Practical, Dynamic and Efficient Integrity Verification for Symmetric Searchable Encryption

Lanxiang Chen(✉) and Zhenchao Chen

College of Mathematics and Informatics, Fujian Normal University,
Fujian Provincial Key Laboratory of Network Security and Cryptology,
Fuzhou, China
lxiangchen@fjnu.edu.cn

Abstract. Symmetric searchable encryption (SSE) has been proposed for years and widely used in cloud storage. It enables individuals and enterprises to outsource their encrypted personal data to cloud server and achieves efficient search. Currently most SSE schemes are working in the semi-honest or curious cloud server model in which the search results are not absolutely trustworthy. Thus, the verifiable SSE (VSSE) schemes are proposed to enable data integrity verification. However, the majority of existing VSSE schemes have their own limitations, such as not supporting dynamics (data updates), working in single-user mode or not practical etc. In this paper, we propose a practical, dynamic and efficient integrity verification method for SSE construction that is decoupled from original SSE schemes. This paper proposed a practical and general SSE (PGSSE for short) scheme to achieve more efficient data integrity verification in comparison with the state-of-the-art schemes. The proposed PGSSE can be applied to any top-k ranked SSE scheme to achieve integrity verification and efficient data updates. Security analysis and experimental results demonstrate that the proposed PGSSE scheme is secure and efficient.

Keywords: Symmetric searchable encryption · Shamir's secret sharing · Merkle Patricia Tree · Integrity verification · Data update

1 Introduction

Accompanying with the explosive growth of cloud computing, data security has become the most important aspect that needs to be guaranteed. To preserve data security and privacy, individuals and enterprises usually encrypt their data before outsourcing them to the remote cloud computing or cloud storage server.

This work was supported by the National Natural Science Foundation of China under Grant No. 61602118, No. 61572010 and No. U1805263, Natural Science Foundation of Fujian Province under Grant No. 2019J01274 and No. 2017J01738.

Y. Mu et al. (Eds.): CANS 2019, LNCS 11829, pp. 163–183, 2019.
https://doi.org/10.1007/978-3-030-31578-8_9

As a result, how to process and search on the encrypted data becomes the critical problem that needs to be resolved. Searchable encryption facilitates search operations on the encrypted data meanwhile it preserves the privacy of users' sensitive data, and thus it has attracted extensive attentions recently.

Since the first searchable encryption scheme [23] proposed in 2000, a large number of related works have emerged in the literature during the last two decades. According to the encryption algorithm adopted in searchable encryption schemes, they can be classified into public key searchable encryption (PKSE) and symmetric searchable encryption (SSE). As there are massive data in a cloud storage, the data usually are encrypted by a symmetric encryption algorithm to guarantee efficiency and data availability.

For a practical SSE scheme, it should satisfy at least the following properties: sublinear search time, compact indexes, supporting ranked search, efficient updates, integrity verification and data security. Unfortunately, none of the existing SSE constructions achieves all these properties at the same time, which has limited their practicability. If an SSE scheme does not support top-k ranked search, the cloud server will return all data files that contain the queried keywords. As the user have no pre-knowledge of the encrypted files, he has to decrypt all these files to further find the most matching files. These will result in unnecessary computing overheads, time consumption and network traffics. Hence, without it, SSE schemes are impractical in the pay-as-you-use cloud computing era. By returning the most related files, ranked search schemes greatly facilitate system practicability.

To enrich the functionalities of SSE, a variety of multi-keyword, multi-user or multiple data owner, dynamic or verifiable SSE schemes have been proposed. However, majority of existing SSE schemes have their own ways of index construction, integrity verification and data updates. A general scheme with more functionalities decoupling from any special constructions is lacking. Motivated by this idea, in this paper, we propose a practical, dynamic and efficient integrity verification method for SSE construction that is decoupled from original SSE schemes. Our work is a one-step forward to the work due to Zhu et al. [31] in terms of top-k ranked search and data update efficiency. The contributions of this work can be summarized as follows.

- We proposed a practical and general integrity verification scheme (PGSSE) with the aid of secret sharing scheme and Merkle Patricia Tree (MPT). Compared with existing SSE schemes, the proposed scheme firstly introduces the secret sharing scheme to SSE to make the general SSE scheme support top-k ranked search.
- Thanks to the secret sharing scheme, users do not need to update the MPT tree but just to update their keys when they update their data without keyword addition. Thus the data updates of the proposed scheme are very efficient.

This paper is arranged as follows. We will discuss related work in Sect. 2. Section 3 gives the preliminaries. The system model and formal definition are

presented in Sect. 4. Section 5 describes the details of our PGSSE scheme construction. Section 6 present security and performance evaluation of the proposed scheme. We give a conclusion in Sect. 7.

2 Related Work

In 2000, Song et al. [23] proposed the first searchable encryption scheme which needs to search all encrypted documents in a non-interactive way to check whether a queried keyword is contained or not. For each queried keyword, it has to scan all files and thus the search time was linear in the length of the documents collection. In addition, the proposed scheme is only adapted to single keyword search. In 2003, Goh [11] first proposed to construct index to achieve search and a Bloom filter based index scheme is introduced. He gave a formal security of IND-CKA for SSE and proved the proposed Bloom filter based SSE scheme is IND-CKA secure. The drawback is that the Bloom filters based construction had a possibility of false positives. In 2006, Curtmola et al. [8] proposed two efficient SSE schemes, SSE-1 and SSE-2 with $O(1)$ search time complexity. They gave the formal security definitions for the proposed schemes and utilized broadcast encryption to enable multi-user search in SSE-2. Both schemes only support single-keyword search.

After that, a variety of functionally rich SSE schemes were proposed in the last two decades, including multi-keyword search, top-k ranked search, dynamic data update, verifiable SSE, fuzzy and similarity search etc. As described above, if an SSE scheme does not support top-k ranked search, the cloud server will return all data files containing the queried keywords, which will greatly reduce the practicability of these schemes.

In 2010, Wang et al. [26,27] first proposed to use order-preserving symmetric encryption (OPSE) to achieve a ranked keyword search scheme which protects the privacy of relevance scores. The measure of relevance scores is based on a TF×IDF model. To reduce the amount of information leakage, they proposed to use a one-to-many OPSE scheme to obfuscate the original relevance score distribution. In 2011, Cao et al. [1,2] proposed a privacy-preserving multi-keyword ranked searchable encryption (MRSE) scheme by using "coordinate matching" and "inner product similarity".

To improve the accuracy of search results, almost all multi-keyword SSE schemes [5,9,18,29,30] support top-k ranked search since the ranked SSE scheme has been proposed. However, the SSE constructions described above are static, which means that they did not have the ability to add or delete documents efficiently.

In 2010, Liesdonk et al. [19] proposed the first dynamic SSE scheme which supports a limited number of updates and has a linear search time in the worst case. In 2012, Kamara et al. [15] constructed a dynamic SSE scheme which is an extension of SSE-1. They presented a formal security definition for dynamic SSE. Their scheme is adaptively secure against chosen-keyword attacks (CKA2) and it is also secure in the random oracle model. Then, in 2013, they presented

a parallelizable and dynamic sub-linear SSE scheme with the help of multi-core architectures [14]. The search time is about $O(r/p)$(r and p is the number of documents and cores respectively) for searching a keyword with a logarithmic number of cores. Compared to the SSE scheme in [15], this SSE scheme does not leak the tokens of the keywords contained in an updated document. Their scheme uses a keyword red-black tree (KRB) to construct index that makes updates simple. However, this scheme focuses on the case of single-keyword equality queries only. Since then, some dynamic SSE schemes are presented [3,6,10,12, 21,28].

As the cloud server is not trustable in some circumstances, verifiable SSE (VSSE) [4,7,13,16,17,20,24,25] is proposed to check the integrity of search results and data. In 2012, Kurosawa and Ohtaki [16] first formulate the security of VSSE against active adversaries and proposed a UC-security (abbreviation of Universal Composability) single-keyword VSSE scheme. Their scheme preserves the search results correct even if the server is malicious. Later in 2013, they gave a more efficient VSSE scheme [17] and extended the scheme to dynamic VSSE scheme. In 2015, Sun et al. [24] proposed a dynamic conjunctive keyword VSSE scheme by using bilinear-map accumulator tree. Recently, Jiang et al. [13] proposed a multi-keyword ranked VSSE scheme and a special data structure QSet based on an inverted index. The basic idea is to estimate the least frequent keywords in the query to reduce the search times. The verification is based on a keyword binary vector.

However, all the above works have their own ways of index construction, integrity verification and data updates. A general scheme with more functionalities decoupling from any special constructions is lacking. Recently, Zhu et al. [31] proposed a generic and verifiable SSE scheme (GSSE). It can be adopted to any SSE scheme to provide integrity verification and data updates. They proposed to use the Merkle Patricia Tree (MPT) and incremental hash to construct proof index and develop a timestamp chain to resist data freshness attacks. As MPT is a kind of prefix tree, it is efficient to insert and delete nodes. Hence the GSSE achieves data integrity verification efficiently.

But there is a shortcoming in the proposed GSSE scheme that the user has to get all documents which contain the queried keyword in the document collection to perform the integrity verification. If the queried keyword is common, there could be quite a lot of documents containing the keyword. Many documents returned may be not desired by the user while they will consume a lot of time to search and verify. It also means that the GSSE scheme does not support top-k ranked search.

Comparing to the GSSE scheme, the proposed PGSSE scheme supports top-k ranked search which makes it more practical. As the incremental hash is utilized for integrity verification in GSSE, users have to compute the hash of all queried documents to verify the root of the MPT tree. Thus, the GSSE scheme cannot support top-k ranked search. To overcome this disadvantage, we propose to utilize a secret sharing scheme to replace incremental hash to perform integrity verification. Comparing to GSSE, the PGSSE allows users to perform integrity

verification when they only get the top-k documents containing the queried keywords. Meanwhile, we found that the proposed scheme can achieve data updates efficiently.

3 Preliminary

In this section, the notations, MPT and Shamir's secret sharing scheme are revisited.

3.1 Notations

In the following sections, the pseudo-random functions h_1, h_2 and h_3 are defined as $\{0,1\}^* \times \{0,1\}^\lambda \rightarrow \{0,1\}^*$. The other notations are described in Table 1.

Table 1. Notations and descriptions

Symbol	Denote
DC	The document collection, including N documents and denoted as $DC = (D_1, D_2, \cdots, D_N)$
C	The encrypted document collection $C = (C_1, C_2, \cdots, C_N)$ stored in the cloud server
I	The encrypted index
W	The keyword dictionary $W = (w_1, w_2, \cdots, w_m)$
m	The number of keywords in W
$DC(w_i)$	The collection of documents that contain keyword w_i
WD_i	The keyword set in the document D_i
Au	The authenticator
$Enc/Dec(\cdot)$	Symmetric encryption/decryption algorithm
Key	The secret key stored by the data owner $Key = \{k_1, k_2, k_3, k_4, (spk, ssk), S, P\}$

For the secret key stored by the data owner, k_1, k_2 and k_3 are used for pseudo-random functions h_1, h_2 and h_3 respectively, k_4 is used for the symmetric encryption algorithm $Enc()$, (spk, ssk) is the public/private key pair, S is a matrix in which each row represents one polynomial's coefficients, P is a set of m arrays.

3.2 Merkle Patricia Tree

Merkle Patricia Tree (MPT) proposed in Ethereum is a mixture of Merkle tree and Patricia tree. It is a kind of prefix tree that has high efficiency in insert and delete operations. There are four kinds of nodes in the MPT, namely null node, leaf node, extension node and branch node. The null node is simple and we use a blank string to represent it. Leaf node (LN) and extension node (EN) are both represented as one key-value pair and those keys are encoded in Hex-Prefix. The keys in extension nodes indicate their descendant nodes' common prefix and their values are their children nodes' hash values. The keys in leaf nodes indicate the rest part except for the common prefix and the values are their own

values. Differing from LN and EN, the branch nodes' keys consist of 17 elements in which 16 elements correspond to Hex-Prefix codes. The last element is used only when the search route terminates here in which the value in BN plays the same role as that in LN.

The construction of a MPT is through the "insert" operation which will be demonstrated according to different situations.

(1) **Insert to branch node (the current key is empty)**

The initial MPT is empty as Fig. 1(a) and a new node with value '223' will be inserted to the MPT. The insert operation directly set the value of the MPT to '223' and get the MPT as Fig. 1(b).

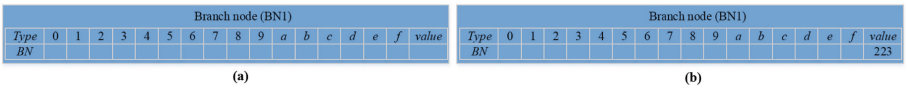

Fig. 1. Insert to branch node (the current key is empty).

(2) **Insert to branch node (the current key is not empty)**

Assume the initial MPT is Fig. 2(a) and a new node with key-value pair ['a2912', '22'] will be inserted to the branch node. As the descendant of the element 'a' is empty, the "insert" algorithm will create a new leaf node (LN2) to store the rest key '2912' and the value '22'. The new MPT is illustrated as Fig. 2(b).

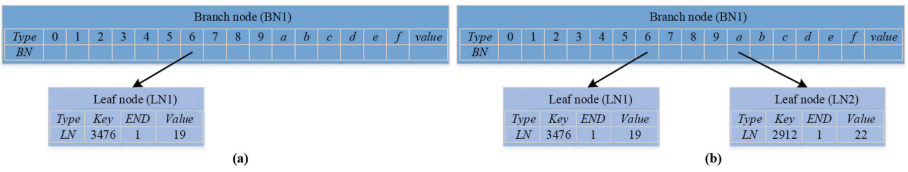

Fig. 2. Insert to branch node (the current key is not empty).

(3) **Insert to Extension node**

Assume the initial MPT is Fig. 2(b) and a new node with key-value pair ['a2535', '57'] will be inserted to the MPT. As the key 'a2535' has a common prefix 'a2' with LN2, the "insert" algorithm will create an extension node (EN1) whose key is the rest common prefix '2' and a branch node (BN2) whose key '5' and key '9' are linked to newly created leaf nodes with key '35' and '12' respectively. The insertion is completed as Fig. 3.

When searching for a node in the MPT, the "Search" algorithm will start from the root to bottom to check the nodes' key at each level. For example, the user wants to search the node with key 'a2535'. The "search" algorithm will first find the BN1 and then go on to EN1, and finally the path from BN1 to LN2 will be found.

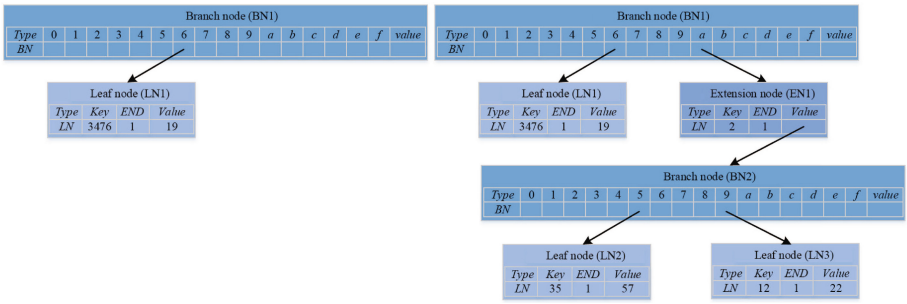

Fig. 3. Insert to extension node.

3.3 Shamir's Secret Sharing Scheme

As pointed out that the incremental hash needs to compute the hash of all queried documents to check the integrity of search results, it is unsuitable for ranked search. With a secret sharing scheme, the pre-defined number of participants can compute the secret without the involvement of all participants. This property can be applied to the ranked search. For the sake of generality, the Shamir's secret sharing scheme is chosen in the proposed PGSSE scheme.

Shamir's secret sharing scheme [22] is a threshold scheme to share a secret in a distributed way. For a (k, n) threshold scheme, a secret is split into n pieces for n participants and any more than $k-1$ participants can reconstruct the secret (k is the threshold), but the secret cannot be reconstructed with fewer than k pieces. With the feature of dynamics, it can be applied to the ranked search with efficient data updates.

With the dynamics of Shamir's secret sharing scheme, the security can be easily enhanced without changing the secret, and only need to change the polynomial coefficients and construct new shares to the participants.

To construct a (k, n) Shamir's secret sharing scheme, it needs to construct a k-1 degree polynomial $f(x)$ in the finite field GF(q) and the polynomial's constant term is the secret s, where q is a big prime number ($q > n$). Firstly, it randomly generates a k-1 degree polynomial based on GF(q) and set $f(0) = a0 = s$. Then, it randomly selects n different non-zero numbers $(x_1, x_2, \ldots, x_n)$ and allocates $(x_i, f(x_i))$ to each participant $p_i (0 < i < n)$, where x_i is public and $f(x_i)$ is kept secret.

Then to recover the secret s, it randomly selects k pairs $(x_j, f(x_j))(0 < j < k)$ and utilizes the Lagrange's polynomial interpolation algorithm as Eq. (1) to reconstruct the secret as Eq. (2).

$$f(x) = \sum_{j=1}^{k} f(x_j) \prod_{l=1}^{k} \frac{x - x_l}{x_j - x_l} \bmod q \tag{1}$$

$$s = (-1)^{k-1} \sum_{j=1}^{k} f(x_j) \prod_{l=1}^{k} \frac{x_l}{x_j - x_l} \bmod q \tag{2}$$

4 System Model and Formal Definition

The system model and formal definition are described in this section.

4.1 System Model

The system model is illustrated in Fig. 4. There are three entities, namely data owner, data user and cloud server. Data owner is in charge of constructing index and authenticator. He receives the request from data user and authenticates the data user. After being authenticated, the data user can access cloud server to obtain some search results and he will perform an integrity verification for the search results and the corresponding document data. The cloud server is responsible for storing users' indexes, authenticator and document data. When receiving the token from a data user, the cloud server will make the corresponding proof and authenticator to the data user.

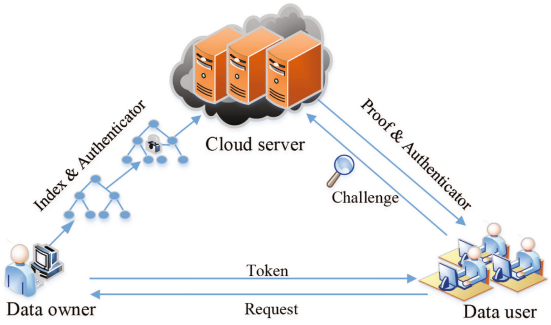

Fig. 4. System model.

4.2 Formal Definition

The proposed PGSSE scheme has seven polynomial-time algorithms.

(1) $Setup(1^{\lambda}) \rightarrow Key$**:** It is run by the data owner to setup the scheme. The algorithm takes as input the security parameter λ and outputs the secret Key.

(2) $MPTBuild(Key, W, DC) \rightarrow \{I, Au\}$**:** It is run by the data owner. It takes as input the Key and keyword dictionary W, and outputs the index and authenticator.

(3) $TokenGen(k_3, Q) \rightarrow \{Token\}$**:** It is run by the data user. It takes as input k_3 and queried keywords, and outputs the token.
(4) $ProofBuild(I, Token, t_q) \rightarrow \{Proof, Au_q^t, Au_c\}$**:** It is run by the cloud server. It takes as input index I, the token and the query time t_q, and outputs the corresponding proof and two authenticators Au_q^t and Au_c.
(5) $CheckAu(k_4, Au_q^t, Au_c) \rightarrow \{result\}$**:** It is run by the data user. It takes as input the key k_4 and two authenticators Au_q^t, Au_c, and outputs a result to indicate whether the root of MPT has been tampered.
(6) $Verify(k_2, k_4, S, P, C_Q, Proof, Token_Q, Au_q^t) \rightarrow \{result\}$**:** It is run by the data user. It takes as input the k_2 and k_4, the search result C, the authenticator Au_q^t and P, and outputs a result to indicate whether the queried documents have been tampered.
(7) $Update(P, D_j, I) \rightarrow \{P', I'\}$**:** It is run by the data owner. It takes as input the set P, update document D_j and index I, and outputs the new P' and new I'.

5 Scheme Construction

In this section, the seven polynomial-time algorithms of the proposed PGSSE are detailed respectively. The authenticator of the "CheckAu" algorithm is used to make this scheme to resist the freshness attack on the root of MPT. As it is the same as the scheme in [31] and we will not elaborate on it here.

5.1 Initialization

The "Setup" algorithm will initiate the system parameters and generate all keys. It is executed by data owner and the detailed process is illustrated in Algorithm 1. There are m polynomials and each keyword corresponds to a polynomial. Meanwhile each keyword corresponds to the secret S_{w_i} of the polynomial which is also called node secret. All the node secrets are stored in the MPT. When a data user receives the top-k documents, he/she would try to recover the node secret and execute the integrity verification. The set of P consists of m arrays and each array is in form of ['key' $\rightarrow$ '$value$']. This array could help user recover the node secret.

5.2 MPT Building

The MPT building algorithm is performed by the data owner and the detailed procedure is illustrated in Algorithm 2. The index I is the MPT and the "insert" algorithm can refer to Sect. 3.2. When $|DC(w_i)| \geq k$, the node secret will be computed and inserted into MPT, and there will be more than k key-value pairs and the node secret can be recovered through the secret sharing scheme. However, when $|DC(w_i)| < k$, there are less than k key-value pairs in the keyword

Algorithm 1. Setup Algorithm

Input: Parameter λ;
Output: $Key = \{k_1, k_2, k_3, k_4, (spk, ssk), S, P\}$.

Randomly generates k_1, k_2, k_3, k_4 and q;
Generates the public/private key pair (spk, ssk);
Compute node secret:
for $w_i \in W$ **do**
  compute $S_{w_i} = h_1(k_1, w_i)$;
end
Generate the matrix $S : S$ is an $m \times k$ matrix, the i-th row is $\{a_{i_1}, a_{i_2}, \cdots, a_{i,k-1}, S_{w_i}\}(i \in [1, m])$ where $\{a_{i_1}, a_{i_2}, \cdots, a_{i,k-1}\}$ are the coefficients of the Shamir's secret sharing polynomials $f_i(x), S_{w_i}$ is the secret of $f_i(x)$;
Generate the set P which consists of m arrays. Each array corresponds to a keyword and the keyword w_i's array $(array_i)$ is generated as follows:
for $w_i \in W$ **do**
  for $D_j \in DC(w_i)$ **do**
    Calculate $key_{D_j} = h_2(k_2, D_j)$;
    Encrypt the key_{D_j} with k_4 and $vx_{D_j} = Enc(k_4, key_{D_j})$;
    Extract the w_i's coefficients in S and build the k-1 degree polynomials $f_i(x)$;
    Calculate $value = f_i(key_{D_j})$;
    Set the $array_i[vx_{D_j}] = value$;
  end
end

arrays and the node secret cannot be reconstructed by the secret sharing scheme. Hence, the sum of document hash values that can be used to check the integrity of all returned documents is calculated in this situation.

The authenticator Au is used to ensure the freshness of the MPT's root rt that is proposed in [31] and it is generated in Eq. (3), where tp is the timestamp, up_i is the i-th update time point, $Sig(ssk, *)$ is a signature with the private key ssk, $Au_{i,j}$ represents the j-th authenticator in the i-th update interval.

Between a fixed update time point and a query time, more than one data update may happen. Under such circumstances, the cloud server may return the old Au in which tp is after the latest fixed update time point but before the query time. Namely there is at least one data update during this period. To resist this type of freshness attack, Zhu et al. introduce a timestamp-chain mechanism in [31]. In each update interval, it generates a timestamp-chain which is constructed according to Eq. (3) and the last authenticator in the chain also locates at the beginning of the next update interval.

Algorithm 2. MPT Building Algorithm

Input: Key, W, DC;
Output: Index I and authenticator Au.

```
Extract keyword dictionary W from DC ;
for w_i ∈ W do
    if |DC(wi)| ≥ k then
        Compute T_{w_i} = h_3(k_3, w_i) ;
        Compute S_{w_i} = h_1(k_1, w_i) ;
        Execute I = I.insert(T_{w_i}, S_{w_i}) ;
        else
            Compute T_{w_i} = h_3(k_3, w_i) ;
            for D_j ∈ DC(w_i) do
                Compute S_{w_i} = ∑ h_1(k_1, D_j) ;
                Execute I = I.insert(T_{w_i}, S_{w_i}) ;
            end
        end
    end
end
Generate the authenticator Au as Eq. (3) with k_4 and ssk ;
Send the index I and authenticator Au to the cloud server.
```

$$
\begin{cases}
\mathsf{xcon}_{i,0} = Enc(k_4, rt_{i,0} \| tp_{i,0}), up_i \le tp_{i,0} \le up_{i+1} \\
Au_{i,0} = \left(\mathsf{xcon}_{i,0}, Sig(ssk, \mathsf{xcon}_{i,0})\right) \\
\vdots \\
\mathsf{xcon}_{i,j} = Enc(k_4, rt_{i,j} \| tp_{i,j} \| \mathsf{xcon}_{i,j-1}), up_{i,j-1} \le tp_{i,j} \le up_{i+1} \\
Au_{i,j} = \left(\mathsf{xcon}_{i,j}, Sig(ssk, \mathsf{xcon}_{i,j})\right) \\
\vdots \\
\mathsf{xcon}_{i,n} = Enc(k_4, rt_{i,n} \| tp_{i,n} \| \mathsf{xcon}_{i,n-1}), tp_{i,n} = up_{i+1} \\
Au_{i,n} = \left(\mathsf{xcon}_{i,n}, Sig(ssk, \mathsf{xcon}_{i,n})\right)
\end{cases}
\tag{3}
$$

If no data update, the data owner just needs to generate the authenticator with a new timestamp at the fixed update time point. If data update happens, the data owner will generate the new authenticator with a new rt and tp. To check whether the rt is the latest one, the data user just needs to compare whether the tp in Au is before the latest update time.

5.3 Token Generation

This algorithm is run by data user and the procedure is described in Algorithm 3. The $Token$ can be regarded as the path from the root of the MPT to the node corresponding to the keyword. The cloud server could find the corresponding keyword in the MPT according to the $Token$.

Algorithm 3. Token Generation Algorithm

Input: k_3;
Output: $Token$.
for $w_i \in Q$ **do**
  Compute $T_{w_i} = h_3(k_3, w_i)$;
end
Send the $Token_Q = \{T_{w_1}, T_{w_2}, \cdots, T_{w_c}\}$ to the cloud server.

5.4 Proof Generation

Proof generation algorithm is run by the cloud server and the detailed description is illustrated in Algorithm 4. The checkpoint is the update time point which is closest to the user's query time.

The *proof* is used to provide necessary information for user to reconstruct the root of MPT. If the $Token$ sent by data user exists in MPT, the cloud server will generate corresponding *proof* to provide necessary information for user to reconstruct the root of MPT. If the $Token$ is not existing in MPT, the server could also generate the *proof*. Namely the server would return *proof* to user no matter whether the keyword exists or not. It will help user to detect whether the server deliberately omits all documents and returns an empty result to evade the integrity verification. In addition, as PGSSE is designed for the multi-keyword SSE scheme, the *proof* is not only a search path but also in the form of a sub-tree.

5.5 Integrity Verification

The integrity verification algorithm is used to check the integrity of search results and the procedure is described in Algorithm 5. According to the value of k, there are corresponding operations to calculate the node secret. According to whether the returned ciphertext and '*remain*' are null for the queried keyword w_i, it performs different operations.

(1) If both the ciphertext and '*remain*' are not null, the data user would look up the $array_i$ and recover the node secret with the help of the returned documents. Then he reconstructs the MPT's root and compare it with rt_q^t decrypted from Au_q^t.
(2) If both the ciphertext and '*remain*' are null, the data user directly reconstructs the MPT's root, and compare it with rt_q^t decrypted from Au_q^t. Only if they are matched, the data user will think that there is no search result for this keyword. Otherwise, the search results must be tampered.
(3) If one of the ciphertext and '*remain*' is null, the search results must be tampered and the verification algorithm return 0.

Algorithm 4. Proof Generation Algorithm

Input: Index $I, Token$ and t_q;
Output: $Proof$, authenticator Au_q^t and Au_c.

for $w_i \in Q$ **do**
  Find the search path $route_{w_i} = \{n_1, n_2, \cdots, n_s\}$ according to the $Token$, where $n_i \in \{EN, BN, LN\}, i \in [1, s]$, and n_1 is the root. The sequence of $route$ is the nodes from top to bottom of the MPT. Find the longest search path $route_{w_l}$;
end
if T_{w_l} *exists* **then**
  for $l = s - 1$ *to* 0 **do**
    if $n_l = BN$ **then**
      $Proof = Proof \cup V_{n_l}$, where V_{n_l} includes all key-value pairs of the descendant nodes of the BN;
    end
    else if $n_l = EN$ **then**
      $Proof = Proof \cup V_{n_l}$, where V_{n_l} is the key which is on the search path;
    end
    else
      $Proof = Proof \cup V_{n_l}$, where V_{n_l} is key-value pair of node n_l;
    end
  end
end
else
  for $l = s$ *to* 0 **do**
    Repeat steps from 6-14;
  end
end
Get the Proof generated based on the keyword w_l;
for $w_j \in Q$ *(except for* w_l*)* **do**
  Scan the $Proof$ and find the position of w_j and shade the node secret in the corresponding position;
end
Get the latest authenticator Au_q^t according to the query time and Au_c at the checkpoint;
Send $Proof, Au_q^t$ and Au_c to the data user.

5.6 Update Algorithm

The update operations include document addition, modification and deletion. According to whether there is addition or deletion of keywords, there are corresponding operations and it is described in Algorithm 6. If a keyword is newly added, it will insert a new node for this keyword. If there is no keyword addition, the MPT would remain unchanged and it only update the set P. For document deletion, it just needs to refresh the set P and keeps the MPT unchanged.

Algorithm 5. Integrity Verification Algorithm

Input: $k_2, k_4, S, P, C_Q, Proof, Token_Q, Au_q^t$;
Output: *result*.

for $w_i \in Q$ **do**
  Compute $remain_{w_i} = String.match(Token_{w_i}, Proof)$;
end
Let $(rt_q^t, tp_q^t, \mathsf{xcon}) = Dec(k_4, \mathsf{xcon}_q^t)$;
if $C_Q \neq null \wedge remain_{w_i} \neq null$ **then**
  Decrypt the C_Q, and get document collection D_Q;
  if $|D_Q| > k$ **then**
    for $D_j \in DC(w_i)$ **do**
      Compute $hash_{D_j} = h_2(k_2, D_j)$, and insert it into set M;
      Get $vx_{D_j} = Enc(k_4, hash_{D_j})$;
    end
    for $hash_{D_j} \in M$ **do**
      if $array_i[vx_{D_j}] = null$ **then**
        return *result*=0;
      end
      else
        According to $value_j = array_i[vx_{D_j}]$, there are k $(hash_{D_j}, value_j)$ pairs. Thus utilizing Shamir's secret sharing scheme with S to recover the node secret;
      end
    end
  end
  else if $|D_Q| < k$ **then**
    for $D_j \in D_Q$ **do**
      Compute $S_{w_i} = \sum h_1(k_1, D_j)$;
    end
    Calculate the root rt' according to the LN and $proof$ from bottom to the root;
    if $rt' = rt_q$ **then**
      return *result*=1;
    end
    else
      return *result*=0;
    end
  end
end
else if $C_Q = null$ *and at least one* $remain_{w_i}$ *doesn't exist*$(w_i \in Q)$ **then**
  Calculate the root rt' according to the $proof$ from bottom to the root;
  if $rt' = rt_q$ **then**
    return *result*=1;
  end
  else
    return *result*=0;
  end
end
else
  return *result*=0.
end

Algorithm 6. Update Algorithm

Input: The set P, update document D_j and I;
Output: New set P' and new MPT I'.

if *the document* D_j *is added into the DC* **then**
  for $w_i \in D_j$ **do**
    if $w_i \in W$ **then**
      Calculate $key_{D_j} = h_2(k_2, D_j)$ and $vx_{D_j} = Enc(k_4, key_{D_j})$;
      Extract the w_i's coefficients in S and rebuild the k-1 degree polynomials $f_i(x)$;
      Calculate $value = f_i(key_{D_j})$ and set $array_i[vx_{D_j}] = value$ in the set P';
    end
    if $w_i \notin W$ **then**
      Compute $T_{w_i} = h_3(k_3, w_i)$ and $S_{w_i} = h_1(k_1, w_i)$;
      Execute $I' = I.\mathsf{insert}(T_{w_i}, S_{w_i})$;
      Randomly generate the coefficients $(a_{i_1}, a_{i_2}, \cdots, a_{i,k-1})$ for w_i and then insert into S;
      Generate keyword w_i's array $(array_i)$ for the set P' according to Setup algorithm;
    end
  end
end
if *the document* D_j *is removed from DC* **then**
  for $w_i \in D_j$ **do**
    Calculate $key_{D_j} = h_2(k_2, D_j)$ and $vx_{D_j} = Enc(k_4, key_{D_j})$;
    Delete the $array_i[vx_{D_j}]$ in the set P'.
  end
end

6 Security and Performance Evaluation

The proposed PGSSE scheme acts as a general method for any SSE scheme for integrity verification. We needs to guarantee that PGSSE can preserve the data confidentiality and results verifiability for SSE schemes. It means that it does not leaks any useful information about documents and keywords in the verification process and it can be detected if the search results are tampered. The security proof of PGSSE is similar to that of Ref. [31]. To achieve top-k ranked search, we utilize Shamir's secret sharing scheme in PGSSE to replace incremental hash in GSSE of [31]. It will not bring more security risks. Because of space limitation, we omitted the formal security proof.

The performance of PGSSE includes storage overhead, the time overhead of index building, integrity verification and data updates. We compared the performance of PGSSE with GSSE to better evaluate its efficiency. The configuration of a PC used in experiments is core i5-M480 2.67 GHz CPU, 8 GB memory, and Win10 (64 bit) operation system. The SHA-1, 256-bit AES and 1024-bit RSA is used as the hash function, symmetric encryption/decryption algorithm and

signature algorithm respectively. The construction of the MPT is implemented in Java with about 800 lines code.

As the basic index structure of PGSSE is MPT that is same as GSSE, the storage overhead of PGSSE and GSSE is close to each other. As the size of MPT largely depends on the number of keywords in the dictionary, the storage overhead of both PGSSE and GSSE grows linearly with the growth of keywords when set the depth of MPT to be fixed. For 5000 documents in the document collection, the storage overhead is about 17 MB and 15 MB for PGSSE and GSSE respectively. As the PGSSE scheme has to store the coefficients of the Shamir's secret sharing polynomials and the set P of m arrays, the storage overhead of PGSSE is slightly more than that of GSSE.

6.1 MPT Construction

The time overhead of MPT construction in the PGSSE scheme largely depends on the number of document-keyword pairs in the inverted index. When set the depth of MPT and the number of documents to be 5 and 3000 respectively, Fig. 5(a) shows the time of MPT construction of both PGSSE and GSSE grows linearly with the growth of document-keyword pairs. For 30000 document-keyword pairs, the time of MPT construction is about 243 ms and 221 ms for PGSSE and GSSE respectively. When set the depth of MPT and the number of document-keyword pairs to be 5 and 5000 respectively, Fig. 5(b) shows the time of MPT construction of both PGSSE and GSSE grows also with the growth of the number of documents. For 3000 documents, the time of MPT construction is about 142 ms and 126 ms for PGSSE and GSSE respectively. It demonstrates that the time overhead of MPT construction for both schemes is positively correlated with the number of document-keyword pairs and documents. As the more the number of document-keyword pairs, the more the dimension of the Shamir's

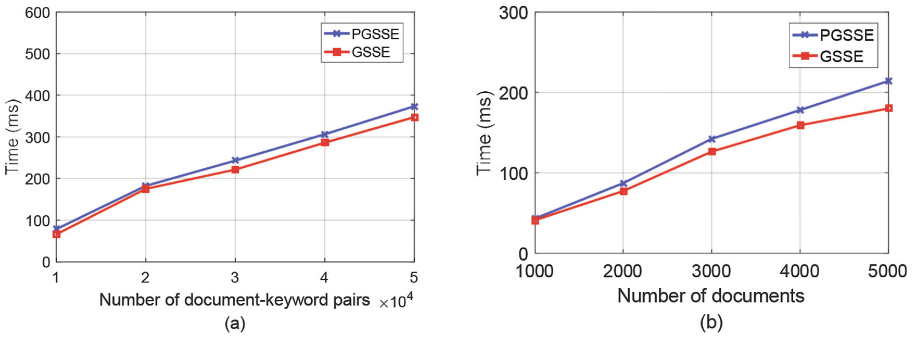

Fig. 5. (a) The time cost of MPT construction with variable number of document-keyword pairs (MPT depth = 5 and the number of documents is $n = 3000$); (b) the time cost of MPT construction with variable number of documents (MPT depth = 5 and the number of document-keyword pairs is 5000).

secret sharing matrix and the more keyword arrays in the set P will be generated in PGSSE, hence the time overhead of PGSSE is a little more than that of GSSE.

6.2 Integrity Verification

The time cost of integrity verification in PGSSE largely depends on the threshold k and the number of queried keywords. The bigger the threshold k is, the more documents will be returned. As a result, the more time will be consumed to construct node secret in the "Verify" algorithm.

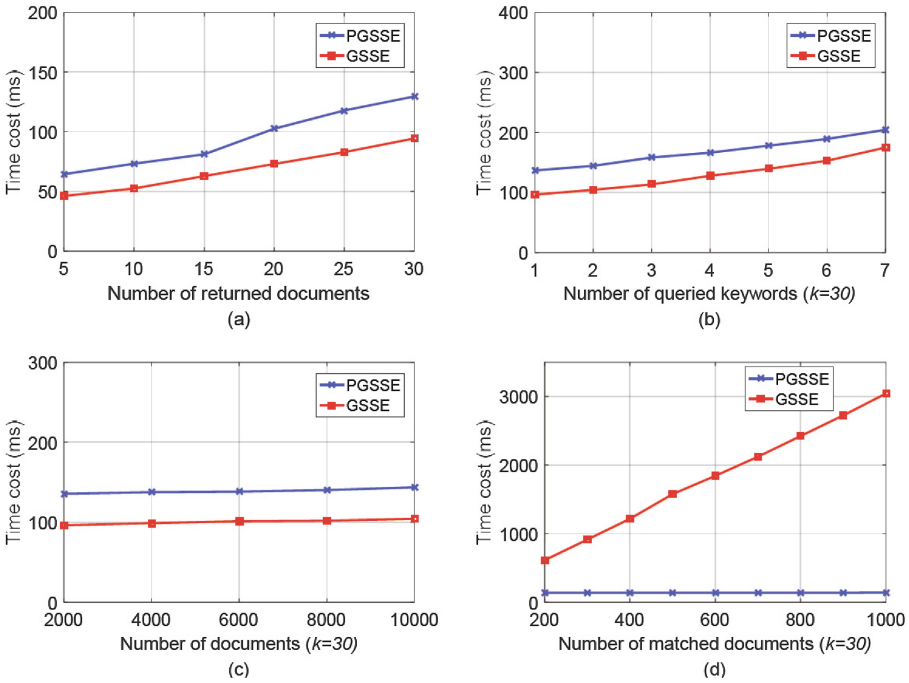

Fig. 6. (a) The time cost of integrity verification with variable k and fixed number of documents $n = 3000$; (b) the time cost of integrity verification with variable number of queried keywords ($n = 3000$ and $k = 30$); (c) the time cost of integrity verification with variable number of documents ($k = 30$); (d) the time cost of integrity verification with variable number of matched documents ($k = 30$).

Assume the number of documents $n = 3000$, Fig. 6(a) shows the time cost of integrity verification of both PGSSE and GSSE grow linearly with k. To return 15 documents, the time of verification is about 81.1 ms and 62.8 ms for PGSSE and GSSE respectively. Assume the number of documents $n = 3000$ and $k = 30$, Fig. 6(b) shows the time cost of integrity verification of both PGSSE and GSSE

grow linearly with the number of queried keywords. For 7 queried keywords, the time of verification is about 204.2 ms and 174.6 ms for PGSSE and GSSE respectively. As the time cost of integrity verification is related to the number of returned documents, for $k = 30$, Fig. 6(c) shows the time cost of both PGSSE and GSSE keeps stable with the growth of the number of documents.

The above experimental results show that the efficiency of integrity verification of PGSSE is a bit lower than that of GSSE. In fact, as PGSSE is proposed to improve the practicability of GSSE to enable the ranked top-k search, Fig. 6(d) shows that the efficiency of integrity verification of PGSSE is superior to that of GSSE when the number of matched documents increases sharply. For $k = 30$, if the number of matched documents is 200, the time cost is about 136 ms and 612 ms for PGSSE and GSSE respectively. If the number of matched documents is 1000, the time cost is about 140 ms and 3045 ms for PGSSE and GSSE respectively. This is because of that GSSE has to get all documents which contain the queried keyword in DC to perform the integrity verification. While PGSSE just needs to get the top-k documents to perform the integrity verification, and thus the verification time keeps stable in PGSSE with the growth of matched documents.

6.3 Data Updates

Differing from GSSE, PGSSE only updates the keyword array and the MPT remains unchanged if there is no keyword addition or deletion. Assume the number of documents $n = 3000$, Fig. 7(a) shows that the time cost of data updates of PGSSE keeps stable with the growth of the number of added documents, while it grows linearly with the growth of the number of added documents for that of GSSE. For adding 400 documents, the time cost is about 77 ms and 355 ms for PGSSE and GSSE respectively.

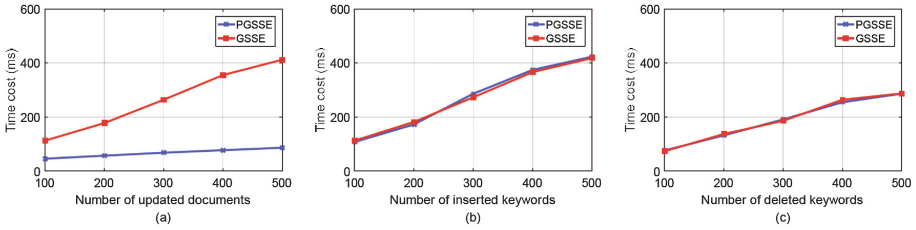

Fig. 7. (a) The update time cost with no keyword insertion and deletion (the number of documents $n = 3000$); (b) the update time cost with keyword insertion ($n = 3000$); (c) the update time cost with keyword deletion ($n = 3000$).

If there is new keywords addition in data updates, it will insert new nodes to the MPT for the new keywords and update the keyword array for PGSSE. While it will also insert new nodes to the MPT for the new keywords and update

the incremental hashes of these nodes for GSSE. Figure 7(b) shows that the time cost of data updates of both PGSSE and GSSE grows linearly with the number of added keywords and the time cost is similar. If there is keywords deletion in data updates, it will delete the corresponding nodes of MPT for both PGSSE and GSSE. Figure 7(c) shows that the time cost of data updates of both PGSSE and GSSE grows linearly with the number of deleted keywords and the time cost is also similar.

7 Conclusion

To improve the practicability of existing SSE schemes, we proposed a general and efficient method that provides dynamic and efficient integrity verification for SSE construction that is decoupled from original SSE schemes. The proposed PGSSE overcomes the disadvantages of the GSSE on ranked search. The experimental results demonstrate that PGSSE is greatly superior to GSSE in integrity verification and data updates for top-k ranked search.

References

1. Cao, N., Wang, C., Li, M., Ren, K., Lou, W.: Privacy-preserving multi-keyword ranked search over encrypted cloud data. In: 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2011, Shanghai, China, 10–15 April 2011, pp. 829–837 (2011)
2. Cao, N., Wang, C., Li, M., Ren, K., Lou, W.: Privacy-preserving multi-keyword ranked search over encrypted cloud data. IEEE Trans. Parallel Distrib. Syst. **25**(1), 222–233 (2014)
3. Cash, D., et al.: Dynamic searchable encryption in very-large databases: Data structures and implementation. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, 23–26 February 2014
4. Chai, Q., Gong, G.: Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In: Proceedings of IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada, 10–15 June 2012, pp. 917–922 (2012)
5. Chen, C., et al.: An efficient privacy-preserving ranked keyword search method. IEEE Trans. Parallel Distrib. Syst. **27**(4), 951–963 (2016)
6. Chen, L., Qiu, L., Li, K., Shi, W., Zhang, N.: DMRS: an efficient dynamic multi-keyword ranked search over encrypted cloud data. Soft Comput. **21**(16), 4829–4841 (2017)
7. Chen, X., Li, J., Weng, J., Ma, J., Lou, W.: Verifiable computation over large database with incremental updates. IEEE Trans. Comput. **65**(10), 3184–3195 (2016)
8. 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, CCS 2006, Alexandria, VA, USA, 30 October–3 November 2006, pp. 79–88 (2006)

9. Fu, Z., Ren, K., Shu, J., Sun, X., Huang, F.: Enabling personalized search over encrypted outsourced data with efficiency improvement. IEEE Trans. Parallel Distrib. Syst. **27**(9), 2546–2559 (2016)
10. Gajek, S.: Dynamic symmetric searchable encryption from constrained functional encryption. In: Proceedings of Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, 29 February–4 March 2016, pp. 75–89 (2016)
11. Goh, E.: Secure indexes. IACR Cryptology ePrint Archive 2003, p. 216 (2003)
12. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014, pp. 310–320 (2014)
13. Jiang, X., Yu, J., Yan, J., Hao, R.: Enabling efficient and verifiable multi-keyword ranked search over encrypted cloud data. Inf. Sci. **403**, 22–41 (2017)
14. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, 1–5 April 2013, Revised Selected Papers, pp. 258–274 (2013)
15. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: The ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, 16–18 October 2012, pp. 965–976 (2012)
16. Kurosawa, K., Ohtaki, Y.: UC-secure searchable symmetric encryption. In: Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, 27 Februray–2 March 2012, Revised Selected Papers, pp. 285–298 (2012)
17. Kurosawa, K., Ohtaki, Y.: How to update documents verifiably in searchable symmetric encryption. In: Proceedings of Cryptology and Network Security - 12th International Conference, CANS 2013, Paraty, Brazil, 20–22 November 2013, pp. 309–328 (2013)
18. Li, R., Xu, Z., Kang, W., Yow, K., Xu, C.: Efficient multi-keyword ranked query over encrypted data in cloud computing. Future Gener. Comp. Syst. **30**, 179–190 (2014)
19. van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P.H., Jonker, W.: Computationally efficient searchable symmetric encryption. In: Proceedings of Secure Data Management, 7th VLDB Workshop, SDM 2010, Singapore, 17 September 2010, pp. 87–100 (2010)
20. Liu, Z., Li, T., Li, P., Jia, C., Li, J.: Verifiable searchable encryption with aggregate keys for data sharing system. Future Gener. Comp. Syst. **78**, 778–788 (2018)
21. Naveed, M., Prabhakaran, M., Gunter, C.A.: Dynamic searchable encryption via blind storage. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, 18–21 May 2014, pp. 639–654 (2014)
22. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
23. Song, D.X., David A. Wagner, Perrig, A.: Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, 14–17 May 2000, pp. 44–55 (2000)
24. Sun, W., Liu, X., Lou, W., Hou, Y.T., Li, H.: Catch you if you lie to me: efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data. In: 2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, 26 April–1 May 2015, pp. 2110–2118 (2015)
25. Sun, W., et al.: Verifiable privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. IEEE Trans. Parallel Distrib. Syst. **25**(11), 3025–3035 (2014)

26. Wang, C., Cao, N., Li, J., Ren, K., Lou, W.: Secure ranked keyword search over encrypted cloud data. In: 2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, 21–25 June 2010, pp. 253–262 (2010)
27. Wang, C., Cao, N., Ren, K., Lou, W.: Enabling secure and efficient ranked keyword search over outsourced cloud data. IEEE Trans. Parallel Distrib. Syst. **23**(8), 1467–1479 (2012)
28. Xia, Z., Wang, X., Sun, X., Wang, Q.: A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. IEEE Trans. Parallel Distrib. Syst. **27**(2), 340–352 (2016)
29. Yu, J., Lu, P., Zhu, Y., Xue, G., Li, M.: Toward secure multikeyword top-k retrieval over encrypted cloud data. IEEE Trans. Dependable Sec. Comput. **10**(4), 239–250 (2013)
30. Zhang, W., Lin, Y., Xiao, S., Wu, J., Zhou, S.: Privacy preserving ranked multi-keyword search for multiple data owners in cloud computing. IEEE Trans. Comput. **65**(5), 1566–1577 (2016)
31. Zhu, J., Li, Q., Wang, C., Yuan, X., Wang, Q., Ren, K.: Enabling generic, verifiable, and secure data search in cloud services. IEEE Trans. Parallel Distrib. Syst. **29**(8), 1721–1735 (2018)