Slicer: Verifiable, Secure and Fair Search over Encrypted Numerical Data Using Blockchain

Haotian Wu*, Rui Song*, Kai Lei†, Bin Xiao*
*The Hong Kong Polytechnic University, China, *{cshtwu, csrsong, csbxiao}@comp.polyu.edu.hk
†Peking University, Shenzhen, China, †leik@pkusz.edu.cn

2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-7177-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDCS54860.2022.00118

***Abstract*—Verifiable Searchable Symmetric Encryption (SSE) enables reliable search over encrypted, privacy-preserving 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 paper, 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. Our search results are verifiable, updated and privacy-preserving by SSE and maintaining the forward security. We illustrate the security and practicality of our design through rigorous analysis and extensive evaluations respectively.**

***Index Terms*—Verifiable search, searchable symmetric encryption, privacy-preserving, blockchain, cloud data**

I. Introduction

Outsourcing data to clouds has become a strong trend for data owners to relieve from great storage cost and heavy online burden. 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 [1], [2] 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., [3], [4], [5], [6], [7], [8]). Nevertheless, there still remain the following three limitations that have not been well addressed.

First, most verifiable SSE schemes [4], [5], [7], [8], [9], [10] 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 [3], [6], [7], [8], [9], [11] let data users locally verify the search results based on the assumption that data users will honestly report the verification outcome. 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 [12] enables verifiable range queries over encrypted data, but its verification requires the decryption of data, which violates the first rule. Several designs [13], [14], [15] 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* [16], 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 paper, we are the first to investigate *the public verification problem for encrypted numerical search with dynamic data*. 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 the public verification algorithms 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, Slicer, to provide verifiable encrypted search over numerical data

2575-8411/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDCS54860.2022.00118

by 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. Based on that, we design a novel verifiable and secure SSE scheme, 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 paper is organized as follows. We first review related literatures in Section II and give the preliminaries in Section III. We then introduce the system model in Section IV and describe our design in Section V. We further analyze the design in Section VI and present the evaluation in Section VII. Section VIII finally concludes our paper.

II. Related Work

A. 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 [3] 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 [11], Stefanov *et al.* achieve the verifiability of dynamic SSE by comparing the message authenticated code and further support forward security. Bost *et al.* [6] improve Stefanov's design and present generic solutions for verifiable SSE. ServeDB [12] designs a tree-based index with cube encoding to support verifiable range queries over dynamic encrypted data. Ge *et al.* [9] propose a verifiable SSE that supports efficient data dynamic update by using a novel accumulative authentication tag. GSSE [7] 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.* [8] propose a verifiable searchable symmetric encryption scheme that supports 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 [10] propose two publicly verifiable SSE constructions upon basic cryptographic primitives. Based on the public key encryption, Zheng *et al.* [4] propose the first verifiable attribute-based keyword search (VABKS) scheme over static data. Sun *et al.* [5] 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) [17], [18]. 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, blockchain technology has been used to devise verifiable searchable encryption schemes [13], [19], [14], [15]. In [13], Hu *et al.* directly store the whole encrypted indexes on the blockchain and execute the search through the smart contract. 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.* [19] offload the storage of encrypted files and indexes to the decentralized storage systems. Their design only supports keyword search over append-only encrypted data due to the immutability of blockchain. In [14], Guo *et al.* design a verifiable and forward-secure SSE scheme by virtue of the blockchain. Li *et al.* [15] 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, providing efficient and verifiable query services [20].

We list state-of-the-art related studies on verifiable SSE in Table I. 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.

B. Numerical Comparison over Encrypted Data

Order Preserving Encryption (OPE) [21] enables the numerical comparison by directly encrypting the plaintexts, making the ciphertexts preserve the numerical order of plaintext-space. CryptDB [22] utilizes OPE to support functionally rich queries over encrypted databases. OPE cannot 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.* [23] propose the first efficient order-revealing encryption (ORE) scheme, which allows the public comparison between ciphertexts. It reveals the location of the first bit where two ciphertexts differ. Lewi and Wu [24] 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 [25], Demertzis *et al.* present a range SSE scheme by employing a novel tree-like directed acyclic graph. Guo *et al.* [26] 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.

III. Preliminaries

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

TABLE I: Comparison with State-of-the-Art Verifiable Searchable Encryption Schemes

Designs[a]		**Dynamics**[b]	**Numerical comparison**	**Freshness**[c]	**Forward security**[d]	**Public verifiability**[e]
Traditional designs	[3]	×	×	N/A	N/A	×
	[11], [6]	√	×	N/A	√	×
	[12]	√	√	×	×	×
	[9]	√	×	×	×	×
	[7]	√	×	√	×	×
	[8]	√	×	×	×	×
	[10]	×	×	N/A	N/A	√
	[4]	×	×	N/A	N/A	×
	[5]	√	×	×	×	√
Blockchain-based designs	[13], [14], [15]	√	×	√	√	√
	[19]	×	×	√	√	√
	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.

A. Blockchain and Smart Contract

A blockchain is a distributed ledger that offers reliable storage for transaction information in a decentralized network. The blockchain data is transparent and immutable due to the underlying hash chain technique and consensus protocols. It can also provide a trusted environment for program execution via the smart contract, e.g., Ethereum. However, due to the limited storage and computation resources, the smart contract charges fees for the execution, rendering itself not suitable for massive data storage and complicated programs.

B. Cryptographic Primitives

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 secret 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) $F : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$, if for all probabilistic polynomial-time (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 $\mathcal{A}$, $\Pr\left[y \xleftarrow{\$} \mathcal{M}, x \leftarrow \mathcal{A}\left(1^\lambda, pk, y\right) : \pi_{pk}(x) = y\right] \leq negl(\lambda)$ 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 work, we employ the $\mathsf{MSet\text{-}Mu\text{-}Hash}$ construction in [27]. 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.

RSA Accumulator. The RSA accumulator is a collision resistant ADS based on strong RSA assumption that can provide authentication for sets. Compared with Merkle Hash Tree, which is another ADS that can provide existence proofs, the proof in the RSA accumulator is constant-size and leaks no extraneous information. The bilinear-map accumulator works like the RSA accumulator, but requires much more storage for the public key. The functions of the RSA accumulator that we use in this work are as follows [28]:

- $\mathsf{Setup}(1^\lambda)$ 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 $g \in QR_n \backslash \{1\}$, where QR_n is the group of quadratic residues modulo n.
- $\mathsf{Accumulation}(X)$ 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., $x_p = \prod_{x \in X} x$.

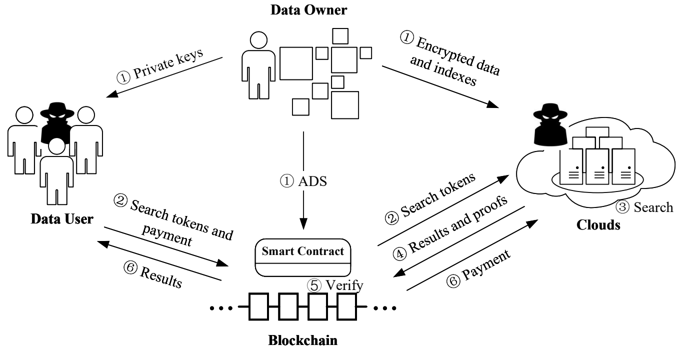


Fig. 1: Verifiable encrypted search using blockchain.

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

To find prime representatives for input elements, we employ the method proposed in [29] and denote it as $H_{prime}(\cdot)$. It can be seen as a random oracle that outputs random prime numbers.

IV. System Model

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

A. Framework Architecture

As shown in Fig. 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.

B. 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.

V. Slicer Design

A. 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 $mc = \{\text{“}=\text{”}, \text{“}>\text{”}, \text{“}<\text{”}\}$.

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 [27] 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.

B. 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 [24], 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:

- $\mathsf{SORE.Token}(k, v, oc)$: Given the queried value v and the order condition $oc \in \{$“>”, “<”$\}$, 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 $tk = \{F_k(tk_1), F_k(tk_2), \cdots, F_k(tk_b)\}$.
- $\mathsf{SORE.Encrypt}(k, v)$: 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 $cmp(\bar{v_i}, v_i) \in \{$“>”, “<”$\}$ 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 $ct = \{F_k(ct_1), F_k(ct_2), \cdots, F_k(ct_b)\}$.
- $\mathsf{SORE.Compare}(ct, tk)$: 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`.

We give an illustrative example of our SORE scheme in Fig. 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.

C. Building Encrypted Indexes and ADS

Algorithm 1 presents the building process of the encrypted indexes and ADS. Following $\mathsf{SORE.Encrypt}$, 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 [16] 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.

Algorithm 1: Build: Indexes and ADS Building

```
Input  : PRF key $K$; encryption key $K_R$; key-value
         database $\mathsf{DB}$; secure PRFs $\{F, G\}$; multiset
         hash function $\mathcal{H}$; random oracle function
         $H_{prime}$.
Output: Encrypted index $I$; ADS information $(X, Ac)$.
1  foreach $(R, v) \in \mathsf{DB}$ do
2      for $i = 1$ to $b$ do
3          $ct_i \leftarrow v_{|i-1} \| \bar{v_i} \| cmp(\bar{v_i}, v_i)$;
4          Put $(R, ct_i)$ into $\mathsf{DB}$;
5  Initialize a dictionary $I$ for indexes, $T$ for trapdoor
   states and $S$ for set hashes;
6  foreach $w \in \{v\} \cup \{ct_i\}$ do
7      Randomly generate a trapdoor $t_0$;
8      $T.put(w, (t_0, 0))$;
9      $G_1 \leftarrow G(K, w\|1)$; $G_2 \leftarrow G(K, w\|2)$; $c \leftarrow 0$;
10     $h \leftarrow \mathcal{H}(\phi)$;
11     foreach $R \in \mathsf{DB}(w)$ do
12         $l \leftarrow F(G_1, t_0\|c)$;
13         $d \leftarrow F(G_2, t_0\|c) \oplus Enc(K_R, R)$;
14         $I.put(l, d)$; $c++$;
15         $h \leftarrow h +_{\mathcal{H}} \mathcal{H}(Enc(K_R, R))$;
16     $S.put(t_0\|0\|G_1\|G_2, h)$;
17 Initialize a list $X$ for primes;
18 foreach $(g, h) \in S$ do
19     $x \leftarrow H_{prime}(g\|h)$; $X.add(x)$;
20 $Ac \leftarrow \mathsf{Accumulation}(X)$;
21 Send $(I, X, Ac)$ to the cloud;
22 Send $Ac$ to the blockchain;
23 Send $(K, K_R, T)$ to the data user;
```

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.

D. Data Insertion

As shown in Algorithm 2, 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.

E. Verifiable Search Protocol

Algorithm 3 describes the search token generation executed by the data owner in the Search protocol. Following

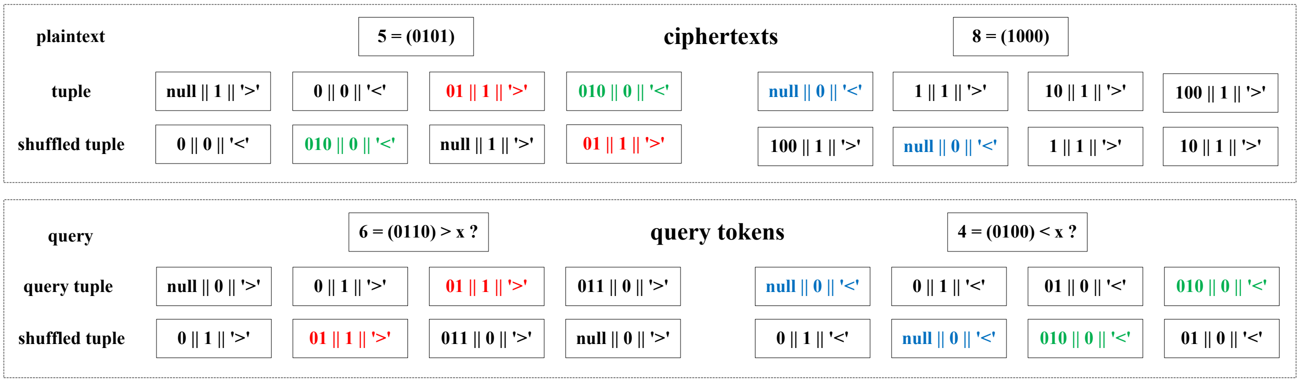


Fig. 2: An illustrative example of SORE.

Algorithm 2: Insert: Forward-Secure Insertion

```
Input  : PRF key $K$; encryption key $K_R$; trapdoor secret key $sk$; key-value pairs to insert $\mathsf{DB}^+$;
         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 $\mathsf{DB}^+$;
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
        $l \leftarrow F(G_1, t\|c)$;
        $d \leftarrow F(G_2, t\|c) \oplus Enc(K_R, R)$;
        $I.put(l, d)$; $c++$;
        $h \leftarrow h +_{\mathcal{H}} \mathcal{H}(Enc(k_R, R))$;
    $S.put(t\|j\|G_1\|G_2, h)$;
    $x^+ \leftarrow H_{prime}(t\|j\|G_1\|G_2\|h)$; $X^+.add(x^+)$;
$X \leftarrow X \cup X^+$;
$Ac \leftarrow \mathsf{Accumulation}(X)$;
Send $(I, X, Ac)$ to the cloud;
Send $Ac$ to the blockchain;
Send $T$ to the data user;
```

Algorithm 3: 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;
```

SORE.Token, the data owner first produce 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 4. 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 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 as mentioned in Section III, and then sent to the blockchain with

Algorithm 4: 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;
```

the results.

Algorithm 5: 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 $vr$.
Blockchain.Verify
    $vr \leftarrow$ True;
    foreach $(t_j, j, G_1, G_2, er, vo)$ do
        $h \leftarrow \mathcal{H}(er)$; $x \leftarrow (H_{prime}(t_j \| j \| G_1 \| G_2 \| h)$;
        $vr \leftarrow$ VerifyMem $(x, vo)$;
        if $vr$=False then
            Refund the payment;
    Proceed the payment;
    The data user decrypts all $er$ using $Dec(K_R, er)$;
```

We present the result verification by the blockchain in Algorithm 5. 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.

F. 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 [16]. In another word, we can use one instance for all inserted data while the other one stores all deleted data. In this way, the final search result becomes the difference of 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.

VI. Design Analysis

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

A. Correctness and Security on SORE scheme

Our SORE scheme is inspired by the ORE schemes in [23], [24], [26]. We devise the 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 1. *Given the PRF key k, two values x, y, and the order condition* $\mathsf{oc} \in \{$“$>$”, “$<$”$\}$*, write $tk \leftarrow \{F_k(tk_1), F_k(tk_2), \cdots, F_k(tk_b)\}$ generated by* SORE.Token*(k, x) and $ct \leftarrow \{F_k(ct_1), F_k(ct_2), \cdots, F_k(ct_b)\}$ generated by* SORE.Encrypt*(k, y). x* oc *y stands if and only if* SORE.Compare(ct, tk) =True.

Proof. Because secure PRF is applied to both sides, 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 SORE.Compare$(ct, tk)$ outputs True since the tuple at index m is the desired common one.
- Suppose that SORE.Compare(ct, tk) =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 $\mathsf{SORE.Token}$, we can find out the leakage between any two values by counting how many common tuples exist. The leakage among the ciphertexts that produced by $\mathsf{SORE.Encrypt}$ 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 $\mathsf{SORE.Compare}$ 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.

B. Security on Encrypted Search

In this subsection, following the security notion of SSE [1], [2], [30], 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:

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

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 1. *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.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$*: $\mathcal{A}$ 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 2. *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 [16] 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.

□

C. Correctness of Verifiable Search

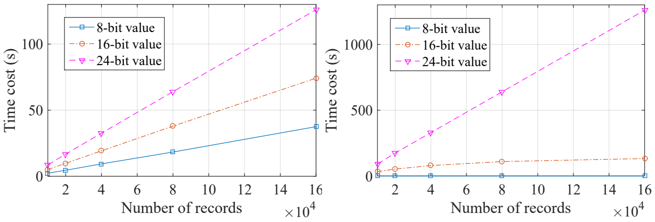
We prove the correctness of verification in terms of soundness and completeness.

Definition 2. *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: $\{R|R \notin Q(\mathsf{DB}) \wedge R \in \{rs\}_n\} \neq \phi \vee \{R|R \in Q(\mathsf{DB}) \wedge R \notin \{rs\}_n\} \neq \phi$.*

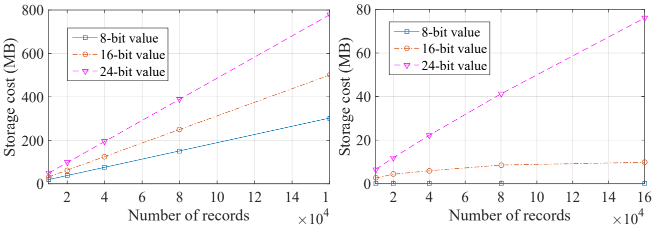
Theorem 3. *Ω is correct if the multiset hash function and prime representation function are collision resistant, and the underlying RSA accumulator is secure.*

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 [28], 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. □



(a) Indexes (b) ADS

Fig. 3: Time cost of Build.



(a) Indexes (b) ADS on cloud

Fig. 4: Storage cost of Build.

VII. Implementations and Evaluation

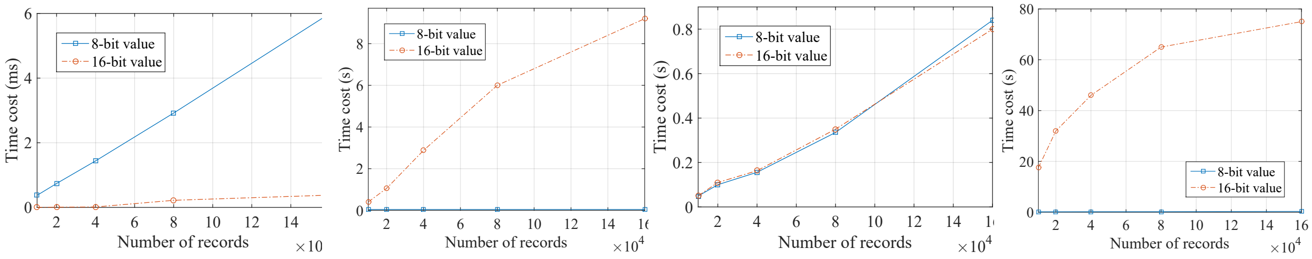
To demonstrate the practical efficiency of our design, we implement a prototype[1], including the data owner, data user and clouds in Python 3.8.0 and the blockchain in Solidity. We perform the evaluation on a machine with i9-9900K CPU, 32 GB memory and 1 TB SSD. For cryptographic primitives, we employ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

A. Building Performance

Fig. 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 Fig. 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 Fig. 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.

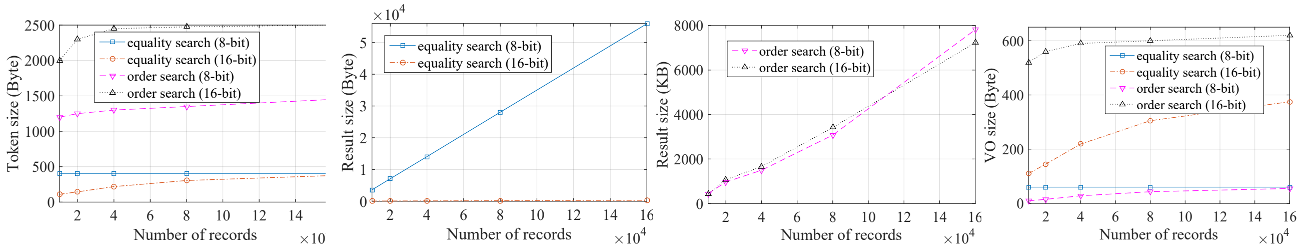
We show the storage cost of the indexes and the ADS during the building phase in Fig. 4a and Fig. 4b respectively. The

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



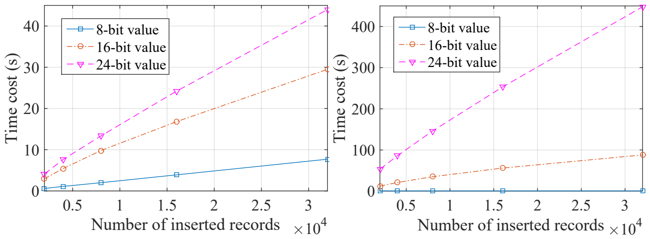
(a) Result generation for equality (b) VO generation for equality (c) Result generation for order (d) VO generation for order

Fig. 5: Time cost of Search.



(a) Search token (b) Equality result (c) Order result (d) Verification object

Fig. 6: Overhead generated by Search.



(a) Indexes (b) ADS

Fig. 7: Time cost of Insert.

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.

B. Search Performance

To evaluate the performance of search, we select random numbers to execute the protocol and average the outcomes. Fig. 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 Fig. 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 Fig. 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 Fig. 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.

Fig. 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 Fig. 6b and Fig. 6c, the size of encrypted results under all settings is proportional to the amount of records. As for the VO in Fig. 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.

C. Insertion Time

We pre-load 160K amount of records and assess the insertion efficiency in terms of indexes and ADS. In Fig. 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.

D. Gas Consumption

We list the gas cost of the smart contract conducted on Rinkeby testnet in Table II. The data insertion in our design is very cheap in gas since it only needs to change a 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

TABLE II: Gas cost of smart contract

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

the price of 3000$. The gas appears practically low because the verification of the ADS can be be finish in $O(\lambda)$.

VIII. Conclusion

Many traditional verifiable SSE schemes are limited to keyword-file search and leave the range search blank. In this paper, we conduct pioneering investigation on the public verification problem atop blockchain for encrypted search over numerical data. Our proposed solution, Slicer, realizes verifiable and secure range search by adopting a novel SSE scheme and guarantees the fairness via public verification. We utilize the blockchain technology to achieve fresh search result. Given data updates, our solution is also privacy-preserving due to the forward security function implementation. We prove its security via formal analysis and show the efficiency through extensive experiments.

Acknowledgment

This work was partially supported by the Key-Area Research Development Program of Guangdong Provice 2020B0101090003 and HK RGC GRF PolyU 15216220.

References

[1] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

[2] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 965–976, 2012.

[3] Q. Chai and G. Gong, "Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers," in *2012 IEEE International Conference on Communications (ICC)*, pp. 917–922, IEEE, 2012.

[4] Q. Zheng, S. Xu, and G. Ateniese, "VABKS: Verifiable attribute-based keyword search over outsourced encrypted data," in *IEEE INFOCOM 2014-IEEE conference on computer communications*, pp. 522–530, IEEE, 2014.

[5] W. Sun, X. Liu, W. Lou, Y. T. Hou, and H. 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)*, pp. 2110–2118, IEEE, 2015.

[6] R. Bost, P.-A. Fouque, and D. Pointcheval, "Verifiable dynamic symmetric searchable encryption: Optimality and forward security.," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 62, 2016.

[7] J. Zhu, Q. Li, C. Wang, X. Yuan, Q. Wang, and K. Ren, "Enabling generic, verifiable, and secure data search in cloud services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 8, pp. 1721–1735, 2018.

[8] X. Liu, G. Yang, Y. Mu, and R. H. Deng, "Multi-user verifiable searchable symmetric encryption for cloud storage," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 6, pp. 1322–1332, 2018.

[9] X. Ge, J. Yu, H. Zhang, C. Hu, Z. Li, Z. Qin, and R. Hao, "Towards achieving keyword search over dynamic encrypted cloud data with symmetric-key based verification," *IEEE Transactions on Dependable and Secure computing*, 2019.

[10] A. Soleimanian and S. Khazaei, "Publicly verifiable searchable symmetric encryption based on efficient cryptographic components," *Designs, Codes and Cryptography*, vol. 87, no. 1, pp. 123–147, 2019.

[11] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage.," in *NDSS*, vol. 71, pp. 72–75, 2014.

[12] S. Wu, Q. Li, G. Li, D. Yuan, X. Yuan, and C. Wang, "ServeDB: Secure, verifiable, and efficient range queries on outsourced database," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 626–637, IEEE, 2019.

[13] S. Hu, C. Cai, Q. Wang, C. Wang, X. Luo, and K. Ren, "Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pp. 792–800, IEEE, 2018.

[14] Y. Guo, C. Zhang, and X. Jia, "Verifiable and forward-secure encrypted search using blockchain techniques," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pp. 1–7, IEEE, 2020.

[15] H. Li, H. Zhou, H. Huang, and X. Jia, "Verifiable encrypted search with forward secure updates for blockchain-based system," in *International Conference on Wireless Algorithms, Systems, and Applications*, pp. 206–217, Springer, 2020.

[16] R. Bost, "Σοφος: Forward secure searchable encryption," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1143–1154, 2016.

[17] K. Ren, Y. Guo, J. Li, X. Jia, C. Wang, Y. Zhou, S. Wang, N. Cao, and F. 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)*, pp. 23–33, IEEE, 2020.

[18] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using SGX," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 264–278, IEEE, 2018.

[19] C. Cai, J. Weng, X. Yuan, and C. Wang, "Enabling reliable keyword search in encrypted decentralized storage with fairness," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[20] H. Wu, Z. Peng, S. Guo, Y. Yang, and B. Xiao, "VQL: Efficient and verifiable cloud query services for blockchain systems," *IEEE Transactions on Parallel Distributed Systems*, vol. 33, pp. 1393–1406, Jun. 2022.

[21] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill, "Order-preserving symmetric encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 224–241, Springer, 2009.

[22] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 85–100, 2011.

[23] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu, "Practical order-revealing encryption with limited leakage," in *International conference on fast software encryption*, pp. 474–493, Springer, 2016.

[24] K. Lewi and D. J. Wu, "Order-revealing encryption: New constructions, applications, and lower bounds," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1167–1178, 2016.

[25] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical private range search revisited," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 185–198, 2016.

[26] X. Yuan, X. Wang, C. Wang, B. Li, X. Jia, *et al.*, "Enabling encrypted rich queries in distributed key-value stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 6, pp. 1283–1297, 2018.

[27] D. Clarke, S. Devadas, M. Van Dijk, B. Gassend, and G. E. 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*, pp. 188–207, Springer, 2003.

[28] J. Li, N. Li, and R. Xue, "Universal accumulators with efficient nonmembership proofs," in *International Conference on Applied Cryptography and Network Security*, pp. 253–269, Springer, 2007.

[29] N. Barić and B. Pfitzmann, "Collision-free accumulators and fail-stop signature schemes without trees," in *International conference on the theory and applications of cryptographic techniques*, pp. 480–494, Springer, 1997.

[30] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: data structures and implementation.," in *NDSS*, vol. 14, pp. 23–26, Citeseer, 2014.