Using Smart Contracts to Improve Searchable Symmetric Encryption

Yanbing Wu[1] and Keting Jia[2](✉)

[1] Institute for Advanced Study, Tsinghua University, Beijing 100084, China
wuyb14@mails.tsinghua.edu.cn
[2] Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University, Beijing 100084, China
ktjia@mail.tsinghua.edu.cn

Abstract. In this paper, we propose a smart contract based searchable symmetric encryption scheme. The existing searchable symmetric encryption protocol can resist malicious servers when using the MAC algorithm; however, it is more effective only under the assumption that the server is running. If the server receives a user's money but does not provide a service to the user (or if the server shuts down after receiving the user's money), the user cannot withdraw the money paid. In addition, if the server wants to reduce computing costs, bandwidth, etc., then it may reduce the number of documents to be searched or omit part of the search results. As a result, there is no guarantee that all files have been searched. We use the Merkle tree to construct search integrity verification. Implementing search integrity verification ensures that it is nearly impossible for searchers to provide integrity verification without searching all documents. Smart contracts use computing resources effectively and help us better search the blockchain. All information is recorded on the blockchain and will not be tampered with. In addition, integrity verification and smart contracts slightly reduce the efficiency but are feasible in practice. Finally, we have theoretically and experimentally verified the safety and feasibility of the proposed scheme.

Keywords: Smart contract · Blockchain · Searchable symmetric encryption · Result integrity · Verifiable · Bloom filter

1 Introduction

A keyword index helps us search for documents containing specified keywords in a certain period of time. Security indexes are extensions of building data structures, e.g., indexes provided by unrelated data structures [13] and historically independent [2,12] data structures. Unfortunately, the standard index structure using hash tables is not suitable for indexing encrypted documents because they leak information about the contents of the document. However, data structures with privacy protection can be used to build secure indexes.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2021
Published by Springer Nature Switzerland AG 2021. All Rights Reserved
B. Chen and X. Huang (Eds.): AC3 2021, LNICST 386, pp. 181–202, 2021.
https://doi.org/10.1007/978-3-030-80851-8_14

To improve the efficiency of searchable symmetric encryption (SSE), Goh et al. proposed using the Bloom filter method to search symmetric encryption [4]. The Bloom filter proposed by Bloom is a space-saving random data structure that uses a bit array to represent a collection very concisely. It can determine whether an element belongs to this collection. The Bloom filter was used in the early UNIX spell checker [9,11]. The Bloom filter is also used as an index for each document to search the keywords [4] in each document. In the index, a keyword is represented by a codeword derived by applying a pseudorandom function twice, i.e., once using the keyword as input and once using a unique document identifier as input.

In 2017, Li et al. [8] combined blockchain technology with SSE, where a blockchain is used as a peer-to-peer network to store user data. This scheme adds data as blocks to the blockchain, thereby storing the data in the common chain. However, this scheme only supports single keyword search. Tang et al. [1] proposed two frameworks that support blockchain technology, and these frameworks can be applied to most existing SSE schemes to achieve fairness while maintaining the original privacy protection. However, these frameworks are inefficient. Zhang et al. [17] proposed a blockchain based searchable encryption scheme in multicloud environment. Here, multiple cloud service providers are combined to share data through an alliance chain. Then, the encrypted document and document index are stored in IPFs, and the hash value of the document is stored in the blockchain. This scheme can provide retrieval of outsourced encrypted data based on multiple keywords; however, it is based on trusted cloud service providers, which cannot resist malicious cloud servers.

We employ the Bloom filter to construct a searchable symmetric encryption scheme and use smart contracts to maintain the fairness of the searchable symmetric encryption scheme.

The smart contract was originally proposed by Nick Szabo in 1996 [14]. A smart contract is a computer protocol designed to facilitate, verify, or execute a contract. Note that a smart contract in the blockchain is traceable and cannot be tampered with [10]. Many contract clauses can be executed partially or completely in an independent manner. The purpose of smart contracts is to provide better security than traditional contracts and reduce other transaction costs associated with the contract. Various cryptocurrencies are implemented as types of smart contracts. A smart contract is a set of commitments specified in digital form, including an agreement for parties to fulfill these commitments [14].

Our Contribution: We exploit computing resources in the blockchain to propose a searchable symmetric encryption scheme based on smart contracts. To address incomplete retrieval of all files on the server, we first propose a non-interactive integrity verification method for search results. The proposed method uses Merkle trees to construct search integrity verification, which can ensure that it is nearly impossible for searchers to provide integrity verification without searching all documents. In addition, using smart contracts, we can use the computing resources in the blockchain effectively to perform ciphertext searches. As a result, neither the owner nor searcher can cheat. All information (including search

results, proof of integrity, and commission payment information) is recorded in the blockchain and cannot be tampered with. By implementing integrity verification and smart contracts, the proposed solution does not reduce efficiency significantly and ensures that searchers cannot obtain information from encrypted documents and search keywords. The proposed scheme guarantees the authenticity and completeness of the results, and, through the Merkle root signature, the proposed scheme ensures that searchers of search results will not be tampered with by miners after submitting the results.

The remainder of this paper is organized as follows. We define notations used in this paper in Sect. 2. In Sect. 3, we provide preliminary information about searchable symmetric encryption and smart contracts. In Sect. 4, we propose the scheme of the non-interactive integrity verification of search results for the first time. In Sect. 5, we present our scheme of smart contract based SSE and discuss the schedule in detail. In Sect. 6, we provide a security definition, security analysis, and performance analysis. Finally, the paper is concluded in Sect. 7.

2 Notations

We define the notations used in this paper in Table 1.

Table 1. Notations

Notation	Description
$\wedge$	Bitwise and
$H_1(), f()$	Pseudorandom function
n	Number of documents
m	Assume that each file has m keywords for brevity
r	Number of function $f()$ used in Bloom filter
s	Key length used in the f function
t	Key length of the encrypted file, the key length of the MAC
l	Security parameter, number of random numbers generated in the integrity verification of search results
n_{tran}	Transaction size
$H()$	Collision resistant hash function
$\|\|$	Concatenation
$bf(y)$	Obtain output of Bloom filter according to y

3 Preliminaries

We propose a searchable symmetric encryption scheme called smart contract-based SSE. The proposed scheme employs a non-interactive result integrity proof method, the Bloom filter, searchable symmetric encryption, and smart contract

technology. Here, we introduce the Bloom filter, searchable symmetric encryption and smart contract technology. Eu-jin Guo introduced a Bloom filter method to solve the problem of searchable symmetric encryption [4], which was used to solve the searchable symmetric encryption efficiency problem. We propose an improved scheme in their framework. Szabo proposed the smart contract in 1994 [15], and the proposed method employs smart contracts as a platform to increase the availability of computing resources.

3.1 Scheme of Searchable Symmetric Encryption

To improve the efficiency of searchable symmetric encryption, Eu-jin Guo introduced a Bloom filter method for searchable symmetric encryption [4] that included two participants, i.e., the file owner and the untrusted server. Here, assume the file owner has n files denoted $D_1, D_2, \ldots, D_n$. The file owner encrypts these documents into ciphertexts $C = (C_1, C_2, \ldots, C_n)$ and constructs the corresponding index set of I, and then sends C, I to the server. When the file owner wants to look up documents including keyword w, he calculates trapdoor T_w for keyword w and sends T_w to the server. The server searches for result C_i based on t_w, C and I, and then sends C_i to the file owner. Finally, the file owner decrypts C_i to obtain D_i.

Note that search symmetric encryption is considered secure if the server does not obtain any information about the plaintexts when it only knows ciphertexts or when it does not know any information about the plaintexts and keywords except the search results when it executes search algorithms.

Assume file owner U has n encryption files $C_1, C_2, \ldots, C_n$ on server $\mathcal{S}$. The SSE scheme comprises functions $(BuildIndex(\cdot), SearchIndex(\cdot))$ and three phases (**setup, search, update**). In the **setup** phase, indexes for $C_1, C_2, \ldots, C_n$ are built by the search system, the retrieval task is performed in the **search** phase, and the **update** phase updates the index when documents are changed, added, or deleted. The process of the **setup** phase is shown in Fig. 1.

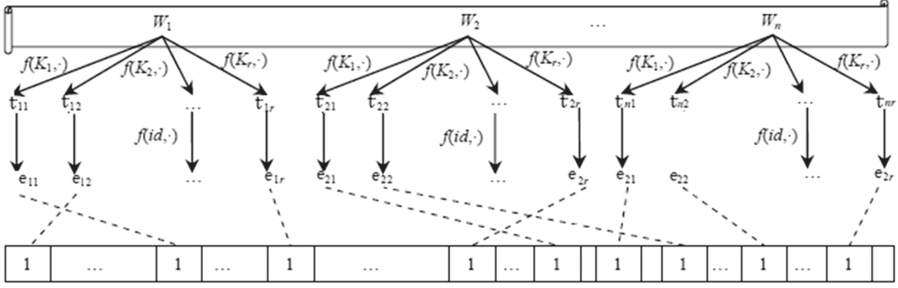

Fig. 1. The secure index setup phase

We introduce the $BuildIndex(\cdot)$ and $SearchIndex(\cdot)$ functions in the following.

- **BuildIndex**$(C, K_{trapdoor})$**:** The function of the input is an encrypted document C with unique file ID $C_{id} \in \{0,1\}^n$ and some keywords $w = (w_0, \ldots, w_t)$, as well as the corresponding keywords trapdoor $K_{trapdoor} = (k_1, \ldots, k_r) \in \{0,1\}^{sr}$, $k_i \in \{0,1\}^s$.
  1. For each document D_i for $i \in [0, n]$, we do the following:
     (a) We compute the trapdoor of each keyword w_i, i.e. $Buildtrapdoor$ $(K_{trapdoor}, w_i) = (t_1 = f(w_i, k_1), \ldots, t_r = f(w_i, k_r)) \in \{0,1\}^{sr}$.
     (b) The corresponding ciphertext ID C_{id} information is added to get the codewords $e : e_1, \ldots, e_r$, where $(e_1 = f(C_{id}, t_1), \ldots, e_r = f(C_{id}, t_r)) \in \{0,1\}^{sr}$.
     (c) The codewords $e : e_1, \ldots, e_r$ is added to the file D_i Bloom filter BF_i.
  2. Output the index of C: $I_C = (C_{id}, BF)$.
- **SearchIndex**(T_w, I_C)**:** The function of the input is the key word trapdoor $T_w = (t_1, \ldots, t_r) \in \{0,1\}^{sr}$ corresponding to keywords W, and the index $I_C = (C_{id}, BF)$ corresponding to encryption file C.
  1. Key word w is encoded according to C_{id} to obtain e, where $e : (e_1 = f(C_{id}, t_1), \ldots, e_r = f(C_{id}, t_r)) \in \{0,1\}^{sr}$.
  2. To determine if each position in y that contains a 1 corresponds to a 1 in the Bloom filter, we must determine if $(bf(e) \wedge BF) == bf(e)$ $(e = e_1, e_2, ..., e_r)$ is correct.
  3. If the above is correct, 1 is output; otherwise, 0 is output.

The **setup**, **search**, and **update** phases are described as follows.

- **Setup phase:** The n files are uploaded to the server after the index is constructed.
  1. First, the proper Bloom filter parameters are derived for each index. In addition, we define $K_{trapdoor} = (k_1, \ldots, k_r) \in \{0,1\}^{sr}$, $k_i \in \{0,1\}^s$.
  2. Each file is assigned a unique ID, which is an integer represented by $i \in [1, n]$.
  3. Each file builds an index as $I_{C_i} \leftarrow BuildIndex(C_i, K_{trapdoor})$.
  4. After each document is compressed and encrypted using standard algorithms, the document and its index can be placed on server $\mathcal{S}$.
- **Search phase:** When file owner $\mathcal{U}$ has a lookup requirement, server $\mathcal{S}$ must be queried for the given keyword w so server $\mathcal{S}$ can return the file containing the given keyword. Then, the following is performed.
  1. File owner $\mathcal{U}$ calculates the trapdoor of w to obtain $T_w \leftarrow Buildtrapdoor(K_{trapdoor}, w)$, and then sends T_w to the server $\mathcal{S}$.
  2. For all the indexes of I_{C_i}, server $\mathcal{S}$ calls SearchIndex (T_w, I_{C_i}) to test matches. All matched files are returned to file owner $\mathcal{U}$.
- **Update phase:** There are three possible scenarios in which an update operation may be used.

1. Add file: When a file is added to the system, a unique ID is first assigned to ciphertext file C, and then index I_{C_i} is created for C.
2. Delete file: When a file is deleted from the system, we must delete both the file and its index file.
3. Changing the contents of an existing document requires assigning a new unique ID to the document and rebuilding the index by calling the *BuildIndex* function with the new unique ID.

3.2 Smart Contract

Smart contracts are computer programs running on the blockchain. A smart contract comprises program code, stored files, and an account balance. Any user can create smart contracts by posting events to the blockchain. When creating a smart contract, the contract's program code is fixed and cannot be changed. As shown in Fig. 2, the contract storage file is stored in the blockchain. The contract's program logic is executed by miners who reach consensus on the execution results and update the blockchain accordingly. The contract code is executed when the user or another contract receives the message. When executing its code, the contract can read or write from its storage file. A contract can have an account to accept transfers from other accounts or contracts, or it can send transactions to other accounts or contracts. Conceptually, the contract can be considered a special "trusted third party." However, the party is only trusted for correctness (not privacy). Note that the entire state of the contract is visible to all nodes. Figure 2 shows a schematic diagram of a smart contract.

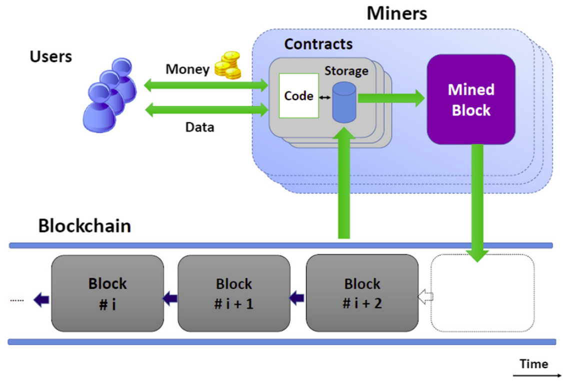

Fig. 2. Schematic of a smart contract

4 Scheme of the Integrity Verification of Search Results

Here, we introduce the non-interactive integrity verification scheme for search results. In this scheme, the Merkle tree is employed to verify the integrity of the

search results. The non-interactive integrity verification scheme can be verified on the blockchain. The searcher generates the result integrity certificate after the search is completed. The smart contract can verify whether the searcher has searched all the files according to the proof submitted by the searcher. The final submitted result is complete without any omission.

4.1 Merkle Tree

In cryptography and computer science, the Merkle tree is a tree in which each leaf node is marked with the hash value of a data block, and each non-leaf node is marked with the hash value of the label of its child nodes. Merkle trees, which are generalizations of hash tables and hash chains, allow efficient and safe verification of the contents of large data structures. To prove that a leaf node is part of a given binary hash tree, a logarithmic hash calculation of the number of leaf nodes of the tree is required. Note that this differs from hash lists because the number of hash lists is proportional to the number of leaf nodes. Figure 3 shows an example of a Merkle tree with four leaf nodes.

Merkle trees can be used to verify any type of data stored, processed, and transmitted on or between computers. They ensure that the data blocks received from other nodes in a point-to-point network are undamaged and unchanged, and can even check whether other peer nodes have tampered with the data or sent fake data. A Merkle tree is primarily used for data integrity verification based on a hash. Many systems use Merkle trees for data integrity verification, e.g., the IPFS, Btrfs, and ZFS file systems [18], as well as the Apache wave protocol [16].

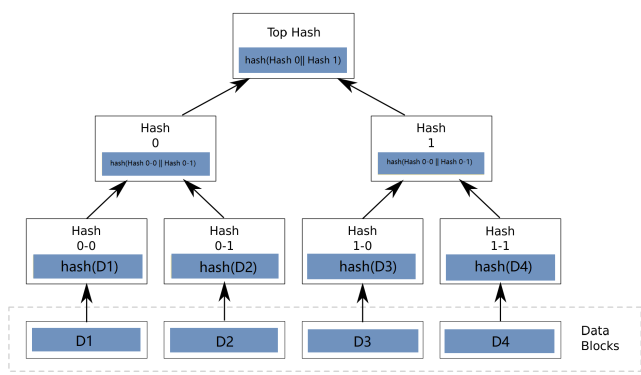

Fig. 3. Example Merkle tree with four leaf nodes

4.2 Proof of Integrity Verification of Search Results

Assuming there are n documents C, the keywords to be searched are represented by W. The file owner encrypts the keywords to trapdoor T_w and publishes

trapdoor T_w and index collections I ($I = (C, BF)$). The searcher then searches n encrypted documents according to the trapdoor T_w. The searcher calculates $bf(y)$ ($y = e_1, e_2, ...e_r$) according to T_w and document identifier C_{id}, and the search results are obtained based on $bf(y)$ and BF. To ensure the integrity of search results (that is, the searchers have searched all documents rather than only some documents), and can be use on blockchains, we propose the scheme of the non-interactive integrity verification of search results.

For the search results for the keywords in each document, we obtain the search results for each document where $search_i = (bf(y) \wedge BF_i)$ (the specific calculation process is described in Sect. 5.2.). We then use each result $search_i$ ($i = 1, 2..., n$) as a leaf node to build a Merkle tree. We obtain the corresponding root node Root when we finish constructing the Merkle tree. Here, the searcher uses their public key to sign the Merkle tree root node and obtain sign $= sign_{sk_{search}}(Root)$. Then, we use the hash function to generate a random number to construct the proof of the result's integrity. We compute $H(sign||i)$ ($i = 1, 2, ..., l$) to generate l random numbers, where l denotes the security parameter. Assuming we have n documents that need to be searched, we create an array to save a proof of the integrity of the search results, which we refer to as the array result integrity proof (RIP). We then put the $H(sign||i)$ mod n ($i = 1, 2, ..., l$) search result that is $H(sign||i)$ mod n and its path in the Merkle tree into the array RIP. We also create a Result array to save the search results. When the searcher completes the search, he outputs Result, RIP, and sign(Root). The algorithm used to search the document and construct the RIP is given in Algorithm 1.

Algorithm 1. Search document and prove integrity of search results.

Search(T_w, I): When a searcher sees a demand that he wants to help search the document, then the searcher will use the trapdoor $T_w = (t_1, \ldots, t_r) \in \{0,1\}^{sr}$ for word w to test every index I_{C_i} in indexed collections I.

1: **for** every C_i in C **do**
2: The key word w is encoded according to C_{id} to get e, and $e : (e_1 = f(C_{id}, t_1), \ldots, e_r = f(C_{id}, t_r)) \in \{0,1\}^{sr}$.
3: To see if each position in y that contains a 1 corresponds to a 1 in the bloom filter, we need to detect $(bf(e) \wedge BF) == bf(e)$ ($e = e_1, e_2, ..., e_r$) is correct, and add $(bf(y) \wedge BF_i)$ to the array search that $search_{H(sign||i)\ mod\ n}$ ($i = 1, 2, ..., l$).
4: If so, add C_i to the array Result.
5: **end for**
6: Use array search to build a Merkle tree and get a Merkle tree root Root.
7: Searcher use his public key to sign the Merkle tree root Root and get $sign = sign_{PK_{searcher}}(Root)$
8: Compute $H(sign||i)$ ($i = 1, 2, ..., l$), the l denote the security parameter, then add the $search_{H(sign||i)\ mod\ n}$ ($i = 1, 2, ..., l$) and the Merkle tree path into the array result integrity proof RIP.
9: Output Result, RIP and sign(Root).

When verifying the integrity of the search results, first we verify that the signature of the Merkle tree root. We then verify that the results in the array Result are correct. Here, according to the root of the Merkle tree, we compute $H(sign||i)$ $(i = 1, 2, ..., l)$ to generate l random numbers, based on the generated random numbers and the document to compute $search_{H(sign||i)\ mod\ n}$ $(i = 1, 2, ..., l)$. Then, we verify that the path which would be provided in the array RIP of the Merkle tree of $search_{H(sign||i)\ mod\ n}$ $(i = 1, 2, ..., l)$ is correct. The algorithm used to verify results and the integrity of the results is given in Algorithm 2.

Algorithm 2. Verification of results and result integrity proof

Verify(searcher, Result, RIP, sign(Root)): This function is used to verify the result and result integrity.

1: Verify that the signature of the Merkle tree root *Root* is signed by the searcher.
2: Verify that the results in the array *Result* are correct, if correct then continues, else incorrect return error 0.
3: According to the root of the Merkle tree to compute $search_{H(sign||i)\ mod\ n}$ $(i = 1, 2, ..., l)$.
4: Verify that the path which be given in the array RIP of the Merkle tree of $search_{H(sign||i)\ mod\ n}$ $(i = 1, 2, ..., l)$ is correct, if correct then continues, else incorrect return error 0.
5: According to the compute process given in the SearchIndex function, the result of the array RIP is correct, if correct return 1, else the error returns 0.

In the following, we discuss an example (Fig. 4) to illustrate the proposed scheme. Here, assume there are four documents. Thus, when the search is complete, we obtain four search results: $search_0$, $search_1$, $search_2$, $search_3$. For each search result, $search_0 = (bf(y) \wedge BF_0)$, $search_1 = (bf(y) \wedge BF_1)$, $search_2 = (bf(y) \wedge BF_2)$, $search_3 = (bf(y) \wedge BF_3)$. By using these four search results as a leaf node to build the Merkle tree, we obtain the corresponding root node. Assume Alice is searching for the result; thus, Alice signs the root with her private key. The signature and i are joined together. We set i to 1 and 2. Thus, we obtain two values: $(sign||1)$ and $(sign||2)$. After hash and modulus, we obtain two random numbers between 0 and 3. Here, assume that $H(sign||1)$ is equal to 0, and $H(sign||2))$ is equal to 2. We save $search_0$, $search_2$ and the corresponding Merkle tree path into to the RIP array. The $search_0$, $search_2$ and the corresponding Merkle tree path are used for subsequent result integrity verification. In addition, we assume that $search_1$ is the result of the search criteria. We save the $search_1$ in the array Result. If Bob is the verifier, Bob first verifies that root is Alice's signature. Then, Bob verifies that the results in the array Result are correct; thus, Bob verifies that $search_1$ is correct. Then, Bob according to compute $H(sign||1)\ mod\ 4$ and $H(sign||2)\ mod\ 4$ to generate two random numbers. Here, Bob obtains the two random numbers 0 and 2. Then, Bob verifies that $search_0$, $search_2$ and the corresponding Merkle tree path are correct, which completes the validation.

Assume the search is complete with p $(0 \leq p \leq 1)$, there are n documents that need to be searched, and $l(l \leq n)$ random numbers need to be generated. If the searcher does not fully search the entire document, only $p * n$ documents are searched. When validation of the Merkle tree is set up, the probability that the searcher build a Merkle tree will be able to verify at once is p^l. The larger the l option, the lower the probability that the searcher will be able to pass validation (when the searcher does not search the entire document), and the greater the cost of not fully searching the entire document and provide correct Result, RIP, and sign(Root). Here larger l results in more complete search results.

In the following, we examine an intuitive example; we see that a searcher searches 99% of the files, l is set to 100, and the probability that the searchers are successful in building the Merkle tree is 0.366. From the above mentioned findings, it is obvious that the proposed method is extremely effective in ensuring the integrity of the search results.

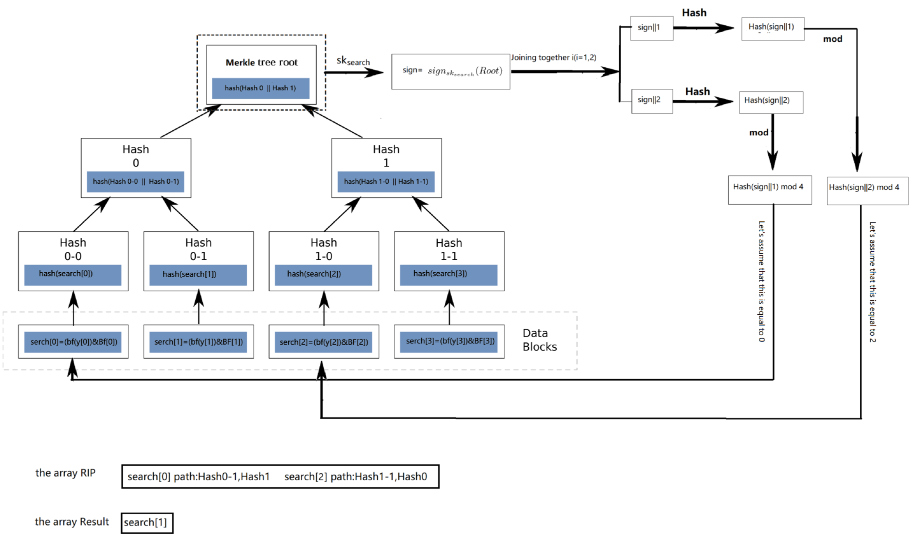

Fig. 4. Example of the proposed search result integrity verification of search results (here, assume four files to be searched and l to 2)

5 Smart Contract Based Searchable Symmetric Encryption

In this section, we propose the smart contract-based SSE scheme. The proposed scheme includes search result integrity verification algorithm, an index generation algorithms, a ciphertext search algorithm, and a demand smart contract. First, we describe the overall architecture of the proposed scheme. We then explain the proposed scheme in greater detail.

5.1 Scheme of Smart Contract Based SSE

Here, we introduce the smart contract-based SSE scheme and provide a security proof. There are three roles in the proposed scheme: file owner, searcher, and miner. The file owner has n documents denoted $D = (D_1, D_2, ..., D_n)$. The searcher's task is to search the corresponding document based on the keywords and indexes provided by the file owner. The miner's task is to verify that the search results are correct and pack the transactions into the blockchain. The owner of the file first inputs the keywords to search and provides commission d. When searcher finds searching demand then he searches the file according to the keywords and index directory, and searcher will submit search results and results integrity prove to the miners; if the verification through, miner will packaged the results and integrity proof to block chain.

Our scheme has two phases. In **series phase 1**, the file owner creates a smart contract for search demand. In **phase 2**, for a smart contract, the searcher can submit the results, and the miner can verify the results. The specific process is shown in Fig. 5, and we introduce our scheme below.

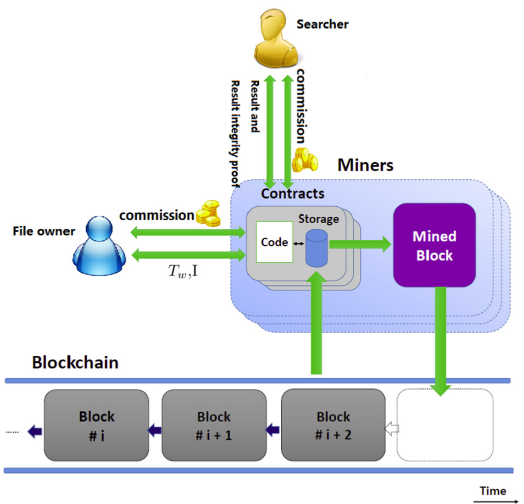

Fig. 5. Smart contract based SSE scheme

The proposed scheme involves two phases. In **phase 1**, the file owner creates a smart contract for the search demand. In **phase 2**, for a smart contract, the searcher can submit the results, and the miner can verify the results. The process is shown in Fig. 5.

In **phase 1**, the file owner wants to search keyword w, and he encrypts the keyword using the key $K_{trapdoor}$ to obtain the search trapdoor $T_w = (t_1 = f(w_i, k_1), \ldots, t_r = f(w_i, k_r))$ $(K_{trapdoor} = (k_1, k_2, ..., k_r))$, Then, the file owner

Algorithm 3. Demand smart contract

```
1: variable T_w, I, d, owner        \\trapdoor T_w index I = (C, BF), commissions d
2:
3: function DEMAND(T_w_input, I_input, d_input)
4:     T_w=T_w_input
5:     I = I_input
6:     d = d_input
7:     owner=msg.sender
8: end function
9:
10: function VERIFY(searcher, Result, RIP, sign(Root))
11:     if sign(R) is signed by searcher then
12:         if Result are correct then
13:             if the path of elements in the RIP is correct then
14:                 count=0
15:                 for i = 1; i <= l; i++ do
16:                     if search[H(sign||i)] == RIP[i] then
17:                         count++
18:                     end if
19:                 end for
20:                 if count==l then
21:                     return 1
22:                 end if
23:             end if
24:         end if
25:     end if
26:     return
27: end function
28:
29: function PAYCOMMISSIONS(searcher, Result, RIP, sign(Root))
30:     if Verify(searcher, Result, RIP, sign(R))==1 then
31:         Sent(owner, searcher, amount, Result)
32:         Selfdestruct()
33:     else
34:         return
35:     end if
36: end function
37:
38: function DESTROY
39:     if msg.sender == owner then
40:         Selfdestruct()
41:     else
42:         return
43:     end if
44: end function
```

creates a smart contract that contains trapdoor T_w, index $I = (C, BF)$ and commissions d.

Algorithm 3 presents a pseudocode framework for a file owner to publish a trapdoor that needs to be looked up. Here, when creating a contract, the file owner inputs trapdoor T_w, index $I = (C, BF)$, the corresponding commission d, and the verification function.

In **phase 2**, when the searcher sees the search demand, he begins to search trapdoor T_w. When the searcher finishes searching the documents, he calls function $Verify(\cdot)$ of the demand smart contracts and uses the results, results integrity proof as parameters. If the verification is conducted through smart contract, the commissions will be sent from the file owner to the searcher, and the demand smart contract calls the function $Destroy(\cdot)$ to destroy itself. Otherwise, authentication failure is returned.

If the file owner does not want to search the document, he calls the function $Destroy(\cdot)$ to destroy the smart contract.

Search keywords can be single or multiple words. For simplicity, we only consider the single keyword case. The proposed smart contract-based SSE scheme is defined as follows:

Definition 1 *(Smart Contract based Searchable Symmetric Encryption). A smart contract based SSE is a set of six polynomial time algorithms SSE = (Gen, Enc, Build, Buildtrapdoor, Search, Demand smart contract):*

- $K \leftarrow Gen(t, s, r)$: This probabilistic algorithm takes security parameters $t,\ s,\ r$ and outputs array key K.
- $C \leftarrow Enc(K_{file}, D)$: This algorithms takes file key K_{file}, data file collection D, and keyword collection W as input, and the file owner outputs encrypted files C.
- $I \leftarrow Build(C, W)$: This algorithms takes encrypt file C and keywords W as input, and the file owner outputs index I.
- $T_w \leftarrow Buildtrapdoor(K_{trapdoor}, w)$: This is the keyword trapdoor generation algorithm. The file owner takes search keyword w, and trapdoor key $K_{trapdoor}$ as input, and obtains trapdoor T_w as the output.
- $(result, RIP) \leftarrow Search(T_w, I)$: The searcher searches the encrypted files according to the file owner's requirements in the smart contract (keyword trapdoor T_w and index I). The search results are returned to the array Result and result integrity proof array RIP.
- Demand smart contract: This is created by the file owner and placed in the blockchain. The file owner initializes the contract and calls the constructor function Demand($\cdot$) to initialize the parameter trapdoor T_w, index $I = (C, BF)$, and commissions d parameters. The searcher calls the function PayCommissions($\cdot$) to submit the result and the RIP. The miner uses the function Verify($\cdot$) to verify the result and the RIP. The file owner calls the function Destroy($\cdot$) to destroy the smart contract.

5.2 Details of Smart Contract Based SSE Scheme

In the proposed smart contract-based SSE scheme, there are three participants, i.e., the file owner, the searcher, and the miner. The file owner has n files denoted $D = (D_1, D_2, ..., D_n)$ that need to be uploaded to the network. Prior to uploading, the file owner selects the secure function. Here, $f : \{0,1\}^* \times \{0,1\}^s \rightarrow \{0,1\}^s$ and $H_1 : \{0,1\}^* \times \{0,1\}^t \rightarrow \{0,1\}^t$ are secure pseudorandom functions. Note that $H : \{0,1\}^* \rightarrow \{0,1\}^t$ is a collision-resistant hash function, and $c = t+t+sr$ denotes the security parameter. The main steps of the algorithm are described as follows.

- **Gen(t, s, r)**. The data owner takes security parameters t, s, r as input, and it outputs secret key array $K = (K_{file}, K_{trapdoor}, K_{MAC})$, where $(K_{file}, K_{trapdoor}, K_{MAC}) \leftarrow \{0,1\}^{t+t+sr}$, $K_{file} \leftarrow \{0,1\}^t, K_{trapdoor} \leftarrow \{0,1\}^t, K_{MAC} \leftarrow \{0,1\}^{sr}$. K_{file} is used to encrypt the data documents, and $K_{trapdoor}$ is used to generate the search trapdoor T_w for the keyword w. K_{MAC} is used to generate a MAC for C_i.
- **Enc($\mathbf{K}_{file}$, D)**: There are two steps in this function. First, the data owner uses key K_{file} to encrypt the documents collection $D = (D_1, D_2, ..., D_n)$: $C_i = Enc_{K_{file}}(D_i)(1 \leq i \leq n)$, $MAC_{C_i} = H_1(K_{MAC}, C_i)$ then set $C \leftarrow ((C_1, MAC_{C_1}), (C_2, MAC_{C_2}), \ldots, (C_n, MAC_{C_n}))$. Then, the data owner generates an index to optimize the search time complexity for future searches. First, the data owner extracts keywords collection $W = (w_1, w_2, \ldots, w_r)$ from each D_i and obtains a total of nr keywords for n documents.
- **Build(C,W)**. Before the n documents are encrypted, the indexes are built as follows. Here, the input is document C comprising a unique identifier $C_{id} \in \{0,1\}^n$ (each encrypted file C_i has a unique identifier $C_{i_{id}}$), and a list of keywords $(w_1, w_2, \ldots, w_m) \in \{0,1\}^*$ (assume each file has m keywords). Here $K_{trapdoor} = (k_1, \ldots, k_r) \in \{0,1\}^{sr}$.
  - I. The following is performed for each document C_i in C.
    1. For each different keyword w_j for $j \in [0, m]$, perform the following.
       - (a) Compute the trapdoor of each keyword: $(t_1 = f(w_j, k_1), \ldots, t_r = f(w_j, k_r)) \in \{0,1\}^{sr}$.
       - (b) The corresponding ciphertext ID C_{id} information is added to get the codeword $e : e_1, \ldots, e_r$, where $(e_1 = f(C_{id}, t_1), \ldots, e_r = f(C_{id}, t_r)) \in \{0,1\}^{sr}$.
       - (c) codeword $e : e_1, \ldots, e_r$ is added to the file D_i Bloom filter BF_i.
    2. The output secure index of C is $I_C = (C_{id}, BF)$.
  - II. Output $I = \{I_{C_1}, \ldots, I_{C_n}\}$
- **Buildtrapdoor($K_{trapdoor}$, w)**: Given trapdoor key $K_{trapdoor} = (k_1, ..., k_r) \in \{0,1\}^{sr}$ and keyword w, $T_w = (f(w, k_1), ..., f(w, k_r)) \in \{0,1\}^{sr}$ is output as the trapdoor.
- **Search(T_w, I)**. When a searcher observes a demand, he wants to help search the file. Then, the searcher uses trapdoor $T_w = (t_1, \ldots, t_r) \in \{0,1\}^{sr}$ for keyword w to test each index I_{C_i} in indexed collections I.

1. The following is performed for C_i in C.
   a. Keyword w is encoded according to C_{id} to obtain e, where $e : (e_1 = f(C_{id}, t_1), \ldots, e_r = f(C_{id}, t_r)) \in \{0,1\}^{sr}$.
   b. To determine if each position in y that contains a 1 corresponds to a 1 in the Bloom filter, we must determine whether $(bf(e) \wedge BF) == bf(e)$ $(e = e_1, e_2, ..., e_r)$ is correct, and then add $(bf(y) \wedge BF_i)$ to the array search that $search_{H(sign||i)\ mod\ n}$ $(i = 1, 2, ..., l)$.
   c. If this is correct, C_i is added the array Result.
2. An array search is employed to build a Merkle tree and obtain the Merkle tree root Root.
3. The searcher uses their private key to sign the Merkle tree root Root and obtains $sign = sign_{PK_{searcher}}(Root)$.
4. Compute $H(sign||i)$ $(i = 1, 2, ..., l)$ and add $search_{H(sign||i)\ mod\ n}$ $(i = 1, 2, ..., l)$ and the Merkle tree path to the array RIP, where l denotes the security parameter.
5. Output Result, RIP, and sign(Root).

– **Demand smart contract**. This contract is described in Algorithm 3. This contract contains four functions: Demand(·), Verify(·), PayCommissions(·), and Destroy(·), which are described as follows.
  – **Demand(T_w, $I = (C, BF)$, d)**. This function is used by the file owner to initialize the smart contract. The file owner initializes trapdoor T_w, index $I = (C, BF)$, commissions d and the initial contract owner.
  – **Verify(searcher, Result, RIP, sign(Root))**. This function is used to verify the result and RIP.
    1. Verify that the signature of the Merkle tree root *Root* is signed by the searcher.
    2. Check that the results in the array *Result* are correct. If the results are correct, continues; otherwise, return error 0.
    3. Compute $search_{H(sign||i)\ mod\ n}$ $(i = 1, 2, ..., l)$ according to the root of the Merkle tree.
    4. Verify that the path given in array RIP of the Merkle tree of $search_{H(sign||i)\ mod\ n}$ $(i = 1, 2, ..., l)$ is correct. If this is correct, then continues; otherwise, return error 0.
    5. According to the compute process given in the SearchIndex function, the result of the array RIP is correct. If this is correct, return 1; otherwise, return error 0.
  – **PayCommissions(searcher, Result, RIP, sign(Root))**. This function is used by the searcher to submit the lookup result and result integrity proof. If the result of the submission is verified, the smart contract transfers the commission from the file owner's account to the searcher's account. Then, the smart contract is destroyed. Here, if validation fails, a validation error is returned.
  – **Destroy()**. This function is used by the file owner to cancel the demand and destroy the smart contract. The function first determines whether the calling function is from the file owner. If so, the destroy operation is performed; otherwise, a call error is returned.

6 Security and Performance Analyses

In this section, we apply a simulation paradigm [4] to define the security of the proposed scheme. We also present a security analysis and security proof. Finally, we analyze the performance of the proposed scheme.

6.1 Security Analysis

The proposed scheme can guarantee the correctness of search data and the integrity of search results. We apply SSE based on the smart contract, and file owners can publish demands through the smart contracts. When a searcher observes the file owner's the file search demand through smart contracts, he starts to search. When the searcher finishes searching, he sends the results to the smart contracts for accuracy and integrity verification. If verification is successful, the searcher obtains the commissions. When the searcher completes the search, a Merkle tree is constructed for each file's search results, and the root of the Merkle tree is signed with the searcher's private key. Then, according to the root of the signature and i, to compute a hash, modulo the number of the files n, $H(sign||i)\ mod\ n$, and then give the search result of the corresponding $search_{H(sign||i)\ mod\ n}$ $(i = 1, 2, ..., l)$ and the path of the result in the Merkle tree. The purpose of the root signature is to ensure that when the result is submitted, the miner will not tamper with the searcher while verifying the result, and that the commission will be sent to the searcher. When the results are submitted, the searcher needs to give the results integrity proof, that are a partial search result random given by hash value and the corresponding Merkle tree path, so our scheme can avoid searcher deliberately omitting the search results to save computational cost or other reasons. The fairness of the proposed scheme is based on the fact that the blockchain cannot be tampered with. Search files on smart contracts are encrypted; thus, when only a ciphertext is retrieved, the searcher cannot learn anything about the plaintext. The search keywords in the smart contract are also encrypted; thus, the server cannot view any information about the plaintext and keywords other than the search results when executing the search algorithm. The information uploaded to the blockchain and smart contract only includes the encrypted keywords (trapdoor T_w), encrypted ciphertext, and Bloom filter; thus, no one (searchers, miners, etc.) can obtain any information related to the keywords and search plaintext through block data and the smart contract. The smart contract-based SSE follows the above two properties; thus, it provides secure SSE.

We present a theorem to prove that the proposed scheme is secure. Here, we used a real/ideal simulation model to prove security, which is fully described in the paper [4]. A number of SSE-related papers [5–7] have used this security model to prove security.

Definition 2 *(IND-CKA2 Security). If a smart contract based SSE protocol is secure, the adversary should not distinguish the real game* $Real^{\Pi}_{\mathcal{A}}(c)$ *and the simulation game* $Ideal^{\Pi}_{\mathcal{A},\mathcal{B},\mathcal{S}}(c)$*.*

Here, let Π be a smart contract based SSE scheme, where $\Pi = (Gen, Enc, Build, Buildtrapdoor,\ Search, Demand\ smart\ contract)$, and $c = t + t + sr$ is the security parameter. $\mathcal{A}$ represents the attacker, $\mathcal{B}$ represents the environment that can simulate the Ethereum system, $\mathcal{C}$ represents the challenger, and $\mathcal{S}$ represents the simulator.

$Real_A^{\Pi}(c)$: Challenger $\mathcal{S}$ can obtain key array K by performing operation $K \leftarrow Gen(j, s, r)$. After $\mathcal{S}$ obtains K, he will give the signature to be verified by the public key to Adversary $\mathcal{A}$. $\mathcal{A}$ gives challenger $\mathcal{C}$ a set of randomly selected files $D = (D_1, D_2, ..., D_n)$. Then, challenger $\mathcal{C}$ can obtain (I, C) by performing operation $Build(Enc(K, D), W)$. After challenger $\mathcal{C}$ obtains (I, C), they send (I, C) to $\mathcal{A}$. Challenger $\mathcal{C}$ returns (commissions, T_w) to attacker $\mathcal{A}$ to perform a polynomial query based on the different keyword w_i selected by the attacker $\mathcal{A}$. Then, attacker $\mathcal{A}$ queries Result and RIP by asking the $\mathcal{C}$. Finally, $\mathcal{A}$ produces a bit b that is returned by the experiment.

$Ideal_A^{\Pi}(c)$: Attacker $\mathcal{A}$ randomly selects D to satisfy condition $|D_{Ideal}| = |D_{Real}|$, where $D \leftarrow \{0, 1\}^*$. Here, simulator $\mathcal{S}$ obtains (I, C) by performing operation $\leftarrow S(L(D))$ ($L(D)$ is defined in [3]). When the simulator calculates (I, C), it sends (I, C) to attacker $\mathcal{A}$. The challenger $\mathcal{C}$ returns Result and RIP to attacker $\mathcal{A}$ to perform a polynomial query in the $\mathcal{B}$ environment based on the different keyword w_i selected by attacker $\mathcal{A}$. Finally, $\mathcal{A}$ produces a bit b that is returned by the experiment.

If we can find a PPT $\mathcal{S}$ for all PPT $\mathcal{A}$, that makes $\mathcal{D}$,
$|Pr[\mathcal{D}(view_{real}) = 1] - Pr[\mathcal{D}(view_{ideal}) = 1]| \leq negl(c)$ distinguishable for all polynomial sizes. Then, we can prove that Π is IND-CKA2 secure.

Theorem 1. *If the proposed SSE based smart contract scheme is an adaptive IND-CKA2 scheme, then the conditions to be met are H_1, f should be a pseudorandom function, H should be an anti-collision hash function, and $\varepsilon = (Enc, Dec)$ should be an IND-CPA [4] symmetric encryption scheme.*

Proof. We can build a PPT simulator $\mathcal{S} = \mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_q$ for attackers, which allows $\mathcal{A} = \mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_q$ to output two results $View_{real}$ and $View_{ideal}$. However these two results are indistinguishable during computation. Here, through the trace of a given history L, simulator $\mathcal{S}$ can be generated (I^*, C^*, t_w, Result, RIP) using the following methods.

- First, we discuss simulation I^*. Here, if $q = 0$, all files sizes are available to the simulator from L. $\mathcal{S}$ could set I^* to a random string of size $|I|$ by taking advantage of that information. $\mathcal{S}$ sets $C_i^* \leftarrow \{0, 1\}^{|D_i|}$ first, and then sets the state of I^* to $st_{\mathcal{S}}$. Note that state $st_{\mathcal{A}_0}$ has no key value for $K_{trapdoor}$; thus, $\mathcal{S}$ will do a uniform random selection of w for each keyword.
  If $q \geq 1$, attacker $st_{\mathcal{A}}$ first select $(st_{\mathcal{A}_0}, t^*_{w^*_0}) \leftarrow \{0, 1\}^*$ and $Mac_{w^*_0}{}^* \leftarrow \{0, 1\}^*$ randomly and evenly for each keyword w^*_0. Then, $\mathcal{S}$ select $w^*_i (q \geq i \geq 1)$ based on $(w^*_{i-1}, st_{\mathcal{A}_{i-1}})$. $\mathcal{S}$ then randomly selects $(st_{\mathcal{A}_i}, t^*_{w^*_i}) \leftarrow \{0, 1\}^*$ and $Mac_{w^*_i}{}^*$ for each keyword w^*_i.
  We know that I^* and I are indistinguishable because H_1 and $\{0, 1\}^* \times \{0, 1\}^t$ are indistinguishable. In the same manner, $\varepsilon = (Enc, Dec)$ is IND-CPA; thus,

we can also know that C^* is indistinguishable from the real ciphertext c. In the absence of key K_{MAC} in state st_{A_0}, there is no way for an attacker to distinguish between the $MAC_{C_i}{}^*$ and MAC_{C_i} generated in the $Enc(\cdot)$ step. This $MAC_{C_i}{}^*$ is selected randomly and uniformly from $\{0,1\}^t$.

- Now, we simulate T_w^*. If t_w^*, K_{tonke}^* and $t_w, K_{trapdoor}$ (generated by function $Build(\cdot)$) are indistinguishable, the conclusion is good. If $q = 0$, $\mathcal{S}$ will randomly and evenly select t_w^*, K_{tonke}^* from $\{0,1\}^k$. If $q \geq 1$, calculate $(t_{w_i^*}^*, K_{trapdoor}^*) \leftarrow \{0,1\}^*$ and select $w_i^*, q \geq i \geq 1$ based on $st_{\mathcal{A}_{i-1}}, w_{i-1}^*, i \geq 1$. Here, the pseudorandom functions H_1 and $\{0,1\}^* \times \{0,1\}^t$ are indistinguishable; thus, we can say that T_w^* and true T_w are indistinguishable.
- Simulate RIP^*. The pseudorandom function f is indistinguishable; thus, e and e^* generated in the search step are indistinguishable. The anti-collision hash function h is indistinguishable, the resulting RIP and the simulated RIP^* are indistinguishable.

Definition 3. *If the scheme is fair to both searchers and file owners, we say that the scheme satisfies fairness.*

Theorem 2. *If the blockchain is irreversible and the smart contract runs in the blockchain, the proposed scheme will satisfy fairness.*

- When the search task is generated, the commission is stored in the smart contract account. If the searcher fails to find the correct result, they will not obtain the commission. When the searcher finds the correct result, the file owner cannot refuse to pay the commission, and the smart contract will automatically transfer the commission to the searcher.
- The smart contract runs in the blockchain; thus, the file owner cannot change the task after publishing the search task, and the searcher cannot change the correct search result after receiving the commission.
- If both parties execute the agreement honestly, the user can obtain the correct results from the smart contract, and the searcher can obtain a commission.

Definition 4. *If we can verify whether the search results are complete and without omission, we say that the SSE scheme satisfies result integrity verification.*

Theorem 3. *If the smart contract can correctly execute our result integrity proof, then the scheme satisfies result integrity verifiable.*

Proof. After the searcher submits the results, the smart contract will implement the result integrity verification algorithm (Sect. 4.2). If the result is correct and there is no omission, the smart contract will pass verification. If the submitted result is correct but missing, it cannot pass the result integrity verification of the smart contract. □

6.2 Performance Analysis

Here, we analyze the efficiency and security of the proposed scheme and compare it to related schemes.

For users, search time determines the practicability and feasibility of the SSE scheme; thus, we primarily consider calculation and communication costs during the search stage. Table 2 shows how the proposed scheme performs in the search phase compared to related schemes. As shown in Table 2, the search time of the scheme proposed Kamara et al. [5] is the fastest because this scheme is designed to run parallel. However, this scheme does not satisfy fairness does not provide an integrity proof of the result. Thus, if the searcher returns incorrect results or if the searcher disappears after receiving a commission from the user, the user will lose both money and the search results. Therein lies the unfairness. To achieve fairness, another scheme [7] requires at least six transactions and three communications, which may prolong the time required for users to obtain results. The proposed scheme only requires two transactions and two rounds communication.

Table 2. Comparison of results of different schemes

Different scheme	Search time complexity	Communication complexity	Verifiable	Attacker	Fairness	Result integrity verifiable
Parallel [5]	O(mlog(n))	1	Yes	Server is honest-but-curious	No	No
Uc-secure [6]	O(n)	m	Yes	Server is malicious	No	No
BC [7]	O(n)	6	Yes	Server is malicious and user is malicious	Yes	No
Our scheme	O(n)	2	Yes	Server is malicious and user is malicious	Yes	Yes

6.3 Experimental Analysis

We develop an encrypted file search system. Here, we conduct relevant experiments on the number of different files and number of keywords, and we compare the time of completeness of the generated results and search time. The experimental data are shown in Table 3.

Table 3. Time required to search for encrypted files and obtain the result integrity verification

Number of documents	Number of keywords	Search time(s)	Result integrity verification(s)
2000	4	3.745	3.747
4000	4	3.912	3.902
6000	4	4.049	4.053
8000	4	4.282	4.286
10000	4	4.459	4.467
12000	4	4.576	4.571
2000	6	3.754	3.736
4000	6	3.909	3.913
6000	6	4.045	4.049
8000	6	4.283	4.287
10000	6	4.461	4.468
12000	6	4.567	4.573
2000	8	3.761	3.765
4000	8	3.908	3.917
6000	8	4.054	4.053
8000	8	4.291	4.292
10000	8	4.473	4.474
12000	8	4.577	4.581
2000	10	3.777	3.781
4000	10	3.931	3.928
6000	10	4.074	4.069
8000	10	4.296	4.293
10000	10	4.485	4.481
12000	10	4.596	4.594

From the experimental data, we conclude that, after adding the result integrity verification, the overall time is doubled, which is still within the acceptable range; thus, we consider that the proposed system provides sufficient availability.

We run our smart contract on the test network of Ethereum. Running our smart contract will burn 72000 gas per time on average, while the normal transfer on Ethereum test network will burn 20000 gas; therefore, our scheme is practical.

7 Conclusion

In this paper, we have proposed smart contract based SSE. Existing SSE protocols can resist malicious servers using MAC algorithms, but only if the server is actively running. If the server receives a user's money but does not provide service to the user or the server is closed after receiving the user's money, the user cannot withdraw the money. In addition, if the server wants to reduce computational costs and bandwidth, it will reduce the number of documents to be searched and omit a part of the search results; thus, there is no guarantee that all files will be searched. The proposed scheme solves this problem effectively using an integrity proof of search results and a smart contract. The proposed scheme guarantees the authenticity and integrity of the search results, and through the signature of the Merkle tree root that the searcher who searches the results will not be tampering by the miners after the submission of the results.

Acknowledgments. We would like to thank the anonymous reviewers. This work is supported by The National Key Research and Development Program of China (Grant No. 2018YFA0704701), National Natural Science Foundation of China (Grant No. 62072270), National Cryptography Development Fund (Grant No. MMJJ20170121), and Shandong Province Key Research and Development Project (Grant Nos. 2020ZLYS09 and 2019JZZY010133).

References

1. Androulaki, E., Karame, G., Roeschlin, M., Scherer, T., Capkun, S.: Evaluating User Privacy in Bitcoin. IACR Cryptology ePrint Archive 2012:596 (2012)
2. Buchbinder, N., Petrank, E.: Lower and upper bounds on obtaining history independence. Inf. Comput. **204**(2), 291–337 (2006)
3. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. J. Comput. Secur. **19**(5), 895–934 (2011)
4. Goh, E.-J.: Secure Indexes. IACR Cryptology ePrint Archive 2003:216 (2003)
5. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258–274. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_22
6. Kurosawa, K., Ohtaki, Y.: UC-secure searchable symmetric encryption. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 285–298. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32946-3_21
7. Li, H., Tian, H., Zhang, F., He, J.: Blockchain-based searchable symmetric encryption scheme. Comput. Electr. Eng. **73**, 32–45 (2019)
8. Li, H., Zhang, F., He, J., Tian, H.: A searchable symmetric encryption scheme using blockchain. arXiv preprint arXiv:1711.01030 (2017)
9. McIlroy, M.: Development of a spelling list. IEEE Trans. Commun. **30**(1), 91–99 (1982)
10. Meitinger, T.H.: Smart contracts. Informatik-Spektrum **40**(4), 371–375 (2017). https://doi.org/10.1007/s00287-017-1045-2
11. Mullin, J.K., Margoliash, D.J.: A tale of three spelling checkers. Softw. Pract. Exper. **20**(6), 625–630 (1990)

12. Naor, M., Teague, V.: Anti-persistence: history independent data structures. In: ACM Symposium on Theory of Computing, pp. 492–501 (2001)
13. Roy, S.S., Karmakar, A., Verbauwhede, I.: Ring-LWE: applications to cryptography and their efficient realization. In: Carlet, C., Hasan, M.A., Saraswat, V. (eds.) SPACE 2016. LNCS, vol. 10076, pp. 323–331. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49445-6_18
14. Szabo, N.: Smart contracts: building blocks for digital markets. EXTROPY J. Transhumanist Thought **18**(16), 2 (1996)
15. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997)
16. Wan, Z., Cai, M., Lin, X., Yang, J.: Blockchain federation for complex distributed applications. In: Joshi, J., Nepal, S., Zhang, Q., Zhang, L.-J. (eds.) ICBC 2019. LNCS, vol. 11521, pp. 112–125. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23404-1_8
17. Zhang, C., Fu, S., Ao, W.: A blockchain based searchable encryption scheme for multiple cloud storage. In: Vaidya, J., Zhang, X., Li, J. (eds.) CSS 2019. LNCS, vol. 11982, pp. 585–600. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-37337-5_48
18. Zhang, Y., Rajimwale, A., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: End-to-end data integrity for file systems: a ZFS case study. In: 8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, 23–26 February 2010, pp. 29–42. USENIX (2010)