Searching for Encrypted Data on Blockchain: An Efficient, Secure and Fair Realization

Jianzhang Chen[1,2], Haibo Tian[1,2], and Fangguo Zhang[1,2(✉)]

[1] School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, China
isszhfg@mail.sysu.edu.cn

[2] Guangdong Province Key Laboratory of Information Security Technology, Guangzhou 510006, China

Abstract. Searchable symmetric encryption (SSE) is a research hotspot in applied cryptography, with the purpose of protecting outsourced data while enabling querying of encrypted data. However, the majority of current research focuses on the scenario in which data is stored on a single server and disregards the possibility that both the clients and servers are malicious. While several existing blockchain-based SSE schemes provide solutions to the issues above, they do not simultaneously achieve security, fairness, and decentralized storage.

In this paper, we explore how to efficiently solve the above problems in the blockchain setting. We build up a decentralized fair SSE framework in a layered fashion. First, we present a practical and efficient method for accessing data on the blockchain. Based on this, we craft a decentralized publicly verifiable SSE scheme in which encrypted indexes are stored on the blockchain and search operations are shifted to be executed off-chain for lightweight decentralized storage and efficient query performance. Then, we use smart contracts to confer fairness to SSE by constructing a game model that makes each party prefer to cooperate. Finally, we implement and evaluate our framework on Ethereum. The experimental results demonstrate that our design is effective and practical.

Keywords: Searchable symmetric encryption · Blockchain · Fairness

1 Introduction

Symmetric searchable encryption (SSE), a cryptographic primitive aimed at enabling the search function of encrypted data while guaranteeing data confidentiality, has received considerable attention. It was initially proposed by Song *et al.* [16]. Since Curtmola *et al.* [7] developed a better definition of the functionality and security of SSE, numerous feature-rich schemes have emerged in recent years, including dynamic SSE [5] and verifiable SSE [3].

However, the majority of SSE schemes store data on a single server, making the single point of failure one of the obstacles to the deployment of SSE schemes.

W. Susilo et al. (Eds.): ISC 2022, LNCS 13640, pp. 194–213, 2022.
https://doi.org/10.1007/978-3-031-22390-7_13

Even though most cloud providers offer redundant backup services, it remains an unresolved question how to fully utilize these backup servers for search. To compound the issue, when users find that the data stored on the server has been altered or deleted, it is difficult to migrate the data to other cloud providers without ensuring the completeness and accuracy of their data.

Beyond that, most SSE schemes are based on the assumption that data users are trustworthy and servers are honest but curious. However, the usability and security of SSE will be significantly weakened if both data users and servers are malicious. Even though verifiable SSE schemes are reasonable solutions to the problem that a malicious server returns incorrect results, it is impossible to prevent a malicious data user from claiming that the server returned incorrect results to avoid paying the remuneration, even if the server performs the search honestly.

Blockchain has emerged in the last decade and has brought the possibility of decentralization and fairness to SSE. Originating from Bitcoin, blockchain is a cryptographic technology that maintains a reliable and tamper-evident database through decentralization. In recent years, some works have been utilizing blockchain to ensure fairness for SSE. Li *et al.* [14,15] first proposed a blockchain-based searchable symmetric encryption scheme whose construction is based on a blockchain transaction paradigm. Zhang *et al.* [22] proposed a fair SSE scheme called TKSE based on the same transaction paradigm and claimed to achieve two-party verifiability and better compatibility with blockchain platforms. These schemes assume that the documents and encrypted indexes are placed on the server, and the blockchain acts as a fair judge, ensuring that all parties behave honestly. However, the introduction of the transaction paradigm makes the construction of SSE nonintuitive and poorly scalable. Moreover, these schemes do not implement decentralized storage because the encrypted indexes are still stored on a single server.

The advent of smart contracts provides solutions to the aforementioned issues. Hu *et al.* [8] proposed the first smart contract-based SSE scheme, in which the index storage and search operations are performed by the smart contract, ensuring fair transactions for all parties. Following the work of Hu *et al.* [8], some efficient schemes (*e.g.*, [6,9,10,13,19]) were proposed to enhance security and functionality. The introduction of smart contracts brings inherent fairness and decentralized storage to the schemes but at the expense of a significant overhead that limits the utility of SSE.

Consequently, some works (*e.g.*, [20,21]) still store encrypted indexes on cloud servers, while smart contracts are responsible for result verification. Essentially, these works replace the transaction paradigms of [15,22] with smart contracts to shield the details of transaction-related operations. However, it remains controversial whether these schemes achieve fairness. Even though these works use MACs or digital signatures to ensure verifiability of results, if a data owner uploads faulty tags (or proofs) in the setup phase, the judge may wrongly conclude that the server is dishonest even if it returns the correct result. Cai *et al.* [4] proposed a fair SSE framework based on smart contracts, in which a voluntary

"arbitration panel" is responsible for verifying the results by simulating the index and search process. The dishonest party is determined by voting. This scheme is effective against the malicious behavior of both users and servers. However, it is an open problem to ensure the motivation and majority reliability of the arbiters continuously. Tang *et al.* [17] shift the responsibility of arbitration to a smart contract, eliminating the need to rely on volunteers to ensure fairness. However, the index reconstructions and search simulations of the smart contract incur a significant validation overhead.

In general, existing SSE schemes do not provide efficiency, fairness, and decentralized storage concurrently.

Contribution. This paper uses the aforementioned challenges as a springboard for proposing a decentralized and fair searchable symmetric encryption system based on blockchain. We choose to store the encrypted index on the blockchain for decentralized storage and try to alleviate the storage and search burdens. In this paper, we build up the decentralized fair SSE framework in a hierarchical manner and conduct experiments to evaluate its practical performance. Specifically, our work makes the following contributions.

- We suggest a practical and efficient way to store and read data on blockchain. We first provide an abstract model of blockchain storage called Append-only Block Storage (ABS), on which the subsequent designs will depend. Subsequently, we present a lookup table data structure Π_{LT} based on ABS and an implementation of it, a B' tree, which is a minor modification of the B^+ tree where nodes are stored via ABS blocks. The B' trees enable high fanout and low tree height to alleviate the performance bottleneck caused by reading ABS blocks.
- Based on ABS and Π_{LT}, we devise a decentralized publicly verifiable SSE scheme $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$. The proposed scheme integrates the design ideas of [3] and [5] and uses digital signatures to enable data confidentiality and public verifiability. In our design, encrypted indexes are organized as ABS-based lookup tables, and search operations are shifted off the chain, which ensures lightweight on-chain storage and efficient query performance.
- We develop a decentralized fair SSE framework Π_{fair} based on Ethereum [18], which empowers $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$ with fairness. It uses smart contracts to guarantee fair transactions between data users and service providers. We build a game paradigm in which all participants tend to behave faithfully, thereby avoiding deliberate fraud and resource waste.

2 Overview

2.1 System Model

We employ smart contracts to devise the decentralized fair SSE framework Π_{fair}. The framework consists of three types of entities: (i) data users (DUs), (ii) service

providers (SPs), and (iii) a smart contract (SC). A data user is an entity that wants to store its sensitive data on the blockchain and enjoy encrypted search services. The DU does not store the complete blockchain data locally, so it needs to outsource the query operation to SPs, which are full nodes. To ensure the fairness of the outsourced search, the DU submits a query request as a task on the SC with remuneration and the task deadline. The SP decides whether to participate in the task based on the task information and pays a deposit if it does. All the participants compete to find the desired result for remuneration. When one of them successfully finds the data from the blockchain, it sends the result and the corresponding proof to the SC for verification. If the validation succeeds (*i.e.*, the result and the proof match), the SC returns the result to the DU and issues the remuneration to the winning SP.

If the SP finds a problem with the outsourced task, it declares the task invalid to seek compensation. When the task deadline passes and none of the participants can find the result, the SC checks whether any SP has declared the task invalid. If so, the data user's remuneration is seized and compensated to the SP who declared the task invalid, and all the SPs' deposits are refunded; if not, the remuneration is returned to the DU, and the deposits of participants are refunded to their original location. To prevent dishonest SPs from maliciously declaring a task invalid, if there exists an SP who finds the result and passes the verification, the SC seizes the dishonest complainants' deposit and releases it to the winner.

2.2 Threat Model

Considering the realistic scenarios, we assume that SPs and DUs are potentially malicious: 1) the SP may return incorrect results in an attempt to cheat the remuneration, 2) the DU may submit an incorrect query request to squander the SPs' computational resources or reject the correct result to refuse to pay the remuneration.

In addition, all blockchain peers can monitor the traffic flowing through the smart contract, including search tokens, results, and proofs, from which they may learn some sensitive information of data.

2.3 Append-Only Block Store

We turn our focus to the study of efficient storage on blockchain. In the literature, most SSE schemes work on random access storage devices. Although there exist some blockchain-based SSE schemes whose underlying storage does not support random access, they are built on a higher-level abstraction, making them work on "virtual" random access storage devices. Expressly, these studies assume that the fragmented append-only data storage has been transformed into a "random access" view of storage through some protocol, such as smart contracts.

However, existing storage abstractions are inefficient due to the performance drain caused by their conversion mechanism. For instance, some SSE schemes that use smart contracts to store indexes generate many transactions in the

setup phase, burdening the blockchain network and costing the data owner a significant amount of money. Our goal is to propose a simple and efficient storage abstraction that can be built on top of blockchain transactions or other types of data shards while avoiding the enormous overhead associated with complex conversion operations.

We introduce an append-only block store (ABS) as a storage abstraction for blockchains. ABS is a subset of the block storage model, where anyone cannot alter previously written blocks and can only add new ones to ABS. ABS returns the block address when a block is appended, which is used to access the data later. The ABS block is limited in length by the public parameter γ. When the length of written data exceeds γ, ABS stops writing rather than slicing the data, requiring the caller to slice the data itself. Without sacrificing generality, we will assume that the length of a block address is constant, denoted by l_{addr}.

We now define the ABS model with a modification of the ADS model proposed by [1] to explicitly constrain the block size. An append-only block store $\Pi_{\mathsf{ABS}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put})$ consists of three algorithms:

- $\mathsf{ABS} \leftarrow \mathsf{Init}(\gamma)$: is an initialization algorithm that takes as input a public parameter γ specifying the maximum block length and outputs an empty append-only block store ABS.
- $v/\perp \leftarrow \mathsf{Get}(\mathsf{ABS}, \mathsf{addr})$: is an algorithm that takes as input an append-only block store ABS and an address addr. If the block specified by addr exists, it returns the block content v; otherwise, it returns $\perp$.
- $(\mathsf{ABS}', \mathsf{addr}) \leftarrow \mathsf{Put}(\mathsf{ABS}, v)$: is an algorithm that takes as input an append-only block store ABS and a value v to be written. If the length of v is greater than the public parameter γ, the algorithm aborts. Otherwise, it outputs the address addr associated with v and the updated append-only block store ABS'.

In our design, when someone wants to write data into the ABS, it needs to broadcast the data through some API in some medium (*e.g.*, transactions). Then, the entire blockchain network writes the data to the ABS through mining.

To read data from ABS, full-node SPs can call the method Get efficiently because they store the complete blockchain data locally, which is an off-chain operation. DUs can also implement the method through some API to establish a connection to a full node, which leads to high latency. Therefore, weighing performance and security, we assume that DUs call the method Put by themselves to guarantee the integrity of written data, outsource heavy operations involving multiple Get method calls to SPs for efficient reads, and take some measures to guarantee reliable reads, which we will describe below.

2.4 ABS-Based Lookup Table

Further, we propose a lookup table data structure Π_{LT} adapted to the ABS model. Π_{LT} provides two algorithms: the initialization algorithm LTInit and the query algorithm LTGet. Unlike conventional lookup tables, Π_{LT} writes data to the ABS only once during initialization and does not permit update operations.

Formally, an ABS-based lookup table $\Pi_{\mathsf{LT}} = (\mathsf{LTInit}, \mathsf{LTGet})$ contains two algorithms:

- $(\mathsf{LT}, \mathsf{ABS}') \leftarrow \mathsf{LTInit}(\{(l_1, v_1), \ldots, (l_n, v_n)\}, \mathsf{ABS})$: is an algorithm that takes as input n label/value pairs and an append-only block store ABS, then it outputs the updated append-only block store ABS' and a lookup table stored on ABS'.
- $v/\perp \leftarrow \mathsf{LTGet}(l, \mathsf{LT}, \mathsf{ABS})$: is an algorithm that takes as input a label, an append-only block store ABS and a lookup table LT stored on ABS. If the label l exists in LT, it outputs the corresponding value stored in LT; otherwise, it returns $\perp$.

Recall that the algorithm LTGet is off-chain for full nodes such as SPs. In the whole paper, the lengths of keys and values in Π_{LT} are fixed, denoted by l_{key} and l_{value}, respectively.

2.5 ABS-Based Publicly Verifiable Searchable Symmetric Encryption

Assume that there is a collection of D documents with identifiers id_1, id_2, ..., id_D. A database $\mathsf{DB} = (id_i, W_i)_{i=1}^{D}$ is a tuple of identifier/keyword-set pairs where $id_i \in \{0,1\}^{l_{id}}$ and $W_i \subseteq \{0,1\}^*$, such that keyword $w \in W_i$ if and only if the file identified by id_i contains the keyword w. The set of keywords contained in DB is $W = \bigcup_{i=1}^{D} W_i$. Let $\mathsf{DB}(w) = \{id_i | w \in W_i\}$ denote the set of documents containing keyword w, and N the number of document/keyword pairs (*i.e.*, $N = \sum_{i=1}^{D} |W_i|$).

We devise an ABS-based publicly verifiable SSE (PVSSE) scheme without considering fairness, which we will introduce in the next section. The blockchain nodes are only regarded as ordinary servers storing encrypted indexes and performing search operations without the function of arbiters, which is consistent with the traditional system model. In our setting, the search results are publicly verifiable, *i.e.*, all entities can use the DU's public key to verify the correctness of the results, which lays the foundation for our fair SSE framework construction.

An ABS-based publicly verifiable SSE scheme $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}} = (\mathsf{KeyGen}, \mathsf{EDBSetup}, \mathsf{TokGen}, \mathsf{Search}, \mathsf{Verify})$ contains five algorithms:

- $(PK, SK) \leftarrow \mathsf{KeyGen}(1^\lambda)$: is a key generation algorithm run by the DU. It takes as input a security parameter λ and then outputs a public key PK and a secret key SK, where PK is open to the public, and SK is kept in secret by the user.
- $(\mathsf{EDB}, \mathsf{ABS}') \leftarrow \mathsf{EDBSetup}(SK, \mathsf{DB}, \mathsf{ABS})$: is run by the DU to encrypt the given database. It takes as input a secret key SK, a database DB, and an append-only block store ABS and then outputs an encrypted database EDB stored on the updated store ABS'.
- $\tau \leftarrow \mathsf{TokGen}(SK, w)$: is a token generation algorithm run by the DU to generate a token for a keyword. It takes as input a string w and a secret key SK and outputs a search token τ.

- $(\mathcal{R}, \mathsf{prf}) \leftarrow \mathsf{Search}(\mathsf{EDB}, \tau, \mathsf{ABS})$: is a search algorithm run by the SP to search for the files that contains the keyword w. It takes as input τ, EDB, and ABS and then outputs the result $\mathcal{R}$ and the corresponding proof prf. Note that the search operations are off-chain.
- $\mathsf{accept/reject} \leftarrow \mathsf{Verify}(PK, \tau, \mathcal{R}, \mathsf{prf})$: is a verification algorithm run by any entity to check whether $\mathcal{R}$ is correct and complete. It takes as input a public key PK, a token τ, a set of results $\mathcal{R}$, and a proof prf, and outputs accept if $\mathcal{R}$ matches prf. Otherwise, it outputs reject.

The definition of ABS-based PVSSE is almost the same as that of traditional SSE, except that the storage model is changed to ABS. Moreover, the security and soundness definitions of ABS-based PVSSE are also compatible with those of the traditional verifiable SSEs, which will not be discussed in detail due to space constraints.

For simplicity, the formalization of PVSSE here does not involve modeling the storage of the actual file payloads. There is no agreement in the literature of SSE in dealing with this issue. Considering the case of decentralized environments, we argue that the encrypted files can be stored in any decentralized file system, such as IPFS.

2.6 Cryptographic Primitives

Pseudo-random Function. A pseudo-random function (PRF) $F : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ is a polynomial-time computable function that cannot be distinguished from a truly random function by any polynomial-time adversary. The formal definition of PRFs is given in [11].

Digital Signature. A digital signature scheme is a triple of algorithms $\Pi_{\text{sig}} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$. The probabilistic key generation algorithm KeyGen takes as input a security parameter and outputs a pair (pk, sk), where sk is called a secret signing key, and pk is called a public verification key. The probabilistic signing algorithm Sign takes as input a secret key sk and a string m and then outputs a signature σ. The deterministic verification algorithm Verify takes as input a public key pk, a message m, and a signature σ and then outputs either accept or reject. Informally, a digital signature scheme is secure if any polynomial-time adversary cannot forge a valid message/signature pair. We refer the reader to [11] for a formal definition of digital signatures.

Symmetric Encryption. We follow the definition of symmetric encryption in [5]. A symmetric encryption scheme is a pair of algorithms (E, D). The encryption algorithm E takes as input a key K and a plain text m and outputs a ciphertext c. The decryption algorithm D takes as input a key K and a ciphertext c, then it outputs m if c was produced by $E(K, m)$. We say that a symmetric encryption scheme is RCPA-secure (a stronger notion than CPA-secure) if the ciphertexts are computationally indistinguishable from truly random strings. The concrete definition of RCPA can be found in [5].

3 The Proposed Constructions

In this section, we give the specific constructions of the ABS-based lookup table, the publicly verifiable SSE, and the final fair SSE framework, respectively, in a step-by-step manner.

3.1 B′ Tree: An Implementation of the ABS-Based Lookup Table

We first propose read-only B′ trees based on the design concept of B^+ trees to instantiate the ABS-based lookup table, where the nodes can be stored via ABS blocks. Similar to B^+ trees, B′ trees store all satellite data in the leaf nodes and only keywords and child pointers in the internal nodes. The difference is that B′ trees do not support update operations and require that all data be written simultaneously in the setup phase. We retain the links between the leaf nodes to facilitate range queries.

We define that an internal node of an M-order B′ tree can hold up to M children. Each node x of a B′ tree has $x.n$ fixed-length keys $x.key_1, \ldots, x.key_{x.n}$ in non-descent order and a boolean $x.leaf$ that marks whether x is a leaf node. Furthermore, if x is an internal node, it also contains $x.n + 1$ children $x.child_1, \ldots, x.child_{x.n+1}$, satisfying that if k_i be any key stored in a subtree rooted at $x.child_i$, then $k_1 < x.key_1 \leq k_2 < x.key_2 \leq \cdots \leq k_{x.n} < x.key_{x.n} \leq k_{x.n+1}$; if x is a leaf node, it additionally contains $x.n$ fixed-length values labeled by keys and a pointer $x.ptr_{next}$ to the next leaf node. For any node x, the lengths of its contained keys, values (if any), $x.n$, and $x.leaf$ are fixed and the same as those of other nodes, which we denote by l_{key}, l_{value}, l_n, and l_{bool}, respectively.

Other properties of B′ trees, as well as the search algorithm LTGet, are consistent with those of B^+ trees, and the reader is referred to [12] for more details.

Initialization Algorithm. Given n key/value pairs $(l_1, v_1), \ldots, (l_n, v_n)$, the initialization algorithm LTInit for constructing an M-order B′ tree is as follows:

1. If $n = 0$, return $\perp$; otherwise:
2. Sort key/value pairs $(l_1, v_1), \ldots, (l_n, v_n)$ in non-descent order according to the key. The result is $(l'_1, v'_1), \ldots, (l'_n, v'_n)$.
3. Slice the ordered key/value pair $\{(l'_1, v'_1), \ldots, (l'_n, v'_n)\}$ into $\lceil n/(M-1) \rceil$ subsets $\{B_1, B_2, \ldots, B_{\lceil n/(M-1) \rceil}\}$ evenly, which means that the size of the last two subsets satisfies $\left|B_{\lceil n/(M-1) \rceil - 1}\right| - \left|B_{\lceil n/(M-1) \rceil}\right| \leq 1$, while the size of the rest is $M - 1$.
4. For subsets $\mathbf{B} = \{B_1, B_2, \ldots, B_{\lceil n/(M-1) \rceil}\}$, call the algorithm $\mathsf{LeafBuild}$ shown in Fig. 1 to generate a B′ tree from the bottom up and return the address of the root node as LT.

3.2 $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$ Construction

Based on the ABS-based lookup table, we further illustrate the detailed construction of $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$, which combines with the ideas of Π_{bas} in [5] and the verifiable

LeafBuild $(\mathbf{B} = \{B_1, B_2, \ldots, B_m\}, \mathsf{ABS})$

If $\mathbf{B}$ is empty, then return $\perp$; otherwise:
Initialize m empty leaf nodes $x_1, \ldots, x_m$
$p \leftarrow \perp$
for $i \leftarrow m$ **to** 1 **do**
  Write the key/value pairs contained in B_i to the leaf node x_i
  $x.ptr_{next} \leftarrow p, \quad x.n \leftarrow |B_i|$
  $addr_i \leftarrow \mathsf{Put}(\mathsf{ABS}, x_i)$
  $p \leftarrow addr_i$
Let κ_i be the smallest key of the subset B_i, where $2 \le i \le m$
return InternalBuild$(\{\kappa_2, \ldots, \kappa_m\}, \{addr_1, \ldots, addr_m\}, \mathsf{ABS})$

InternalBuild $(\mathbf{K} = \{k_1, \ldots, k_m\}, \mathbf{ADDR} = \{addr_1, \ldots, addr_{m+1}\}, \mathsf{ABS})$

If $\mathbf{K}$ is empty, then return $addr_1$; otherwise:
Initialize $\lfloor m/M \rfloor$ empty internal nodes $x_1, \ldots, x_{\lfloor m/M \rfloor}$
Initialize two empty lists $\mathbf{K}', \mathbf{ADDR}'$
$i \leftarrow 0, j \leftarrow 1$
while $m - i \ge M$ **do**
  $i' \leftarrow i$
  $i \leftarrow i + \min(M - 1, \lceil (m - i)/2 \rceil)$
  $\kappa_j \leftarrow k_i$
  $x_j.n \leftarrow i - i' - 1$
  Write $k_{i'+1}, \ldots, k_{i-1}$ to $x_j.key_1, \ldots, x_j.key_{x_j.n}$
  Write $addr_{i'+1}, \ldots, addr_i$ to $x_j.child_1, \ldots, x_j.child_{x_j.n+1}$
  $(\mathsf{ABS}, addr'_j) \leftarrow \mathsf{Put}(\mathsf{ABS}, x)$
  Push κ_j to $\mathbf{K}'$, and push $addr'_j$ to $\mathbf{ADDR}'$
  $j \leftarrow j + 1$
$x_{\lfloor m/M \rfloor}.n \leftarrow m - i$
Write $k_{i+1}, \ldots, k_m$ to $x_{\lfloor m/M \rfloor}.key_1, \ldots, x_{\lfloor m/M \rfloor}.key_{x_{\lfloor m/M \rfloor}.n}$
Write $addr_{i+1}, \ldots, addr_{m+1}$ to $x_{\lfloor m/M \rfloor}.child_1, \ldots, x_{\lfloor m/M \rfloor}.child_{x_{\lfloor m/M \rfloor}.n+1}$
$(\mathsf{ABS}, addr'_{\lfloor m/M \rfloor}) \leftarrow \mathsf{Put}(\mathsf{ABS}, x_{\lfloor m/M \rfloor})$
Push $addr'_{\lfloor m/M \rfloor}$ to $\mathbf{ADDR}'$
return InternalBuild$(\mathbf{K}', \mathbf{ADDR}', \mathsf{ABS})$

Fig. 1. Tree build algorithm of B′ Tree.

hash table (VHT) in [3]. Let $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ be a variable-input-length PRF, $\mathsf{LT} = (\mathsf{LTInit}, \mathsf{LTGet})$ be an ABS-based lookup table, Π_{sig} be a digital signature scheme, and $\mathcal{E} = (E, D)$ be a symmetric encryption scheme. The detailed construction is given in Fig. 2.

Unlike previous verifiable SSE schemes, our design uses digital signatures instead of MACs to enable public verifiability by DUs' public keys. On the skeleton of Π_{bas}, we embed the VHT into the construction, replace MACs with

KeyGen(1^λ)

$K \leftarrow\$ \{0,1\}^\lambda, K' \leftarrow\$ \{0,1\}^\lambda$
$(pk, sk) \leftarrow \Pi_{\mathsf{sig}}.\mathsf{KeyGen}(1^\lambda)$
return $(PK = pk, SK = (K, K', sk))$

EDBSetup$(SK, \mathsf{DB}, \mathsf{ABS})$

Parse SK as (K, K', sk)
Initialize three empty lists L, L', L''
foreach $w \in W$ **do**
  $K_1 \leftarrow F(K, 1\|w), K_2 \leftarrow F(K, 2\|w)$
  $K_3 \leftarrow F(K', 1\|w), \mathsf{wtag} \leftarrow F(K', 2\|w)$
  Initialize a counter $c \leftarrow 0$
  foreach $id \in \mathsf{DB}(w)$ **do**
    $l \leftarrow F(K_1, c), d \leftarrow E(K_2, id)$
    $c \leftarrow c + 1$
    Push (l, d) to L
  $\widetilde{id} \leftarrow id_1\|id_2\| \ldots \|id_{N_w}$, where $id_i \in \mathsf{DB}(w)$
  $\mathsf{prf} \leftarrow \Pi_{\mathsf{sig}}.\mathsf{Sign}(\mathsf{wtag}\|\widetilde{id})$
  $d_{\mathsf{prf}} \leftarrow E(K_3, \mathsf{prf})$
  Push $(\mathsf{wtag}, d_{\mathsf{prf}})$ to L'
$(\mathsf{LT}_{\mathsf{in}}, \mathsf{ABS}) \leftarrow \mathsf{LTInit}(L, \mathsf{ABS})$
Sort L' in ascending lexicographic order of keys
$i \leftarrow 0$
for $(\mathsf{wtag}, d_{\mathsf{prf}}) \in L'$ **do**
  $\mathsf{prf}' \leftarrow \Pi_{\mathsf{sig}}.\mathsf{Sign}(sk, \mathsf{wtag}\|i)$
  Push $(\mathsf{wtag}, d_{\mathsf{prf}}, i, \mathsf{prf}')$ to L''
  $i \leftarrow i + 1$
$(\mathsf{LT}_{\mathsf{prf}}, \mathsf{ABS}) \leftarrow \mathsf{LTInit}(L'', \mathsf{ABS})$
return $(\mathsf{EDB} = (\mathsf{LT}_{\mathsf{in}}, \mathsf{LT}_{\mathsf{prf}}), \mathsf{ABS})$

TokGen(SK, w)

Parse SK as (K, K', sk)
$K_1 \leftarrow F(K, 1\|w), K_2 \leftarrow F(K, 2\|w)$
$K_3 \leftarrow F(K', 1\|w), \mathsf{wtag} \leftarrow F(K', 2\|w)$
return $\tau = (K_1, K_2, K_3, \mathsf{wtag})$

Search$(\mathsf{EDB}, \tau, \mathsf{ABS})$

Parse EDB as $(\mathsf{LT}_{\mathsf{in}}, \mathsf{LT}_{\mathsf{prf}})$ and τ as $(K_1, K_2, K_3, \mathsf{wtag})$
Initialize an empty list $\mathcal{R}$
for $c \leftarrow 0$ **until** LTGet returns $\perp$ **do**
  $d \leftarrow \mathsf{LTGet}(F(K_1, c), \mathsf{LT}_{\mathsf{in}}, \mathsf{ABS})$
  $id \leftarrow D(K_2, d)$
  Push id to $\mathcal{R}$
if $\mathcal{R} \neq \emptyset$ **do**
  $(d_{\mathsf{prf}}, i, \mathsf{prf}') \leftarrow \mathsf{LTGet}(\mathsf{wtag}, \mathsf{LT}_{\mathsf{prf}}, \mathsf{ABS})$
  $\mathsf{prf} \leftarrow D(K_3, d_{\mathsf{prf}})$
else
  Find i such that $\mathsf{wtag}_i < \mathsf{wtag} < \mathsf{wtag}_{i+1}$
  $(d_{\mathsf{prf}_i}, i, \mathsf{prf}'_i) \leftarrow \mathsf{LTGet}(\mathsf{wtag}_i, \mathsf{LT}_{\mathsf{prf}}, \mathsf{ABS})$
  $(d_{\mathsf{prf}_{i+1}}, i+1, \mathsf{prf}'_{i+1}) \leftarrow \mathsf{LTGet}(\mathsf{wtag}_{i+1}, \mathsf{LT}_{\mathsf{prf}}, \mathsf{ABS})$
  $\mathsf{prf} \leftarrow (\mathsf{wtag}_i, i, \mathsf{prf}'_i, \mathsf{wtag}_{i+1}, \mathsf{prf}'_{i+1})$
return $(\mathcal{R}, \mathsf{prf})$

Verify$(PK, \tau, \mathcal{R}, \mathsf{prf})$

Parse τ as $(K_1, K_2, K_3, \mathsf{wtag})$
if $\mathcal{R} \neq \emptyset$ **do**
  Parse $\mathcal{R}$ as $(id_1, id_2, \ldots, id_{|\mathcal{R}|})$
  $\widetilde{id} \leftarrow id_1\|id_2\| \ldots \|id_{|\mathcal{R}|}$
  return $\Pi_{\mathsf{sig}}.\mathsf{Verify}(PK, \mathsf{wtag}\|\widetilde{id}, \mathsf{prf})$
else
  Parse prf as $(\mathsf{wtag}_i, i, \mathsf{prf}'_i, \mathsf{wtag}_{i+1}, \mathsf{prf}'_{i+1})$
  if $\mathsf{wtag}_i < \mathsf{wtag} < \mathsf{wtag}_{i+1}$ **do**
    return $\Pi_{\mathsf{sig}}.\mathsf{Verify}(PK, \mathsf{wtag}_i\|i, \mathsf{prf}'_i)$ and $\Pi_{\mathsf{sig}}.\mathsf{Verify}(PK, \mathsf{wtag}_{i+1}\|i+1, \mathsf{prf}'_{i+1})$
  else
    return reject

Fig. 2. The detailed construction of $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$.

digital signatures, and simplify some operations to meet the smart contract environment.

Specifically, to build the encrypted database, the key generation algorithm KeyGen called by the DU selects two keys K, K', where K is used to derive keys for PRF (to derive the retrieving labels) and encryption (to encrypt the identifiers) per keyword, and similarly K' is used to derive keys for PRF (to derive the proof labels) and encryption (to encrypt the proof information) per keyword. In addition, KeyGen invokes the underlying digital signature scheme to obtain the

signing key and verification key. Subsequently, the setup algorithm $\mathsf{EDBSetup}$ iterates over the identifiers in $\mathsf{DB}(w)$ for each keyword w. For each identifier, it computes a retrieving label by applying the PRF to a counter, encrypts the identifier, and adds the retrieving label/ciphertext pair to a list L. To achieve public verifiability, it also uses PRF to derive a proof label wtag for each keyword w, concatenates and signs all the document identifiers in $\mathsf{DB}(w)$, and then creates a list L' of all label/signature pairs. In order to prevent malicious services from returning faulty empty results, it sorts the list L', assigns the ordinal, and generates another signature on the ordinal and the label wtag for each item. Finally, it obtains a list L'' of quadruples of the form (wtag, a signature on the result, an ordinal, a signature on the ordinal) and creates two ABS-based lookup tables $\mathsf{LT}_{\mathsf{in}}$ and $\mathsf{LT}_{\mathsf{prf}}$ from L and L'', respectively.

To search for keyword w, the DU re-derives the keys and the proof label wtag for w and sends them to the SP. The search algorithm called by the SP starts by computing retrieving labels and decrypting the result. If the result is not empty, it looks up the signature prf corresponding to the result from $\mathsf{LT}_{\mathsf{prf}}$ and returns the result and signature. By contrast, if the result is empty, the algorithm queries $\mathsf{LT}_{\mathsf{prf}}$ for the two labels wtag_i and wtag_{i+1} adjacent to wtag, and returns the two labels, their ordinals, and the corresponding signatures.

The verification algorithm takes the following checks depending on whether the result is empty. If the result is not empty, it verifies the signature on the result. Otherwise, it checks whether wtag is between wtag_i and wtag_{i+1}, and verifies the signatures on the ordinals of these two labels. If the verification passes, wtag does not exist in $\mathsf{LT}_{\mathsf{prf}}$, and hence the result does not exist.

We argue that for SPs, the search algorithm involving multiple Get calls of ABS is off-chain, which significantly costs less time than the counterparts where the search operations are executed by smart contracts.

3.3 Π_{fair} Construction

Based on the ABS-based PVSSE scheme, we finally give the specific construction of the fair SSE framework. Let $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$ be an ABS-based PVSSE scheme, of which the digital signature scheme Π_{sig} is provided by the specific blockchain platform. We give a formal construction of Π_{fair} in Fig. 3, where the global variable `msg.sender` denotes the method caller, `msg.value` denotes the fee attached to the call, and `currentTime` denotes the current time. The smart contract maintains a dictionary T, where the key is the task tag and the value is the task information, including the search token, the address of the encrypted index, and the task remuneration.

Setup. In the setup phase, each participant generates a public/private key pair (pk, sk) using a wallet program. Any peer node can deploy a smart contract of Π_{fair} on the blockchain. After being created, the smart contract initializes an empty dictionary T.

Subsequently, the DU generates an SSE key by calling $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}.\mathsf{KeyGen}$, then it generates and uploads the encrypted database by calling $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}.\mathsf{EDBSetup}$.

PublishTask(tag, tk, tx, ddl)

1: Assert $T[tag] == \perp$
2: Assert $ddl >$ `currentTime`
3: $P_{all} \leftarrow \emptyset, P_{cit} \leftarrow \emptyset$
4: $issuer \leftarrow$ `msg.sender`
5: $\$remuneration \leftarrow$ `msg.value`
6: Generate a tuple $\tau \leftarrow (tk, tx, ddl, issuer, \$remuneration, P_{all}, P_{cit})$
7: Put the key/value pair $[tag : t]$ to T
8: Broadcast the event of the arrival of a new task

Withdraw(tag)

1: Assert $T[tag] \neq \perp$
2: $(tk, tx, ddl, issuer, \$remuneration, P_{all}, P_{cit}) \leftarrow T[tag]$
3: Assert $issuer ==$ `msg.sender`
4: Assert $ddl >$ `currentTime` and $P_{all} = \emptyset$
5: Delete $T[tag]$
6: Send $\$remuneration$ to `msg.sender`

AnnounceResult$(tag, result, prf)$

1: Assert $T[tag] \neq \perp$
2: $(tk, tx, ddl, issuer, \$remuneration, P_{all}, P_{cit}) \leftarrow T[tag]$
3: Assert $issuer \neq$ `msg.sender`
4: Assert $ddl >$ `currentTime`
5: Assert `msg.sender` $\in P_{all}$
6: Assert `msg.sender` $\notin P_{cit}$
7: Assert $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}.\mathsf{Verify}(pk_{issuer}, tk, result, prf)$ returns accept
8: $\$award \leftarrow \$remuneration$
9: **foreach** $p \in P_{all}$ **do**
10: **if** $p \in P_{cit}$ **then**
11: $\$award \leftarrow \$award + \$remuneration$
12: **else** Send $\$remuneration$ to p
13: Send $\$award$ to `msg.sender` and inform `issuer` of the result
14: Delete $T[tag]$

Participate(tag)

1: Assert $T[tag] \neq \perp$
2: $(tk, tx, ddl, issuer, \$remuneration, P_{all}, P_{cit}) \leftarrow T[tag]$
3: Assert $issuer \neq$ `msg.sender`
4: Assert $ddl >$ `currentTime`
5: Assert $\$remuneration \leq$ `msg.value`
6: Assert `msg.sender` $\notin P_{all}$
7: $P_{all} \leftarrow P_{all} \cup \{$`msg.sender`$\}$
8: $\$change \leftarrow$ `msg.value` $- \$remuneration$
9: Send $\$change$ to `msg.sender`
10: $T[tag] \leftarrow (tk, tx, ddl, issuer, \$remuneration, P_{all}, P_{cit})$

ClaimInvalid(tag)

1: Assert $T[tag] \neq \perp$
2: $(tk, tx, ddl, issuer, \$remuneration, P_{all}, P_{cit}) \leftarrow T[tag]$
3: Assert $issuer \neq$ `msg.sender`
4: Assert $ddl >$ `currentTime`
5: Assert `msg.sender` $\in P_{all}$
6: Assert `msg.sender` $\notin P_{cit}$
7: $P_{cit} \leftarrow P_{cit} \cup \{$`msg.sender`$\}$

ClaimTimeout(tag)

1: Assert $T[tag] \neq \perp$
2: $(tk, tx, ddl, issuer, \$remuneration, P_{all}, P_{cit}) \leftarrow T[tag]$
3: Assert $ddl \leq$ `currentTime`
4: **if** $P_{cit} \neq \emptyset$ **then**
5: $\$refund \leftarrow \$remuneration/|P_{cit}|$
6: **foreach** $p \in P_{cit}$ **then**
7: Send $\$refund$ to p
8: **else**
9: Send $\$remuneration$ to $issuer$
10: **foreach** $p \in P_{all}$ **then**
11: Send $\$remuneration$ to p
12: Delete $T[tag]$

Fig. 3. The detailed construction of Π_{fair}.

Recall that in order to enable fairness, when generating the encrypted index, the DU needs to sign the $\mathsf{DB}(w)$ for each keyword w in DB as proof of faithful search execution by SPs.

Task Publishing and Withdrawl. The DU who wants to search for its data on the blockchain publishes a search task by calling $\Pi_{\mathsf{fair}}.\mathsf{PublishTask}$ with arguments (tag, tk, tx, ddl), where tag is the task identifier generated by the DU, tk

is the search token generated by calling the method $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}.\mathsf{TokGen}$, tx is the address of the encrypted index, and ddl is the task deadline. In addition, the DU needs to set a fee for the task, which will be included in the message sent to the SC.

The SC initially checks if the task tag exists in the dictionary T (*i.e.*, another task with the same tag has not finished) and if the task deadline is valid. Subsequently, the SC creates two empty sets P_{all} and P_{cit} and extracts the issuer and the task remuneration. Then the SC creates a tuple to record the task detail and adds the tuple as a value and tag as a key to the dictionary T. Finally, it broadcasts the arrival of the new task to the peers.

When the issuer stops outsourcing the search, it can submit a task withdrawal request by calling $\Pi_{\mathsf{fair}}.\mathsf{Withdraw}$ with argument tag. The SC will check the validity of the task and the canceler. If no SP has participated in the task, the SC permits the task cancellation, deletes the corresponding item from T, and refunds the remuneration to the issuer.

Task Participation. When a task is published, the SC broadcasts an event including the above task arguments to all subscribed SPs. Based on the content, the SP determines whether to participate. If it decides to participate and compete for the reward, it can call $\Pi_{\mathsf{fair}}.\mathsf{Participate}$ with the argument tag and attach the message with its deposit equivalent to the remuneration. The SC will check if the task, the participant, and the deposit meet the conditions. If the above conditions are satisfied, the SC adds the participant to the P_{all} and refunds the excess deposit.

After participating in the task, the SP utilizes the search token tk provided by the DU to search over the EDB whose address is specified by tx stored on the blockchain. When the SP finds the result and the related proof successfully, denoted by $result$ and prf respectively, it verifies whether $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}.\mathsf{Verify}(pk_{\mathrm{DU}}, tk, result, prf)$ returns accept using the public key pk_{DU} of the DU. If the equation holds, the SP calls $\Pi_{\mathsf{fair}}.\mathsf{AnnounceResult}$ with arguments $(tag, result, prf)$ to announce the successful completion of the task. After the SC receives the message, it first checks that the task and the participant are valid and verifies that the result returned by the SP is correct and complete by calling $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}.\mathsf{Verify}(pk_{\mathrm{DU}}, tk, result, prf)$. If the validation succeeds, which means that the SP performed the search honestly and returned the correct result, the SC performs a series of monetary operations and returns the deposit to the honest SPs (*i.e.*, those who have not invalidated the task). As for dishonest complainants, the SC seizes their deposits and sends them to the winner as a reward. Finally, the SC informs the DU of the correct result and removes the task information from T.

When the SP finds that the search token or encrypted index provided by the DU is wrong, or the proof previously incorporated in the encrypted index is invalid, it can raise a task invalidity complaint by calling $\Pi_{\mathsf{fair}}.\mathsf{ClaimInvalid}$ with parameter tag. The SC performs a series of validity checks and adds the

complainant to P_{cit}. Once the task has expired and no SP has found the result, the SC divides the remuneration equally as compensation to each complainant.

Task Expiration Without Any Winner. When the current time exceeds the deadline, anyone can raise a task expiration declaration to the SC. The SC validates the existence of the task and ensures that it has indeed expired. If no SP claims the task is invalid, the task remuneration is refunded; otherwise, the task remuneration is shared equally among the SPs that declared the task invalid. Finally, all SPs have their deposits refunded.

4 Security Analysis

In this section, we discuss the security of our proposed framework in terms of confidentiality, soundness, and fairness.

4.1 Confidentiality

We first discuss the confidentiality of the proposed ABS-based PVSSE construction $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$, which is the basis for that of the fair SSE framework Π_{fair}.

We follow the ideal/real simulation paradigm of SSE [2,5] to demonstrate the confidentiality of the scheme. We define the leakage function $\mathcal{L}$ of scheme $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$ as

$$\mathcal{L}(\mathsf{DB}, \boldsymbol{w}) = \left(N, \{\mathsf{DB}(w)\}_{w \in \boldsymbol{w}}, |W|\right),$$

where the leakage function $\mathcal{L}$ takes as input a database DB and a list of queries $\boldsymbol{w}$ and outputs the size of the database N, the plain file identifiers contained in the database DB for each query w, and the number of keywords $|W|$.

Theorem 1. *If F is a secure PRF and $\mathcal{E} = (E, D)$ is RCPA-secure, then $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$ is $\mathcal{L}$-secure against non-adaptive attacks.*

The proof of the theorem is basically identical to [5], and the sketch is given later; a complete and formal proof can be found in [5].

Proof Sketch: To prove non-adaptive security, we give the construction of the simulator $\mathcal{S}$, which takes as input the return value of the leak function $\mathcal{L}$ (*i.e.*, the size N of the database, $\mathsf{DB}(w)$ for each query keyword w, and the number of keywords $|W|$) and outputs the view of the server (*i.e.*, EDB) and the corresponding search token for each query. Without loss of generality, we assume that the adversary's queries $\boldsymbol{w}$ are non-repeating.

The simulator $\mathcal{S}$ iterates over the queries and generates $K_{i,1}, K_{i,2}, K_{i,3}, \mathsf{wtag}_i \leftarrow_{\$} \{0,1\}^{\lambda}$ for the i-th query w_i. Next, for each $id \in \mathsf{DB}(w_i)$, $\mathcal{S}$ calculates l, d, and d_{prf}, then it adds (l, d) to L and $(\mathsf{wtag}_i, d_{\mathsf{prf}})$ to L' as $\mathsf{EDBSetup}$ does. Subsequently, $\mathcal{S}$ adds random pairs to L until L has N items and creates $\mathsf{LT}_{\mathsf{in}}$ by calling $\mathsf{LTInit}(L, \mathsf{ABS})$. Similarly, $\mathcal{S}$ adds random pairs to L' until L' has $|W|$

items and creates $\mathsf{LT}_{\mathsf{prf}}$ (as in lines 16 to 22 of $\mathsf{EDBSetup}$). Finally, $\mathcal{S}$ outputs $\mathsf{EDB} = (\mathsf{LT}_{\mathsf{in}}, \mathsf{LT}_{\mathsf{prf}})$ and $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_q)$, where $\tau_i = (K_{i,1}, K_{i,2}, K_{i,3}, \mathsf{wtag}_i)$.

The hybrid argument given in [5] can also be applied to this proof. The first hybrid shows that selecting $K_{i,1}, K_{i,2}$ randomly is indistinguishable from deriving them from the secure PRF $F(K, \cdot)$. Similarly, the second hybrid shows that selecting $K_{i,3}, \mathsf{wtag}_i$ randomly is indistinguishable from deriving them from the secure PRF $F(K', \cdot)$. The third hybrid states that the unqueried k-v pairs in $\mathsf{LT}_{\mathsf{in}}$ and $\mathsf{LT}_{\mathsf{prf}}$ are pseudo-random. Therefore, the output produced by $\mathcal{S}$ is indistinguishable from the view of the real world. □

We turn to the confidentiality analysis of Π_{fair}. The confidentiality of Π_{fair} relies on the underlying PVSSE scheme $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$. SPs cannot sniff any information other than the leakage $\mathcal{L}$. It is noted that the scope of the leakage is extended from a single server to all peers of the blockchain network since the state of the smart contract is public.

4.2 Soundness

Intuitively, the soundness of PVSSE signifies that the server cannot forge a result/proof pair $(\mathcal{R}, \mathsf{prf})$ where $\mathcal{R}$ has not been previously signed by the user such that $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}.\mathsf{Verify}$ returns accept. Our design relies on the security of the underlying PRF and digital signature. Depending on whether the result forged by the server is empty, we analyze the soundness separately. On the one hand, if the forged result is not empty, the server cannot forge evidence unless it knows the signing key owned by the user or finds a collision of the retrieving labels computed by F. On the other hand, if the forged result is empty, the server needs to provide the two adjacent wtags, their ordinals, and the corresponding signatures, which is also a difficult problem for a server that does not know the secret signing key. Therefore, if the underlying PRF F and the digital signature scheme are secure, our scheme can be inferred to be sound.

The soundness of Π_{fair} is dependent on $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$. As long as $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$ is sound, Π_{fair} is equally sound, *i.e.*, a malicious SP cannot forge proof for a wrong result to deceive the smart contract.

4.3 Fairness

Π_{fair} achieves fairness by constructing a multi-party game in which both SPs and DUs tend to cooperate. We discuss the fairness of the scheme in several cases as follows.

- If DUs and SPs are honest, the scheme will work appropriately, the remuneration will be awarded to the winning SP, and all SPs will be returned their deposits.
- Malicious DUs may refuse to pay the remuneration, send a faulty search request, or upload fraudulent proof in the setup phase (*i.e.*, the proof and the result may not match, even if the result is correct). Refusal to pay is impossible because our scheme requires the DU to pay the remuneration beforehand.

The complaint mechanism can solve the latter two cases, *i.e.*, SPs can file a complaint if they find something wrong with the request or the proof. If none of the SPs finds the result until the end of the task, the smart contract can conclude that the task is wrong and seize the remuneration to recompense the complainants. Under the stimulation of the penalty mechanism, DU tends to behave honestly.

- Malicious SPs may send incorrect results or directly claim the task is invalid without performing any substantive operation. The former is impossible due to the soundness of $\Pi^{\mathsf{ABS}}_{\mathsf{PVSSE}}$. For the latter case, whenever an honest SP succeeds in finding the result and proof within the time limit, the deposits of the malicious SPs will be seized and compensated to the honest winner by the SC. With the combination of penalty and incentive mechanisms, SPs tend to behave honestly.
- It is crucial to set the task time appropriately to ensure fairness. Too short task time will lead to fraud from dishonest SPs, whereas too long task time may lead to the decline of user experience. Therefore, the task time needs to be set according to the specific network situation, the number of nodes, and the performance of each node.

5 Implementation and Experimental Results

5.1 Implementation Details

We implement the proposed schemes in Python and Solidity and conduct the experiments on a computer with an AMD Ryzen 5700G CPU, 32 GB of RAM, running Ubuntu 22.04. Inspired by [1], we use Ethereum to instantiate ABS by storing blocks via transactions, whose hashes are used for block addresses. We conduct our experiments on the Ganache test network and interface with the network via Metamask and the Ethereum API `web3.py`. The Init method of ABS is instantiated by generating a Metamask wallet, while the Put and Get methods are instantiated using the Ethereum APIs `web3.eth.sendTransaction` and `web3.eth.getTransaction`, respectively.

For cryptographic primitives, we use the Python cryptography package to implement symmetric encryption via AES-256 and PRF via HMAC-SHA-256. Furthermore, we implement Π_{sig} using Web3 APIs `web3.eth.accounts.sign` and `web3.eth.accounts.recover`, whose underlying signature algorithm is ECDSA with SHA3-256. Thus, the data user's signature key corresponds to the key pair of the Metamask wallet. For the fair SSE framework Π_{fair}, we program a smart contract `FairContract` using the Solidity language and deploy it to the Ganache test network at the cost of $2,946.65$k gas.

For full implementation, we set the parameters as follows: we set γ to 64 KB for ABS implementation, $l_{key} = l_{value} = l_{addr} = 32$ bytes (the hash length of an Ethereum transaction), $l_{bool} = 1$ byte and $l_n = 4$ bytes for B′ tree implementation, and $l_{id} = 8$ bytes for SSE implementation.

5.2 Performance Evaluation

For comparison, we implement the scheme of [8], where the storage and search of data are performed by the smart contract. We modify the scheme of [8] by removing the `Add` and `Delete` methods and restricting the writing of encrypted indexes to the setup phase. Due to the gas limit, we slice the massive $\mathsf{DB}(w)$ into multiple chunks of size $1,000$ and send each chunk to the smart contract in turn. We program a smart contract `SSEContract` and deploy it to the same test network at the cost of 930.89k gas.

Table 1. Database properties in our experiment.

DB name	(w, id) pairs	Distinct keywords
DB_1	100,381	1,000
DB_2	100,180	10,000
DB_3	99,596	5,000
DB_4	500,012	25,000

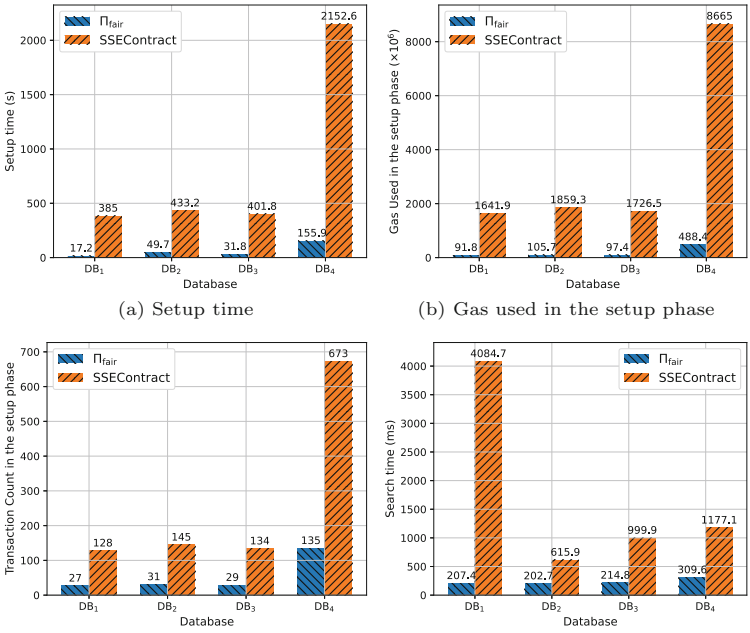
Using the Python Faker library and the `os.random` method, we generate four databases whose main properties are summarized in Table 1.

We measure the gas used, the number of transactions, the running time in the setup phase, and the running time and the gas used by each participant in the query phase.

Setup Performance. Figure 4(a)–(c) depict the time costs, gas usage, and transaction counts for Π_{fair} and `SSEContract` on various datasets. We can see that our design reduces the initialization time and gas consumption by about 90% and the number of transactions by approximately 80%. Specifically, when deploying DB_4 with large amounts of data, `SSEContract` requires more than 30 min, whereas our solution requires less than 3 min. The results demonstrate that our design achieves a significantly lower initialization overhead than `SSEContract`, indicating that Π_{fair} will reduce deployment costs significantly and be suitable for a wider range of use cases.

Table 2. The gas consumption of different entities for search in the unit of 10^3 gas.

	Gas Used By the DU				Gas Used By the winning SP				Gas Used By other SPs
	DB_1	DB_2	DB_3	DB_4	DB_1	DB_2	DB_3	DB_4	DB_1-DB_4
Π_{fair}	116.3				180.0	160.9	164.3	163.2	92.7
`SSEContract`	2,400.9	368.8	625.9	665.4					



(a) Setup time (b) Gas used in the setup phase

(c) Transaction count in the setup phase (d) Search time

Fig. 4. Efficiency evaluations.

Query Performance. In the query phase, we further measure the time cost and the gas usage of each party, averaging the results over 100 randomly selected queries. Figure 4(d) shows the "entire search time" for both schemes, which refers to the time interval between when the DU posts a search task and when it receives the correct result. Overall, our design outperforms `SSEContract` and performs substantially better on databases containing high-frequency keywords, owing to the underlying B′ tree, which reduces the frequency of reading ABS blocks per search.

Table 2 shows the comparison of the gas consumption of different entities for search between the two schemes. In our design, the gas consumption varies between different entities. For `SSEContract`, only the DU executing the smart contract method consumes gas. As Table 2 shows, in our design, the DU consumes only 116.3k of gas to publish the search task, and the SP uses 92.7k of gas for participation. In addition, the SP that finds the result consumes more gas due to the additional call to the AnnounceResult method, the amount of which varies depending on the size of the result. As for `SSEContract`, the gas consumed by the DU is positively correlated with the size of the result. Overall, our scheme efficiently reduces the amount of gas consumed for search compared to `SSEContract`, since the search operation with high consumption is shifted to off-chain execution, while the smart contract is only used for the lifecycle of the task.

6 Conclusion

Existing blockchain-based SSE schemes cannot simultaneously achieve high efficiency and fairness. This paper uses the aforementioned issue as a springboard to provide an effective and fair solution for searching for encrypted data on the blockchain. We first build a generic abstraction model ABS for blockchain storage, compatible with the majority of blockchain platforms. Based on ABS, we propose a lookup table data structure and an implementation of it to achieve efficient storage and search. We further propose a publicly verifiable SSE scheme in which indexes are organized as ABS-based lookup tables, and the search operations are shifted to be executed off the chain, thereby significantly reducing the time overhead and gas usage. Then, we use smart contracts to introduce fairness to SSE via multi-party gaming. We implement our scheme in Solidity and Python and deploy it on Ethereum. The experimental results show that our design is effective and practical.

Acknowledgement. This work is supported by Guangdong Major Project of Basic and Applied Basic Research (2019B030302008) and the National Natural Science Foundation of China (No. 61972429).

References

1. Adkins, D., Agarwal, A., Kamara, S., Moataz, T.: Encrypted blockchain databases. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies, pp. 241–254 (2020)
2. Asharov, G., Naor, M., Segev, G., Shahaf, I.: Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In: Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing, pp. 1101–1114 (2016)
3. Bost, R., Fouque, P.A., Pointcheval, D.: Verifiable dynamic symmetric searchable encryption: optimality and forward security. Cryptology ePrint Archive (2016)
4. Cai, C., Weng, J., Yuan, X., Wang, C.: Enabling reliable keyword search in encrypted decentralized storage with fairness. IEEE Trans. Depend. Secure Comput. (2018)
5. Cash, D., et al.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: NDSS, vol. 14, pp. 23–26. Citeseer (2014)
6. Chen, L., Lee, W.K., Chang, C.C., Choo, K.K.R., Zhang, N.: Blockchain based searchable encryption for electronic health record sharing. Futur. Gener. Comput. Syst. **95**, 420–429 (2019)
7. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, pp. 79–88 (2006)
8. Hu, S., Cai, C., Wang, Q., Wang, C., Luo, X., Ren, K.: 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)
9. Jiang, S., et al.: Privacy-preserving and efficient multi-keyword search over encrypted data on blockchain. In: 2019 IEEE International Conference on Blockchain (Blockchain), pp. 405–410. IEEE (2019)

10. Jiang, S., Liu, J., Wang, L., Yoo, S.M.: Verifiable search meets blockchain: a privacy-preserving framework for outsourced encrypted data. In: ICC 2019–2019 IEEE International Conference on Communications (ICC), pp. 1–6. IEEE (2019)
11. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press, Hoboken (2007)
12. Leiserson, C.E., Rivest, R.L., Cormen, T.H., Stein, C.: Introduction to Algorithms, vol. 3. MIT Press, Cambridge (1994)
13. Li, H., Gu, C., Chen, Y., Li, W.: An efficient, secure and reliable search scheme for dynamic updates with blockchain. In: Proceedings of the 2019 the 9th International Conference on Communication and Network Security, pp. 51–57 (2019)
14. Li, H., Tian, H., Zhang, F., He, J.: Blockchain-based searchable symmetric encryption scheme. Comput. Electr. Eng. **73**, 32–45 (2019)
15. Li, H., Zhang, F., He, J., Tian, H.: A searchable symmetric encryption scheme using blockchain. arXiv preprint arXiv:1711.01030 (2017)
16. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000, pp. 44–55. IEEE (2000)
17. Tang, Q.: Towards blockchain-enabled searchable encryption. In: Zhou, J., Luo, X., Shen, Q., Xu, Z. (eds.) ICICS 2019. LNCS, vol. 11999, pp. 482–500. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41579-2_28
18. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger. In: Ethereum Project Yellow Paper, vol. 151, pp. 1–32 (2014)
19. Xu, C., Yu, L., Zhu, L., Zhang, C.: A blockchain-based dynamic searchable symmetric encryption scheme under multiple clouds. Peer-to-Peer Network. Appl. **14**(6), 3647–3659 (2021). https://doi.org/10.1007/s12083-021-01202-6
20. Yan, X., Yuan, X., Ye, Q., Tang, Y.: Blockchain-based searchable encryption scheme with fair payment. IEEE Access **8**, 109687–109706 (2020)
21. Yang, Y., Lin, H., Liu, X., Guo, W., Zheng, X., Liu, Z.: Blockchain-based verifiable multi-keyword ranked search on encrypted cloud with fair payment. IEEE Access **7**, 140818–140832 (2019)
22. Zhang, Y., Deng, R.H., Shu, J., Yang, K., Zheng, D.: TKSE: trustworthy keyword search over encrypted data with two-side verifiability via blockchain. IEEE Access **6**, 31077–31087 (2018)