and integrity during rich queries on the entire blockchain data. We first leverage the framework that combines TEE with ORAM to meet the objectives, and then design a two-level indexing method named TELEX to speed up the query on the entire blockchain data. Especially in the intra-partition index, we utilize the learned index to realize a novel hashing scheme. We also realize the query processing of a variety of rich queries, including exact queries, aggregate queries, Boolean queries, and range queries. Extensive evaluations validate the feasibility and superiority of our system. It provides a practical solution for private and reliable rich queries on generic blockchain data.

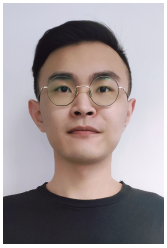
Acknowledgments

This work was supported in part by the Hong Kong Research Grants Council General Research Fund (No. 12202922, 15238724) and the Shenzhen Science and Technology Program (No. JCYJ20230807140412025).

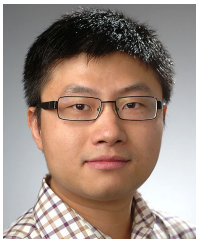
References

[1] L. Zhou, X. Xiong, *et al.*, "SoK: Decentralized Finance (DeFi) Attacks," in *IEEE SP*, pp. 2444–2461, 2023.
[2] B. D. Deebak and S. O. Hwang, "Healthcare applications using blockchain with a cloud-assisted decentralized privacy-preserving framework," *IEEE Trans. Mob. Comput.*, vol. 23, no. 5, pp. 5897–5916, 2024.
[3] K. Gai, Y. Zhang, M. Qiu, and B. Thuraisingham, "Blockchain-enabled service optimizations in supply chain digital twin," *IEEE Trans. Serv. Comput.*, vol. 16, no. 3, pp. 1673–1685, 2023.
[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[5] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
[6] "Emergency security warning multiple sites (reddit)." https://www.reddit.com/r/CryptoCurrency/comments/up2mna/emergency_security_warning_multiple_sites/.
[7] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing: Improvements and extensions," in *CCS*, pp. 1292–1303, ACM, 2016.
[8] H. Corrigan-Gibbs, A. Henzinger, and D. Kogan, "Single-server private information retrieval with sublinear amortized time," in *EUROCRYPT*, vol. 13276, pp. 3–33, 2022.
[9] H. Wang, C. Xu, C. Zhang, *et al.*, "vChain+: Optimizing verifiable blockchain boolean range queries," in *IEEE ICDE*, pp. 1927–1940, 2022.
[10] S. Matetic, K. Wüst, M. Schneider, K. Kostiainen, G. Karame, and S. Capkun, "BITE: bitcoin lightweight client privacy using trusted execution," in *USENIX Security Symposium*, pp. 783–800, 2019.
[11] C. Cai, L. Xu, A. Zhou, and C. Wang, "Toward a secure, rich, and fair query service for light clients on public blockchains," *IEEE Trans. Dependable Secur. Comput.*, vol. 19, no. 6, pp. 3640–3655, 2022.
[12] K. Wüst, S. Matetic, *et al.*, "ZLiTE: Lightweight clients for shielded zcash transactions using trusted execution," in *Financial Cryptography*, vol. 11598, pp. 179–198, 2019.
[13] Z. Peng, C. Xu, H. Wang, J. Huang, J. Xu, and X. Chu, "P^{2}B-Trace: Privacy-preserving blockchain-based contact tracing to combat pandemics," in *ACM SIGMOD*, pp. 2389–2393, 2021.
[14] A. Gervais, G. O. Karame, *et al.*, "On the security and performance of proof of work blockchains," in *ACM CCS*, pp. 3–16, 2016.
[15] H. Wu, Z. Peng, J. Xiao, *et al.*, "HeX: Encrypted rich queries with forward and backward privacy using trusted hardware," *IEEE Transactions on Dependable and Secure Computing*, 2025.
[16] S. Arnautov, B. Trach, F. Gregor, *et al.*, "SCONE: secure linux containers with intel SGX," in *USENIX OSDI*, pp. 689–703, 2016.
[17] P. Ferragina *et al.*, "Learned monotone minimal perfect hashing," in *Proc. of the Annual European Symposium on Algorithms*, 2023.
[18] D. Belazzougui *et al.*, "Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses," in *SIAM SODA*, pp. 785–794, 2009.
[19] S. Assadi *et al.*, "Tight bounds for monotone minimal perfect hashing," in *SODA*, pp. 456–476, SIAM, 2023.
[20] H. Lim, B. Fan, *et al.*, "SILT: a memory-efficient, high-performance key-value store," in *ACM SOSP*, pp. 1–13, 2011.
[21] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *CRYPTO*, vol. 6841, pp. 578–595, 2011.
[22] G. Navarro, "Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 52:1–52:47, 2013.
[23] D. Belazzougui, F. Cunial, *et al.*, "Linear-time string indexing and analysis in small space," *ACM Trans. Algorithms*, vol. 16, no. 2, pp. 17:1–17:54, 2020.
[24] H. Wu, Z. Peng, S. Guo, Y. Yang, and B. Xiao, "VQL: efficient and verifiable cloud query services for blockchain systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 6, pp. 1393–1406, 2022.

[25] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, p. 86, 2016.
[26] F. Piessens, "Verifying the security of enclaved execution against interrupt-based side-channel attacks," in *TIS@CCS*, p. 1, ACM, 2019.
[27] T. Yavuz, F. Fowze, *et al.*, "ENCIDER: detecting timing and cache side channels in SGX enclaves and cryptographic apis," *IEEE Trans. Dependable Secur. Comput.*, vol. 20, no. 2, pp. 1577–1595, 2023.
[28] M. van der Maas and S. W. Moore, "Protecting enclaves from intra-core side-channel attacks through physical isolation," in *ACM CCS*, 2020.
[29] D. Townley, K. Arikan, Y. D. Liu, D. Ponomarev, and O. Ergin, "Composable cachelets: Protecting enclaves from cache side-channel attacks," in *USENIX Security Symposium*, pp. 2839–2856, 2022.
[30] I. Eyal and E. G. Sirer, "Majority is not enough: bitcoin mining is vulnerable," *Commun. ACM*, vol. 61, no. 7, pp. 95–102, 2018.
[31] S. Gao, Z. Li, *et al.*, "Power adjusting and bribery racing: Novel mining attacks in the bitcoin system," in *ACM CCS*, pp. 833–850, 2019.
[32] A. Kipf, R. Marcus, *et al.*, "RadixSpline: a single-pass learned index," in *Proc. of ACM SIGMOD*, 2020.
[33] P. C. Dillinger, L. Hübschle-Schneider, P. Sanders, and S. Walzer, "Fast succinct retrieval and approximate membership using ribbon," in *SEA*, vol. 233 of *LIPIcs*, pp. 4:1–4:20, 2022.
[34] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *CCS*, pp. 668–679, ACM, 2015.
[35] E. Stefanov, M. van Dijk, E. Shi, T. H. Chan, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," *J. ACM*, vol. 65, no. 4, pp. 18:1–18:26, 2018.
[36] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "SOSD: A benchmark for learned indexes," *NeurIPS Workshop*, 2019.
[37] R. Marcus, A. Kipf, *et al.*, "Benchmarking learned indexes," *Proc. VLDB Endow.*, vol. 14, no. 1, pp. 1–13, 2020.
[38] "STX B+ Tree." https://panthema.net/2007/stx-btree/.
[39] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *IEEE ICDE*, pp. 38–49, 2013.
[40] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1162–1175, 2020.
[41] Q. Shao, S. Pang, Z. Zhang, and C. Jin, "Authenticated range query using SGX for blockchain light clients," in *DASFAA (3)*, vol. 12114, pp. 306–321, 2020.
[42] Q. Shao, Z. Zhang, C. Jin, and A. Zhou, "Query authentication using intel SGX for blockchain light clients," *J. Comput. Sci. Technol.*, vol. 38, no. 3, pp. 714–734, 2023.
[43] R. Dingledine, N. Mathewson, and P. F. Syverson, "Tor: The second-generation onion router," in *USENIX Security Symposium*, 2004.
[44] R. Masood, D. Vatsalan, M. Ikram, and M. A. Kâafar, "Incognito: A method for obfuscating web data," in *WWW*, pp. 267–276, ACM, 2018.
[45] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, no. 6, pp. 965–981, 1998.
[46] R. Cheng, F. Zhang, J. Kos, *et al.*, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *EuroS&P*, pp. 185–200, IEEE, 2019.
[47] Y. Yan, C. Wei, X. Guo, *et al.*, "Confidentiality support over financial grade consortium blockchain," in *ACM SIGMOD*, pp. 2227–2240, 2020.
[48] C. Cai, L. Xu, A. Zhou, R. Wang, C. Wang, and Q. Wang, "EncELC: Hardening and enriching ethereum light clients with trusted enclaves," in *INFOCOM*, pp. 1887–1896, IEEE, 2020.
[49] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *ACM SIGMOD*, pp. 489–504, 2018.
[50] A. Galakatos, M. Markovitch, *et al.*, "FITing-tree: A data-aware index structure," in *ACM SIGMOD*, pp. 1189–1206, 2019.



Haotian Wu is a postdoctoral fellow in the Department of Industrial and Systems Engineering, The Hong Kong Polytechnic University. He received his B.Sc. and M.Sc. degrees from Southeast University, and his Ph.D. degree from The Hong Kong Polytechnic University. His research interests include data security, applied cryptography, and blockchain systems.



Yuzhe Tang is an associate professor at Syracuse University. His research interests are vulnerability discovery, attack detection, defense design, and large-scale measurement, performance and optimization. Full-stack systems security in blockchain is the main theme of his current research. His works on this topic have been published on USENIX Security, CCS, NDSS, FSE, WWW, IMC, ICDE, ACSAC, Euro S&P, etc. His research also creates real-world impacts, and the code patch his team develops has been adopted in the mainstream open-source projects. He received the Best Paper award in IEEE Cloud 2012 and in ACM/IEEE CCGrid 2015, and the Ethereum Foundation academic grant awards. He is an invited speaker at SBC 2023. He obtains his Ph.D. at Georgia Tech.

Zhaoyan Shen received the BE and ME degrees in the Department of Computer Science and Technology, Shandong University, China, in 2012 and 2015, respectively, and the Ph.D. degree in the Department of Computing, Hong Kong Polytechnic University, in 2018. Currently, he is a professor at Shandong University. His research interests include big data systems, storage systems, embedded systems, system architecture, and hardware/software co-design.

Jun Tao received the B.S. and M.S. degrees in computer science from the Department of Computer Science and Engineering, Northeast University, in 1998 and 2001, respectively, and the Ph.D. degree in computer science from the Department of Computer Science and Engineering, Southeast University, in 2005. He is currently a full professor and a supervisor of Ph.D. candidates in the School of Cyber Science and Engineering at Southeast University. His research interests include social networking, blockchain systems, and information security.

Chenhao Lin received the B.E. degree in automation from Xi'an Jiaotong University in 2011, the M.Sc. degree in electrical engineering from Columbia University, in 2013 and the Ph.D. degree from The Hong Kong Polytechnic University, in 2018. He is currently a Research Fellow at the Xi'an Jiaotong University of China. His research interests are in artificial intelligence security, adversarial attack and robustness, identity authentication, and pattern recognition.

Zhe Peng is currently a research assistant professor in the Department of Industrial and Systems Engineering, The Hong Kong Polytechnic University. He received the B.S. degree from Northwestern Polytechnical University, the M.S. degree from University of Science and Technology of China, and the Ph.D. degree from The Hong Kong Polytechnic University. He was a visiting scholar in the Department of Electrical and Computer Engineering, Stony Brook University. His research interests include blockchain, distributed computing, internet of things, big data analysis and management, and data security.