Int. J. Inf. Secur. (2018) 17:549–568
https://doi.org/10.1007/s10207-017-0394-9

SPECIAL ISSUE PAPER

Integrity-verifiable conjunctive keyword searchable encryption in cloud storage

Yuxi Li[1] · Fucai Zhou[1] · Yuhai Qin[2] · Muqing Lin[3] · Zifeng Xu[1]

Published online: 8 November 2017

Abstract Conjunctive searchable encryption is an efficient way to perform multi-keyword search over encrypted data in cloud storage. However, most existing methods do not take into account the integrity verification of the search result. Moreover, existing integrity verification methods can only verify the integrity of single-keyword search results, which cannot meet the requirements of conjunctive search. To address this problem, we proposed a conjunctive keyword searchable encryption scheme with an authentication mechanism that can efficiently verify the integrity of search results. The proposed scheme is based on the dynamic searchable symmetric encryption and adopts the Merkle tree and bilinear map accumulator to prove the correctness of set operations. It supports conjunctive keyword as input for conjunctive search and gives the server the ability to prove the integrity of the search result to the user. Formal proofs and extensive experiments show that the proposed scheme is efficient, unforgeable and adaptive secure against chosen-keyword attacks.

Keywords Conjunctive keyword search · Integrity authentication · Searchable encryption · Secure cloud storage

✉ Fucai Zhou
fczhou@mail.neu.edu.cn

[1] Software College, Northeastern University, Shenyang 110819, People's Republic of China

[2] Cyber Crime Investigation Department, National Police University of China, Shenyang 110819, People's Republic of China

[3] Information Security Institute, Beijing Electronic Science and Technology Institute, Beijing 100070, People's Republic of China

1 Introduction

Cloud computing is an innovative Internet-based computing paradigm that enables cloud users to move out their data and applications to a remote cloud in order to deploy scalable and elastic services on demand without having to provision a data center. Cloud services have many advantages, and one of the most popular services is cloud storage. However, data breaches raise privacy concerns and slow down the improvement of cloud storage. According to a survey launched by Twin Strata in 2015, only 38% of organizations would like to put their inactive data stored in public cloud; about 24% of users were using cloud storage for data backup, archiving and disaster recovery. This shows that the issue of data security [1,2] is one of the major obstacles to the promotion of cloud storage. Since the user's data is outsourced to distributed cloud servers, the service provider can easily access the data.

Data breach statistics launched by Breach Level Index shows that data records lost or stolen since 2013–2017 are more than 9 billion times, and only 4% of breaches were "Secure Breaches" where encryption was used and the stolen data was rendered useless. Therefore, to prevent data from being maliciously accessed by cloud providers, data owners tend to encrypt their private data before outsourcing to the cloud, and they only share the decryption key to other authorized users. Although this method can protect the privacy of the data, it brings data retrieval problems. For example, it prevents the server from searching the content of the data when responding the users search request, such as searching in a backup or email archive. This limitation has motivated much research on advanced searchable encryption schemes that enable searching on the encrypted data while protecting the confidentiality of the data and queries.

The solution of searchable encryption that first proposed by Song et al. [3] provides a way to perform efficient keyword searches over encrypted data. Promoted by Songs pioneering work, many efforts have been devoted to construct more efficient searchable symmetric encryption (SSE) schemes, such as [4–9]. An SSE scheme allows users to encrypt their data using symmetric encryption, and then uses files and keywords to create the encrypted index for further searches. When the user wants to retrieve some files, he needs to choose a keyword and use it to generate a search request. After that, the server uses this special request to search over its internal data structure. At last, the server finds all the files related to that keyword and returns the file collection to the user. Besides performing successful searches, the privacy feature of the SSE also ensures that, given encrypted files, encrypted indexes and a series of search requests, the server cannot learn any useful information about the files and the keywords.

The solutions above are single-keyword oriented, which are inefficient in practice since the searches may return a very large number of files, such as when searching in a remote-stored email archive. The works in [10,12–14] extend the search primitive to the multi-keyword conjunctive search, which avoid this limitation and are more practical for real-world scenarios.

However, most of these solutions do not consider the integrity verification of the search result. In most of existing works, the threat models of malicious server are often defined as honest but curious, which are insufficient in dealing with real-world security threats. For example, due to the storage failure or some other reasons, the service provider may ignore some special files on purpose during a search and return the rest of the files to the user. In this case, the user can only determine if the files he received contain the keywords he is querying, but cannot examine whether the file set received is the complete set. Therefore, an authentication mechanism is required to guarantee the correctness and integrity of search results.

To the best of our knowledge, few works consider the searchable encryption and the search authentication together. Kamara et al. [14] presented a dynamic SSE construction with a search authenticate mechanism that allows user to verify the integrity of the search results. They used a simple Merkle tree structure and a precomputed basis to authenticate the given dataset. Kurosawa et al. [15] introduced the definition of UC security and proposed a verifiable SSE scheme that allows the user to detect search results integrity. However, the authentication methods in these schemes are only capable for the single-keyword search. They cannot produce proofs for the conjunctive keyword search results. If a user wants to perform a conjunctive keyword search using these schemes, he needs to execute each single-keyword search respectively, and verify all those result sets, then compute the intersection at local to get the final result. The drawback of this method is evident, because the search may return very large numbers of files in most situations like searching in a remote-stored email archive or a database backup. The communication complexity is linear and may have performance problems when the data scale is very large.

Even today, efficient integrity-verifiable conjunctive keyword search over encrypted data remains a challenging problem. Hence, in this paper, we focus on enabling search authentication in conjunctive keyword searchable encryption schemes to fulfill the practical needs. We reduce the conjunctive keyword search problem to the single-keyword case by performing a search for each individual keyword and doing the intersection between each resultant file sets to get the final result. To lower the communication overhead during a search, the intersection of each keywords search result should be computed at the server side. The only thing that the user needs to do is to receive the final result and verify its integrity. Thus, the new approach should meet the following requirements: (1) the server should be able to take conjunctive keyword as input, and give the final result directly; (2) for the server that honestly executes the search algorithm, a valid proof can be formed and pass the verification; no one can generate a valid proof for a maliciously modified search result and still pass the verification. Theoretical basis of proposed solution is inspired by the authenticated data structure in [10] to verify set operations on out sourced sets.

We use dynamic SSE to realize the single-keyword search and use Merkle tree as the base data structure to prove the correctness of the intersection. Given a search request that contains multiple keywords, e.g., $\{w_1, \ldots, w_n\}$, the SSE scheme outputs a set collection $\{S_1, \ldots, S_n\}$, where each S_i is a file set that related to the keyword w_i. It is then straight forward for the server to compute the intersection $\mathbf{I} = S_1 \cap \cdots \cap S_n$ and return to the user the final set $\mathbf{I}$. In the meantime, for the intersection $\mathbf{I} = S_1 \cap \cdots \cap S_n$, we prove its correctness in three steps. First, the Merkle tree proof is used to ensure the integrity of each S_i. Second, bilinear map accumulator is adopted to prove that the set $\mathbf{I}$ is the subset of each S_i. Third, to ensure $\mathbf{I}$ is the complete set of the intersection, we use the extended Euclidean algorithm to give the completeness proof to ensure that $\mathbf{I}$ is not the proper subset of $S_1 \cap \cdots \cap S_n$. These procedures can accurately guarantee the correctness of the intersection. The underlying mathematical assumption of these proofs is the bilinear q-Strong Diffie–Hellman assumption. If an adversary can forge a fake intersection proof and pass the verification, it then has the ability to solve the bilinear q-Strong Diffie–Hellman problem with non-negligible probability.

Based on these insights, we present a dynamic integrity-verifiable conjunctive keyword searchable encryption scheme which maintains the adaptive chosen-keyword security and is unforgeable against adaptive adversaries. The main contributions of this paper are:

1. We proposed the first conjunctive keyword searchable encryption method that is capable for the verification of search results integrity and allows user to dynamically add or delete files.
2. We gave the formal mathematical proofs under random oracle model to claim that our scheme is secure against the adaptive chosen-keyword attack and cannot be forged by adaptive adversaries.
3. We implemented our scheme and performed an evaluation using real-world data set. The results show that our scheme has low communication overhead.

The remainder of this paper is arranged as follows. Section 2 lists a few related works in the literature. Section 3 is the preliminaries of our scheme. Section 4 gives the formal definition and security models of proposed solution. Section 5 provides the detailed construction of our integrity preserving searchable encryption scheme. Section 6 gives the formal security proofs of proposed scheme. Section 7 presents the experiment results, and Sect. 8 concludes this work.

2 Related works

2.1 Searchable encryption

The question how to do keyword searches on encrypted data efficiently was first raised by Song et al. [3]. Their scheme allows a user, given a trapdoor for a word, to test if a cipher text block contains the word. Their scheme is not secure against statistical analysis across encrypted data, since their approach could leak the locations of the keyword in a document. Goh [4] introduced the formal security definitions for searchable symmetric encryption (SSE) and proposed a construction that is based on Bloom filters. The construction associates an index to each document in a collection and the server has to search each of these indexes to answer a query. The schemes search time is linear in the number of documents and may return false positives because of the adoption of Bloom filters. The index-based approach done by Goh was followed by Chang et al. [5] who gave a simulation-based definition and a construction with linear search time but no false positives. It was observed by Curtmola et al. [6] that previous security definitions are insufficient for the setting of SSE. They introduced and formalized a stronger notion of security against adaptive chosen-keyword attacks. Furthermore, they gave the first construction to achieve sublinear search time. Kamara et al. [7] proposed a dynamic SSE scheme that is adaptive chosen-keyword secure and supports add and delete dynamically. Cast et al. [8] designed a dynamic searchable encryption which achieves efficient searches over large scale databases. Their scheme can support searches in server-held encrypted databases with tens of billions of record–keyword pairs. Stefanov et al. [9] proposed a dynamic searchable encryption scheme which has very small leakage against the untrusted server. The scheme also supports both updates and searches in sublinear time in the worst case.

The solutions proposed in [10,12,14] focus on how to do multi-keyword conjunctive search over encrypted data while providing privacy guarantees. Computational overheads of these schemes are linear to the size of the file set. Cash et al. [8] proposed a solution which reduces the computational overhead to sublinear and extends the search pattern to Boolean queries. Cao et al. [13] built a conjunctive keyword ranked search scheme, which allows privacy preserving conjunctive keyword searches, and can sort the results in the order of their relevance to these keywords.

Encryption with keyword search has also been considered in the public-key setting [16–18] , which is beyond the scope of this paper.

2.2 Searchable encryption with authentication

There are few works that focus on both the searchable encryption and the search authentication. Kamara et al. [7] presented a cryptographic cloud storage system which combines an adaptive secure searchable symmetric encryption scheme with a search authenticate mechanism to allow the user to verify the integrity of the search result. Kurosawa et al. [15] considered the untrusted server as an active adversary and defined the UC security that enables the user to detect whether the files are received correctly. They then proposed a verifiable SSE scheme to ensure the search results integrity. The search authentication techniques in these solutions are all single-keyword oriented, which are inefficient in the multi-keyword scenario.

3 Preliminaries

3.1 CPA-secure private key encryption

A private key encryption scheme is defined as $\mathrm{SKE} = (\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$, in which $K \leftarrow \mathrm{Gen}(1^k)$ and $c \leftarrow \mathrm{Enc(K, m)}$ are two probabilistic algorithms and $m \leftarrow \mathrm{Dec(K, c)}$ is a deterministic algorithm. A private key encryption scheme is CPA-secure [19], if the ciphertexts it produces do not reveal any partial information about the plaintext even to an adversary that can adaptively query an encryption oracle. Formally, for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, the probability:

$$|Pr[K \leftarrow \mathrm{Gen}(1^k); (m_0, m_1) \leftarrow \mathcal{A}^{\mathrm{Enc_K}}(\cdot)(1^k);$$
$$b \leftarrow \{0, 1\}; c \leftarrow \mathrm{Enc_K}(m_b); \hat{b} \leftarrow \mathcal{A}^{\mathrm{Enc_K}}(\cdot)(c) :$$

$\hat{b} = b] - 1/2| \leq \text{negl}(k)$,

where negl(k) is a negligible function with input k.

3.2 Merkle tree

A Merkle tree [20] is a complete binary tree with hashes in the internal nodes. The indexed data are assigned to the leaves, and the values of internal nodes are computed through the hash function with the value of their two children as input. After the construction, the root node characterizes all the leaves, and is possessed by the verifier. Merkle tree can be used to prove the given data are indeed the data stored in the leaf. Such proof includes the leaf and all the sibling nodes in the path from that leaf to the root. Verifier can compute the hashes in the proof and get a root. By comparing the new root with the old one, the verifier can determine whether the proof is valid. The collision-resistant property of the hash function guarantees that any PPT adversary cannot forge a valid proof using maliciously altered data.

3.3 Bilinear map accumulator

The bilinear map accumulator [21] is an efficient tool to provide proofs of membership for elements that belong to a set. It has a number of attractive properties which are useful to verify the correctness of the final result **I** by the user which is unaware of the value of $S_1, \ldots, S_n$. To the best of our knowledge, the bilinear map accumulator has the shortest signature sizes compared to corresponding previously proposed such accumulators. We call $(p, \mathbf{G}, \mathcal{G}, e, g)$ a tuple of admissible bilinear pairing parameters, produced as the output of a PPT algorithm that runs on input.

Given the bilinear pairing parameters $(p, \mathbf{G}, \mathcal{G}, e, g)$, let $s \in \mathbf{Z}_p^*$ be a randomly chosen trapdoor. The accumulator accumulates elements in $\mathbf{Z}_p$, and outputs an element in $\mathbf{G}$. For a set of elements $\mathcal{X}$ in $s \in \mathbf{Z}_p^*$, the accumulation value acc$(\mathcal{X})$ is defined as:

$$\text{acc}(\mathcal{X}) = g^{\prod_{x \in \mathcal{X}}(x+s)}.$$

Without knowing the trapdoor s, the value acc$(\mathcal{X})$ can also be constructed using $\mathcal{X}$ and the precomputed $(g, g^s \ldots, g^{s^q})$, where $q \geq \#\mathcal{X}$. The proof of subset containment of a set $\mathcal{S} \subseteq \mathcal{X}$ is the witness $(\mathcal{S}, \mathcal{W}_{\mathcal{S},\mathcal{X}})$ where:

$$\mathcal{W}_{\mathcal{S},\mathcal{X}} = g^{\prod_{x \in \mathcal{X}-\mathcal{S}}(x+s)}.$$

Subset containment of $\mathcal{S}$ in $\mathcal{X}$ can be verified by checking:

$$e\left(\mathcal{W}_{\mathcal{S},\mathcal{X}}, g^{\prod_{x \in \mathcal{S}}(x+s)}\right) = e\left(\text{acc}(\mathcal{X}), g\right)$$

The security of the bilinear map accumulator relies on the bilinear q-Strong Diffie–Hellman assumption. Let $(p, \mathbf{G}, \mathcal{G}, e, g)$ be the bilinear pairing parameters with security parameter k. The bilinear q-SDH assumption in $(p, \mathbf{G}, \mathcal{G}, e, g)$ is defined as follows: Given a $(q+1)$-tuple $(g, g^s, \ldots, g^{s^q})$ as input for some randomly chosen $s \in \mathbf{Z}_p^*$, no PPT adversary can output a pair with non-negligible probability, where a is an element in $\mathbf{Z}_p^*$.

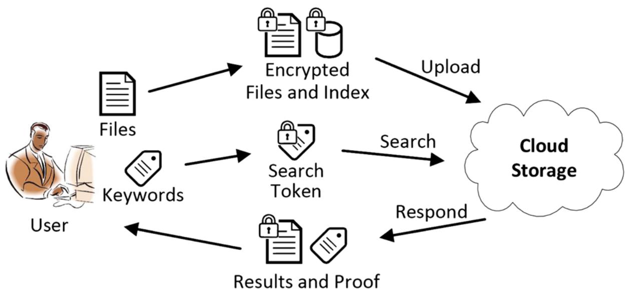

Fig. 1 Integrity-verifiable conjunctive keyword encrypted search

4 Definition and security model

4.1 Notations and definitions

We consider the scenario that consists of two types of entities. One of them is the user that owns the data, and the other is the cloud storage provider, as known as the server, which provides storage services to the user. Our scheme allows a user to encrypt his data and outsource the encrypted data to the server. After uploading the encrypted data, the user only needs to store a secret key and an authenticated data state, regardless of the file number and size, i.e., the users storage overhead is constant size. User can later generate search requests using single or conjunctive keyword and submit to the server. Given a search request, the server searches over the encrypted data and returns the set of encrypted files and a corresponding proof. The correctness of this search result can be verified by the user, using this result and proof. User can also dynamically update the file set on demand after the first uploading. The main system architecture is shown in Fig. 1.

While using conjunctive keyword in a search, we define the search result to be the intersection of the sets generated by searching for each individual keyword. Concretely speaking, the question we discussed in this paper is the conjunctive keyword search. We use token to describe the request sent by user. For example, to start a search, the user forms a search token and sends it to the server. The server uses this token as input to perform the search. Since our scheme is dynamic, there are two additional tokens, the add token and the delete token.

We will use the notations in Table 1 and through the rest of the paper. Various data structures are used including the linked lists, arrays and the lookup tables. If L is a list, then #L

Table 1 Notations

Notation	Definition
$\mathbf{v}_i$ or $\mathbf{v}[i]$	ith element of a sequence of elements
$\#\mathbf{v}$	Total number of elements in $\mathbf{v}$
$\mathbf{v}[i] = v$	Stores v at location i in $\mathbf{v}$
$\mathbf{f} = (f_1, \dots, f_n)$	Universal set of files
$\mathbf{w}$	Set of all keywords
$\mathbf{f}_w$	All files contained the word w
$\#\mathbf{f}_w$	Number of files contain word w
$\mathbf{w}_f$	Set of all words in file f
$\#\mathbf{w}_f$ or $\#f$	Number of all keywords in f
$\|f\|$	Size of file f
id_f or $id(f)$	Identifier of file f
$id(w)$	Identifier of keyword w
δ	Inverted index
γ	Encrypted index
$\tau_s/\tau_a/\tau_d$	Search/add/delete token
α	An authenticated structure MHT
st	Root node in MHT
θ_w	Lead node corresponding to w

denotes its total number of nodes. If A is an array, then #A is its total number of cells, A$[i]$ is the value stored at location i, and A$[i] = v$ denotes the operation that stores v at location i in the array A. The lookup table is a data structure that stores key/value pairs (σ, v) that if the pair (σ, v) is in the lookup table T, then T$[\sigma]$ denotes the value v associated with the key σ. #T denotes the number of pairs in the table.

The formal definition of our scheme is defined as follows.

Definition 1 Our scheme is a tuple of eight polynomial time algorithms and protocols such that:

$K \leftarrow \text{Gen}(1^k)$ is a probabilistic algorithm run by the user that takes a security parameter 1^k as input, outputs a secret key K.

$(\gamma, \mathbf{c}, st, \alpha) \leftarrow \text{Setup}(K, \delta, \mathbf{f})$: is a probabilistic algorithm run by the user that takes the secret key K, an index δ and a set of files $\mathbf{f}$ as input, outputs an encrypted index γ, a set of ciphertexts $\mathbf{c}$, a data state st and an authenticated structure α.

$\tau_s \leftarrow \text{SrchToken}(K, W)$ is a deterministic algorithm run by the user that takes as input the secret key K and a set of words W, outputs search token τ_s.

$(\mathbf{I}_W, \pi) \leftarrow \text{Search}(\alpha, \gamma, \mathbf{c}, \tau_s)$ is a deterministic algorithm run by the server that takes as input the authenticated structure α, the encrypted index γ, the set of ciphertexts $\mathbf{c}$ and the search token τ_s, outputs a set of file identifiers $\mathbf{I}_W$, and a proof π.

$b \leftarrow \text{Verify}(K, st, \tau_s, \mathbf{I}', \pi)$ is a deterministic algorithm run by the user that takes as input the secret key K, the data state st, a search token τ_s, a set of file identifiers $\mathbf{I}'$ and a proof π, outputs 1 as accept or 0 as reject.

$f \leftarrow \text{Dec}(K, c)$ is a deterministic algorithm run by the user that takes as input the secret key K and a ciphertext c, outputs a plaintext file f.

Generally, there are three phases in the scheme. The initialization phase contains the first two algorithms, Gen and Setup. This phase allows the user to encrypt his batch files and upload the encrypted file set $\mathbf{c}$ to the server, along with the encrypted index γ and the authenticated structure α. After this phase, the user only possesses the secret key K and the data state st.

The search phase contains four algorithms, SrchToken, Search, Verify and Dec. Before a search, the user uses the SrchToken algorithm to compute a search token τ_s and sends it to the server. The server uses this token to search over its internal data and output a set of file identifiers $\mathbf{I}_W$ and a proof π. Given a set $\mathbf{I}'$ and a proof π, the user can verify its integrity by running the Verify algorithm. The Dec algorithm is used to decrypt the files that the user received according to those identifiers.

The update phase contains two interactive protocols, Add/Update and Del/Update. This phase is used when the file set on the server side needs to be updated. The add token τ_a and delete token τ_d are used in the protocols to serve as the update requests from user to server. After these protocols, the servers internal data have been updated, along with the users data state st too.

The correctness of the scheme implies that, for all security parameter k, for all K generated by Gen(1^k), for all $\mathbf{f}$ and δ, for all $(\gamma, \mathbf{c}, st, \alpha)$ outputs by Setup$(K, \delta, \mathbf{f})$, and for all sequences of search, add or delete operations on γ and α, searches always output the correct results, and the proofs are always accepted by the user.

4.2 Security model

We consider the server to be an untrusted entity, which may deliberately steal or sabotage the users data, or ignore some special files in the search result. Intuitively, an integrity preserving searchable encryption scheme should meet the following security features: (1) The encrypted files and data structures on the server side should not leak any information about the files to the server; (2) the search requests generated by the user should not leak any information about the keywords he uses; (3) for a fallacious result, the server cannot produce a valid proof and pass the users verification.

However, the first two requirements are difficult to meet in practice. Most of the known efficient searchable encryption algorithms more or less leak information such as the users access pattern. The leakage problem is inevitable in searchable encryption, which is also the reason why searchable encryption scheme is more efficient than ORAM [22]. To give the formal security model in this case, we follow the methods in [6] and capture which information is leaked in the scheme by defining a set of leakage functions. The formal security definitions and proofs will be given under the presence of these functions.

4.2.1 Dynamic adaptive chosen-keyword security

This security requirement characterizes the feature that the scheme does not leak any information to the adversary except those defined in the leakage functions. The security definition will be parameterized by the four leakage functions $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$, and $\mathcal{L}_4$. $\mathcal{L}_1$ denotes the information about the message that is leaked in the Setup algorithm. $\mathcal{L}_2$ denotes $id(w)$ for all $w \in W$that reveals to the server during the search operation. $\mathcal{L}_3$ is the information that the server can learn in the add protocol. Similarly, $\mathcal{L}_4$ represents the message leaked to the server in the delete protocol. The adversary is allowed to be adaptive, i.e., its queries could base on the previous results. This gives the adversary more ability to attack than those selective ones. Let $\mathcal{A}$ be a stateful adversary that executes the server-side algorithm, game represents the interaction between $\mathcal{A}$ and user or simulator, view represent all the information that $\mathcal{A}$ can collect during the game. We assume that, $\mathcal{A}$ can choose the encrypted message, and then generates the queries by interacting with the user adaptively. Therefore, in our security definition, the view of $\mathcal{A}$ should only contain the information specified by$\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$ and $\mathcal{L}_4$ in a simulated way.

The real game $\text{Real}_{\mathcal{A}}(1^k)$ is run between $\mathcal{A}$ and the user using the real scheme. In the real game, user initializes his data structures using the data provided by $\mathcal{A}$. After that, $\mathcal{A}$ makes at most polynomial times queries to the user. These queries may be the search, add file or delete file operations composed in any order $\mathcal{A}$ chooses. Each queries result can be the input for $\mathcal{A}$ to generate the next query. After all q times queries, all the data that $\mathcal{A}$ collects constitute the $view_{\text{Real}}$, and $\mathcal{A}$ outputs one bit as the games output.

The ideal game $\text{Ideal}_{\mathcal{A},\mathcal{S}}(1^k)$ is run between $\mathcal{A}$ and a stateful simulator $\mathcal{S}$. The only difference between the ideal game and the real game is that, in the ideal game, $\mathcal{S}$ does not run the real scheme, it responds to $\mathcal{A}$s queries using randomly generated data, with these leakage functions as the only input. After all q times queries, all the data that $\mathcal{A}$ collects constitute the $view_{\text{Ideal}}$, and $\mathcal{A}$ outputs one bit as the games output.

The dynamic adaptive chosen-keyword security requires that, there exists a PPT simulator $\mathcal{S}$ such that no PPT adversary can distinguish between the $view_{\text{Real}}$ and the $view_{\text{Ideal}}$. Namely, the adversary should not be able to tell which one its interacting with, the real user or the simulator. We give the formal definition as follow.

Definition 2 Given the scheme described in Definition 1, describe $\mathcal{A}$ as a stateful adversary, $\mathcal{S}$ as a stateful simulator, $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$, $\mathcal{L}_4$ as stateful leakage functions. Consider the following games:

```
$\text{Real}_{\mathcal{A}}(1^k)$ :
  $K \leftarrow \text{Gen}(1^k)$
  $(\delta, \mathbf{f}) \leftarrow \mathcal{A}(1^k)$
  $(\gamma, \mathbf{c}, st, \alpha) \leftarrow \text{Setup}(K, \delta, \mathbf{f})$
  for $1 \le i \le q$
    $\{W_i, f_i, f'_i\} \xleftarrow[\text{each time}]{\text{one query}} \mathcal{A}(\alpha, \gamma, \mathbf{c}, \tau_1, \ldots \tau_{i-1}, c_1, \ldots c_{i-1})$
    $\tau_i \overset{\mathcal{A}}{\leftarrow} \text{SrchToken}(K, W_i)$, or
    $(U : st';\ \mathcal{A} : \tau_i, c_i) \overset{\mathcal{A}}{\leftarrow} \text{Add/Update}(U : K, \delta_f, f, st;\ \mathcal{A})$,
    or $(U : st';\ \mathcal{A} : \tau_i) \overset{\mathcal{A}}{\leftarrow} \text{Del/Update}(U : K, \delta_f, f, st;\ \mathcal{A})$
  output $b \leftarrow \mathcal{A}(\alpha, \gamma, \mathbf{c}, \tau_1, \ldots, \tau_q, c_1, \ldots c_q)$
$\text{Ideal}_{\mathcal{A},\mathcal{S}}(1^k)$ :
  $(\delta, \mathbf{f}) \leftarrow \mathcal{A}(1^k)$
  $(\tilde{\alpha}, \tilde{\gamma}, \tilde{\mathbf{c}}) \leftarrow \mathcal{S}^{\mathcal{L}_1(\delta,\mathbf{f})}(1^k)$
  for $1 \le i \le q$
    $\{W_i, f_i, f'_i\} \xleftarrow[\text{each time}]{\text{one query}} \mathcal{A}(\tilde{\alpha}, \tilde{\gamma}, \tilde{\mathbf{c}}, \tilde{\tau}_1, \ldots \tilde{\tau}_{i-1}, \tilde{c}_1, \ldots \tilde{c}_{i-1})$
    $\tilde{\tau}_i \overset{\mathcal{A}}{\leftarrow} \mathcal{S}^{\mathcal{L}_2(\delta,\mathbf{f},W_i)}(1^k)$, or
    $(\mathcal{S} : st';\ \mathcal{A} : \tilde{\tau}_i, \tilde{c}_i) \overset{\mathcal{A}}{\leftarrow} \text{Add/Update}(\mathcal{S}^{\mathcal{L}_3(\delta,\mathbf{f},f_i)}(1^k);\ \mathcal{A})$,
    or $(\mathcal{S} : st';\ \mathcal{A} : \tilde{\tau}_i) \overset{\mathcal{A}}{\leftarrow} \text{Del/Update}(\mathcal{S}^{\mathcal{L}_4(\delta,\mathbf{f},f'_i)}(1^k);\ \mathcal{A})$
  output $b \leftarrow \mathcal{A}(\tilde{\alpha}, \tilde{\gamma}, \tilde{\mathbf{c}}, \tilde{\tau}_1, \ldots, \tilde{\tau}_q, \tilde{c}_1, \ldots \tilde{c}_q)$
```

Our scheme is $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$-secure against adaptive dynamic chosen-keyword attacks if for all PPT adversary$\mathcal{A}$, there exist a probabilistic polynomial time simulator $\mathcal{S}$ such that: $\left|\Pr\left[\text{Real}_{\mathcal{A}}(1^k) = 1\right] - \Pr\left[\text{Ideal}_{\mathcal{A}}, \mathcal{S}(1^k) = 1\right]\right| \le \text{negl}(1^k)$, where $\text{negl}(1^k)$ is a negligible function with input 1^k.

4.2.2 Unforgeability

We use game $\text{Forge}_{\mathcal{A}}(1^k)$ to describe our schemes unforgeability. In the unforgeability game, the adversary interacts with a user that honestly executes the scheme. User initializes his data structures using the data provided by the adversary. After making polynomial times queries, the adversary produces a set of keywords, a wrong search result and a proof to this result. If these outputs pass the users verification algorithm, the game outputs 1, otherwise it outputs 0. The unforgeability requires that, all PPT adversary have at most negligible probability to let the game output 1. We give the formal definition as follow.

Definition 3 Given the scheme described in Definition 1, for a stateful adversary $\mathcal{A}$, consider the following game:

```
$\mathrm{Forge}_{\mathcal{A}}(1^k)$ :
  $K \leftarrow \mathrm{Gen}(1^k)$
  $(\delta, \mathbf{f}) \leftarrow \mathcal{A}(1^k)$
  $(\gamma, \mathbf{c}, st, \alpha) \leftarrow \mathrm{Setup}(K, \delta, \mathbf{f})$
  for  $1 \le i \le q$
    $\{W_i, f_i, f'_i\} \xleftarrow[\text{each time}]{\text{one query}} \mathcal{A}(\alpha, \gamma, \mathbf{c}, \tau_1, \ldots \tau_{i-1}, c_1, \ldots c_{i-1})$
    $\tau_i \overset{\mathcal{A}}{\longleftarrow} \mathrm{SrchToken}(K, W_i)$, or
    $(\tau_i, c_i) \overset{\mathcal{A}}{\longleftarrow} \mathrm{Add/Update}(U : K, \delta_{f_i}, f_i, st;\ \mathcal{A})$,
    or $\tau_i \overset{\mathcal{A}}{\longleftarrow} \mathrm{Del/Update}(U : K, \delta_{f'_i}, f'_i, st;\ \mathcal{A})$
  $(W, \mathbf{I}', \pi) \leftarrow \mathcal{A}(\alpha, \gamma, \mathbf{c}, \tau_1, \ldots, \tau_q, c_1, \ldots c_q)$
  $\tau_s \leftarrow \mathrm{SrchToken}(K, W)$
  output $b \leftarrow \mathrm{Verify}(K, st', \tau_s, \mathbf{I}', \pi)$
```

where the set $\mathbf{I}' \neq \mathbf{I}_W$. We say the our scheme is unforgeable if for all PPT adversary $\mathcal{A}$, the probability:

$$\Pr[\mathrm{Forge}_{\mathcal{A}}(1^k) = 1] \le \mathrm{negl}(1^k),$$

where $\mathrm{negl}(1^k)$ is a negligible function with input 1^k.

5 Integrity-verifiable conjunctive keyword searchable encryption scheme

In this section, we first construct a conjunctive keyword searchable encryption scheme, and then add the search authentication mechanism to it to make the search results integrity-verifiable.

In our construction, the set of files $\mathbf{f}$ along with the inverted indexes δ are the initial input. In contrast to the file index, an inverted index is a set of lists that lead by keywords, and each keyword is followed by a set of files that contain that keyword. The keywords of each file are preselected and can be considered as the outputs of some other algorithms, which wont be discussed here.

5.1 Dynamic searchable encryption

In the literature, most searchable encryption schemes use symmetric encryption to improve performance. We follow the prior constructions and build our scheme upon the CPA-secure private key encryption. Figure 2 shows the structure that constructed based on the inverted index. The lookup table contains all the keywords in the system, and each keyword in the table leads a list that stored in the search array. For example, the list of keyword w_2 starts at address 4 in the array, and the node at address 4 has a pointer that points to address 7, and then address 8. By traversing this list, all files that contain the keyword w_3 can be retrieved. All the nodes are stored at random location in the search array. To support efficient file updating, there are also a deletion table and a deletion array. They work the same way, except those lists are led by files.

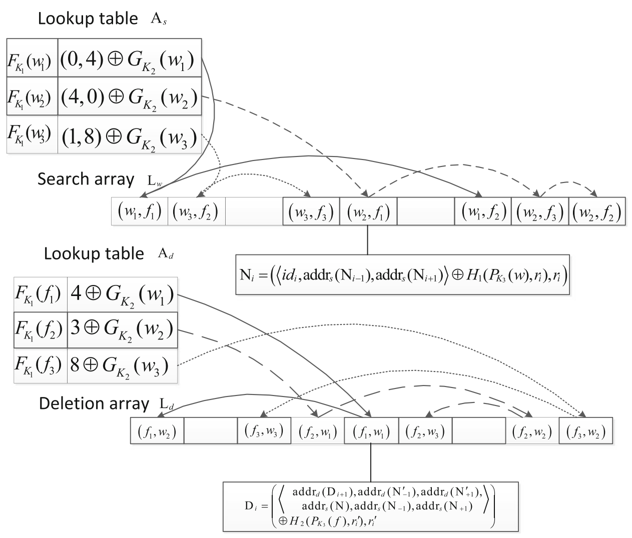
Lookup table A_s
$F_{K_1}(w_1)$ | $(0,4) \oplus G_{K_2}(w_1)$
$F_{K_1}(w_2)$ | $(4,0) \oplus G_{K_2}(w_2)$
$F_{K_1}(w_3)$ | $(1,8) \oplus G_{K_2}(w_3)$
Search array L_w
(w_1, f_1) (w_3, f_2) (w_3, f_3) (w_2, f_1) (w_1, f_2) (w_2, f_3) (w_2, f_2)
$N_i = (\langle id_i, \mathrm{addr}_s(N_{i-1}), \mathrm{addr}_s(N_{i+1})\rangle \oplus H_1(P_{K_3}(w), r_i), r_i)$
Lookup table A_d
$F_{K_1}(f_1)$ | $4 \oplus G_{K_2}(w_1)$
$F_{K_1}(f_2)$ | $3 \oplus G_{K_2}(w_2)$
$F_{K_1}(f_3)$ | $8 \oplus G_{K_2}(w_3)$
Deletion array L_d
(f_1, w_2) (f_3, w_3) (f_2, w_1) (f_1, w_1) (f_2, w_3) (f_2, w_2) (f_3, w_2)
$D_i = (\langle \mathrm{addr}_d(D_{i+1}), \mathrm{addr}_d(N'_{-1}), \mathrm{addr}_d(N'_{+1}), \mathrm{addr}_s(N), \mathrm{addr}_s(N_{-1}), \mathrm{addr}_s(N_{+1})\rangle \oplus H_2(P_{K_3}(f), r'_i), r'_i)$

Fig. 2 The schematic search structure

In order to prevent the server from learning the data, all the tables entry, all the pointers in the table, and all the nodes in those arrays are encrypted. During a search, given the encrypted keywords, the server first decrypts the pointers in the lookup table and then uses the pointers to find the corresponding file identifiers in the search array. Those keywords remain encrypted throughout the search. Even if the server has searched all those keywords, it can only learn the relationship between the encrypted keywords and the related file identifiers but cannot obtain any useful knowledge about the keywords itself. This could prevent the curious server from learning the files and keywords.

We use pseudo-random functions, collision-resistant hash functions and XOR operations to encrypt the data inside the structure. Therefore, the XORs homomorphic nature enables the server to update the encrypted nodes without decrypting.

Specifically, the construction works as follows. Let F, G and P be pseudo-random functions, H_1 and H_2 be two collision-resistant hash functions, and $K = (K_1, K_2, K_3)$ be the private key generated by user. To encrypt a collection of files $\mathbf{f}$ and the inverted index δ, we construct a list $\mathrm{L}_w$for each keyword $w \in \mathbf{w}$. Each list is composed of $\#\mathbf{f}_w$ nodes $(\mathrm{N}_1, \ldots, \mathrm{N}_{\#\mathbf{f}_w})$, and is stored at random locations in the search array A_s. The node is defined as $\mathrm{N}_i = \langle id_i, \mathrm{addr}_s(\mathrm{N}_{i-1}), \mathrm{addr}_s(\mathrm{N}_{i+1})\rangle$, where id_i is the unique file identifier of a file that contains w. The $\mathrm{addr}_s(\mathrm{N})$ denotes the location of node N in A_s. Each node in L_w is encrypted using $H_1(P_{K_3}(w), r_i)$, where r_i is a random value, and K_3 is the key of the PRF P. The unused nodes in A_s are padded with random bits.

For each keyword w, the lookup table T_s stores the encrypted pointer, which points to the head of L_w , along the search key $F_{K_1}(w)$, where K_1 is the key of the PRF F. The pointer is encrypted using $G_{K_2}(w)$, where K_2 is the key of the PRF G. The array A_s and the table T_s are regarded as the encrypted indexes and are stored at the server side. To search with some keywords $W = \{w_1, \ldots w_n\}$, the user needs to form a search token τ_s and send to the server. The τ_s includes $F_{K_1}(w_i)$, $G_{K_2}(w_i)$, $P_{K_3}(w_i)$ for all $w_i \in W$. The server uses $F_{K_1}(w_i)$ as search key in T_s to find the encrypted pointer, and uses $G_{K_2}(w_i)$ to recover the pointer to find the head of L_{w_i}, and uses $P_{K_3}(w_i)$ and the value r stored in each nodes to decrypt the list, then recovers the identifiers set S_i of the files that contain w_i. In other word, S_i is equivalent to $\mathbf{f}_{w_i}$. For a series of sets $S_1, \ldots, S_{\#W}$, the server computes the intersection $\mathbf{I}$ and outputs it as the result.

To dynamically add or delete a file, the server needs an deletion array A_d that stores for each file f a list L_f of nodes $(\mathrm{D}_1, \ldots, \mathrm{D}_{\#f})$. Each node D_i is associated with a word w, and a corresponding node N in L_w. Let N_{+1} be the node after N in L_w, and N_{-1} be the node before N in L_w. The node D_i is defined as $\mathrm{D}_i = \langle \mathrm{addr}_d(\mathrm{D}_{i+1}), \mathrm{addr}_d(\mathrm{N'}_{-1}), \mathrm{addr}_d(\mathrm{N'}_{+1}), \mathrm{addr}_s(\mathrm{N}), \mathrm{addr}_s(\mathrm{N}_{-1}), \mathrm{addr}_s(\mathrm{N}_{+1})\rangle$, where $\mathrm{N'}_i$ refers to the dual node of N_i, i.e., the node in A_d that corresponds to the node N_i. For example, in Fig. 3, the node at address 4 in A_s has a dual node at address 3 in A_d. Each node in L_f is encrypted using $H_2(P_{K_3}(f), r'_i)$, where r'_i is a random value, and K_3 is the key to the PRF P. Similarly, for each f, the deletion table T_d stores the pointer to the head of L_f under the search key $F_{K_1}(f)$, and the pointer is encrypted using $G_{K_2}(f)$. While adding or deleting files, user forms add token τ_a or delete token τ_d and sends to the server. The server uses the data in the token to update those pointers directly using XOR without decrypting them. In this construction, the storage occupied by the search structure is reasonably small in practice. It is the file identifiers that are stored in these tables and arrays, not the files themselves; likewise the keywords. Besides, with the amount of files increasing, the whole structure may indeed become large. However, in the search procedure, those nodes are located based on the pointers, which means the size of this structure does not affect the search's performance.

5.2 Making result verifiable

In the following content, we discuss the method to make result verifiable. This method can allow a server to prove to a client that it answered a conjunctive keyword search query correctly.

The method proposed in [11] is a Merkle tree based solution that it computes the accumulated value for each word w, and uses these values as leaves to construct the tree. In a search, the server returns a file set S, and a Merkle tree proof to this set. The user can compute his own accumulated value using the files in S, and use it to perform the Merkle tree verification. If the newly computed root equals to the original one, then the result is correct and can be accepted by the user.

However, while switching to the conjunctive keyword setting, this solution is obsolete to prove the correctness of the intersection of the results. The server could only generate the proof for each set separately. These sets and proofs must be transferred to the user side to be verified, and subsequently the intersection of these sets could be computed by the user. Obviously, the communication complexity is linear and may have performance problems when the sets are very large.

The reasonable way to address this problem is to let the server compute the intersection, and give the user final result directly. In this case, the correctness of the intersection operation should be proved. We use the bilinear map accumulator to realize this functionality. Intuitively, the correctness of the intersection could be defined as follows: given a set $\mathbf{I}$ and a series of sets $S_1, \ldots, S_n$, $\mathbf{I}$ is the correct intersection of $S_1, \ldots, S_n$ if and only if the following conditions hold:

1. The subset condition: $(\mathbf{I} \subseteq S_1) \wedge \cdots \wedge (\mathbf{I} \subseteq S_n)$.
2. The completeness condition: $(S_1 - \mathbf{I}) \cap \cdots \cap (S_n - \mathbf{I}) = \emptyset$.

The subset condition is easy to understand, because as the intersection, the set $\mathbf{I}$ must be included in each set S_i. We use Merkle tree to authenticate the value $\mathrm{acc}(S_i)$. For all $w \in \mathbf{w}$, the values $\mathrm{acc}(\mathbf{f}_w)$ are computed according to $\mathrm{acc}(\mathcal{X}) = g^{\prod_{x\in\mathcal{X}}(x+s)}$, then the tree is constructed using these values as leaves.

Since the user does not store those accumulated values, the server should first generates Merkle tree proofs for each $\mathrm{acc}(S_i)$. Its then straightforward to produce the subset witness $(\mathbf{I}, \mathcal{W}_{\mathbf{I},S_i})$ in $\mathcal{W}_{\mathcal{S},\mathcal{X}} = g^{\prod_{x\in\mathcal{X}-\mathcal{S}}(x+s)}$ for each set S_i. Given the $\mathrm{acc}(S_i)$ and the witness $(\mathbf{I}, \mathcal{W}_{\mathbf{I},S_i})$, the validity of the value $\mathrm{acc}(S_i)$ should be first verified using Merkle tree proofs, then the subset containment relationship could be checked by performing the verifications according to $e\left(\mathcal{W}_{\mathcal{S},\mathcal{X}}, g^{\prod_{x\in\mathcal{S}}(x+s)}\right) = e\left(\mathrm{acc}(\mathcal{X}), g\right)$.

The completeness condition is also necessary since the set $\mathbf{I}$ must contain all the common elements. To construct the completeness proof, we define the polynomial:

$$P_i(s) = \prod\nolimits_{f\in S_i-\mathbf{I}} (s + id(f)).$$

The following result is based on the extended Euclidean algorithm over polynomials and provides verification for checking the completeness of set intersection.

Lemma 1 *The set* $\mathbf{I}$ *is complete if and only if there exist polynomials* $q_1(s), \ldots, q_n(s)$ *such that* $q_1(s)P_1(s) + \cdots + q_n(s)P_n(s) = 1$, *where* $P_i(s)$ *is defined in (5).*

Suppose $\mathbf{I}$ is not the complete set, then there exist at least one common factor in $P_1(s), \ldots, P_n(s)$. Thus there are no

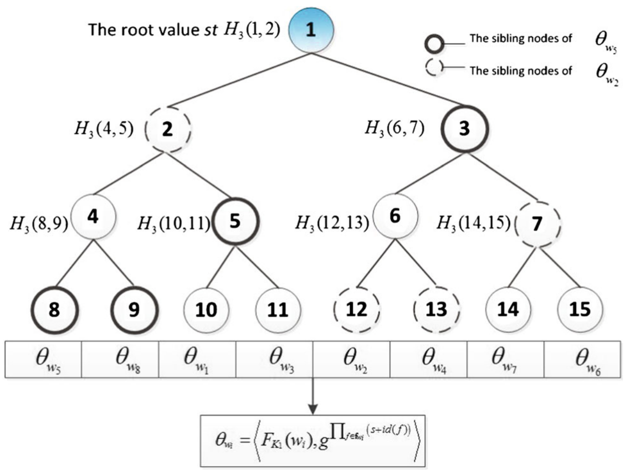
The root value st $H_3(1,2)$
The sibling nodes of θ_{w_5}
The sibling nodes of θ_{w_2}
$H_3(4,5)$
$H_3(6,7)$
$H_3(8,9)$
$H_3(10,11)$
$H_3(12,13)$
$H_3(14,15)$
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
θ_{w_5} θ_{w_8} θ_{w_1} θ_{w_3} θ_{w_2} θ_{w_4} θ_{w_7} θ_{w_6}
$\theta_{w_i} = \left\langle F_{K_1}(w_i), g^{\prod_{f \in \mathbf{f}_{w_i}}(s+id(f))} \right\rangle$

Fig. 3 An illustrative Merkle tree

polynomials $q_1(s), \ldots, q_n(s)$ to satisfy $q_1(s)P_1(s) + \cdots + q_n(s)P_n(s) = 1$. The formal proof is given in Sect. 6.

- An illustrative example

Figure 3 is an example that shows a Merkle tree with eight keywords. The leaf nodes θ_{w_i} are composed of $F_{K_1}(w)$ and $g^{\prod_{f \in \mathbf{f}_w}(s+id(f))}$, and permuted in a random order. All other nodes are the hash values of their children nodes using a collision-resistant hash function H_3. We denote st as the root value. The tree is stored at the server side, and the user only needs to store the root value st. When the user generates a conjunctive keyword search query to retrieve the files that contains the keywords w_2 and w_5, he first generates the search token $\tau_s = (\tau_2, \tau_5)$ that contains $F_{K_1}(w_2)$ and $F_{K_1}(w_5)$as search keys, and then send it to the server.

After the server received the search token, it first retrieves the two file sets $S_2 = \{id_1, \ldots, id_t\}$ and $S_5 = \{id_{1'}, \ldots, id_{t'}\}$ that contains the keywords w_2 and w_5 separately. Then let $\mathbf{I}_W = S_1 \cap S_2$ be the intersection of search results, and the server computes the proof $p_i = \{\mathcal{T}, \mathcal{S}, \mathcal{C}\}$ in the following steps:

1. Finds the leaves θ_2, θ_5 in the tree whose first elements are $F_{K_1}(w_2)$ and $F_{K_1}(w_5)$ in the search token, generates the proofs t_2, t_5, which include θ_2, θ_5 and all the sibling nodes $\mathcal{T} = \{t_2, t_5\}$in the path from the leaf to the root.
2. Compute the subset witness $\mathcal{S} = \{g^{P_2}, g^{P_5}\}$ using public parameters $(g, g^s, \ldots, g^{s^q})$, where $P_i = \prod_{f \in S_i - \mathbf{I}_W}(s + id(f))$. Then finds the polynomial q_2, q_5 that satisfying $q_2P_2 + q_5P_5 = 1$. Let $\mathcal{C} = \{g^{q_2}, g^{q_5}\}$ be the completeness witness.

Given the search result $\mathbf{I}_w$ and a proof $p_i = \{\mathcal{T}, \mathcal{S}, \mathcal{C}\}$, user verifies as follows:

1. Since θ_2, θ_5 are the leaf nodes in $\mathcal{T} = \{t_2, t_5\}$. Parse θ_2 as $(\theta_{2,1}, \theta_{2,2})$, and$\theta_5$ as $(\theta_{5,1}, \theta_{5,2})$, verify if $\theta_{2,1} = F_{K_1}(w_2)$ and in the search token. Then using the root value st , verify the proofst_2 and $t_5$are both composed of the sibling nodes in the path from the two leaves to the root.
2. Perform the subset condition verification by checking:
$$e\left(g^{\prod_{k=1}^{m}(s+id_k)}, g^{P_2}\right) \overset{?}{=} e\left(\theta_{2,2}, g\right),$$
$$e\left(g^{\prod_{k=1}^{m}(s+id_k)}, g^{P_5}\right) \overset{?}{=} e\left(\theta_{5,2}, g\right)$$
where $s \in \mathbf{Z}_p^*$ is the trapdoor, $(id_1, \ldots id_m)$ is from $\mathbf{I}_w$ and g^{P_2}, g^{P_5} are elements in $\mathcal{S}$.
3. Verify the completeness condition by checking:
$$e(g^{P_2}, g^{q_2}) \overset{?}{=} e(g^{P_5}, g^{q_5}) \overset{?}{=} e(g, g),$$
where g^{P_i} are elements in $\mathcal{S}$ and g^{q_i} are the corresponding elements in $\mathcal{C}$.

If all the verifications succeed, then the search result $\mathbf{I}_w$ is the correctness of the intersection of the results, and can be accepted by the user.

5.3 Explicit construction

The explicit construction is given as follows:

- Parameter initialization

Let Π = (Gen, Enc, Dec) be a private key encryption system. $F : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$, $G : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^*$, $P : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ be pseudo-random functions. Let $H_1 : \{0,1\}^* \rightarrow \{0,1\}^*$, $H_2 : \{0,1\}^* \rightarrow \{0,1\}^*$ and $H_3 : \{0,1\}^* \rightarrow \{0,1\}^k$ be collision-resistant hash functions. Let $(p, \mathbf{G}, \mathcal{G}, e, g)$ be the initial size of the free list, and $\mathbf{0}$ be a series of 0.

Choose bilinear pairing parameters $(p, \mathbf{G}, \mathcal{G}, e, g)$.

- **Gen($\mathbf{1^k}$)**

Randomly choose three $k-$bit strings K_1, K_2, K_3 and generate $K_4 \leftarrow \Pi.\text{Gen}(1^k)$. Choose at random and output $K = (K_1, K_2, K_3, K_4, s)$ as the private keys. Compute $(g, g^s, g^{s^2}, \ldots, g^{s^q})$ as public parameters where q should be large enough, i.e., should at least satisfy $q \geq \max\{\#\mathbf{f}_w\}_{w \in \mathbf{w}}$.

- **Setup($\mathbf{K}, \delta, \mathbf{f}$)**

1. Let A_s and A_d be arrays of size $|\mathbf{c}|/8 + z$ and let T_s and T_d be dictionaries of size $\#\mathbf{w}$ and $\#\mathbf{f}$, respectively. Use

free to represent a $k-$length word not in $\mathbf{w}$.The following step 2 and step 3 should be performed synchronously to set up A_s and A_d at the same time.

2. For every keyword $w \in \mathbf{w}$,

   (a) Generate a list L_w of $\#\mathbf{f}_w$ nodes $(N_1, \ldots, N_{\#\mathbf{f}_w})$ randomly stored in A_s, which are defined as

   $$N_i = (\langle id_i, \mathrm{addr}_s(N_{i-1}), \mathrm{addr}_s(N_{i+1})\rangle \oplus H_1(P_{K_3}(w), r_i), r_i),$$

   where id_i is the identity of the $i$thfile in $\mathbf{f}_w$, r_i is a $k-$bit string which is generated uniformly at random, and $\mathrm{addr}_s(N_{\#\mathbf{f}_w+1}) = \mathrm{addr}_s(N_0) = \mathbf{0}^{\log \# A_s}$.

   (b) Store a pointer to the first node of L_w in the search table by setting: $T_s\left[F_{K_1}(w)\right] = \langle \mathrm{addr}_s(N_1), \mathrm{addr}_d(N'_1)\rangle \oplus G_{K_2}(w)$, where N'_1 is the dual of N_1 in the list L_f, which has the same (f_1, w) pair as node N_1.

3. For each file f in $\mathbf{f}$,

   (a) Create a list L_f of $\#f$ dual nodes $(D_1, \ldots, D_{\#f})$ $(N_1, \ldots, N_{\#\mathbf{f}_w})$ randomly stored in the deletion array A_d. Each node D_i is associated with a word w, and a corresponding node N in L_w. Let N_{+1} be the node after N in L_w, and N_{-1} be the node before N in L_w. Define D_i as:

   $$D_i = \left(\begin{array}{l}\left\langle\begin{array}{l}\mathrm{addr}_d(D_{i+1}), \mathrm{addr}_d(N'_{-1}), \mathrm{addr}_d(N'_{+1}),\\ \mathrm{addr}_s(N), \mathrm{addr}_s(N_{-1}), \mathrm{addr}_s(N_{+1})\end{array}\right\rangle\\ \oplus H_2(P_{K_3}(f), r'_i), r'_i\end{array}\right),$$

   where r'_i is a random $k-$bit string, $\mathrm{addr}_d(D_{\#f+1}) = \mathbf{0}^{\log \# A_d}$.

   (b) Store a pointer to the first node of L_f in the deletion table by setting:

   $$T_d\left[F_{K_1}(f)\right] = \mathrm{addr}_d(D_1) \oplus G_{K_2}(f).$$

   (c) Generate the free list L_{free} by choosing z unused cells at random in A_s and in A_d. Let $(F_1, \ldots, F_z)$ and $(F'_1, \ldots, F'_z)$ be the free nodes in A_s and A_d, respectively. Set:

   $$T_s\,[\mathrm{free}] = \left\langle \mathrm{addr}_s(F_1), \mathbf{0}^{\log \# A_s}\right\rangle,$$

   and for $1 \le i \le z$, set

   $A_s\,[\mathrm{addr}_s(F_i)] = \left\langle \mathbf{0}^{\log \#\mathbf{f}}, \mathrm{addr}_s(F_{i+1}), \mathrm{addr}_d(F'_i), \mathbf{0}^k\right\rangle$, where $\mathrm{addr}_s(F_{z+1}) = \mathbf{0}^{\log \# A_s}$.

4. Fill the remaining entries of A_s and A_d with random strings.
5. For $1 \le i \le \#\mathbf{f}$, let $c_i \leftarrow \Pi.\mathrm{Enc}_{K_4}(f_i)$.
6. For all $w \in \mathbf{w}$, form the leaf node by letting

   $$\theta_w = \left\langle F_{K_1}(w), g^{\prod_{f\in\mathbf{f}_w}(s+id(f))}\right\rangle.$$

   Construct a Merkle tree using H_3 with leaves $\mathcal{L} = \{\theta_w\}_{w\in\mathbf{w}}$ permuted in a random order.
7. Output $(\gamma, \mathbf{c}, st, \alpha)$, where $\gamma = (A_s, T_s, A_d, T_d)$,$\mathbf{c} = (c_1, \ldots, c_{\#\mathbf{f}})$, st is the root of the tree, and α is the tree itself.

- **SrchToken**$(\mathbf{K}, \mathbf{W})$

For $W = (w_1, \ldots, w_n)$, compute each $\tau_i = (F_{K_1}(w_i), G_{K_2}(w_i), P_{K_3}(w_i))$, and then output $\tau_s = (\tau_1, \ldots \tau_n)$.

- **Search**$(\alpha, \gamma, \mathbf{c}, \tau_s)$

1. For each τ_i in τ_s, parse τ_i as $(\tau_{i,1}, \tau_{i,2}, \tau_{i,3})$,

   (a) Recover a pointer to the first node of the list by computing $(\alpha_1, \alpha'_1) = T_s[\tau_{i,1}] \oplus \tau_{i,2}$.

   (b) Lookup node $N_1 = A[\alpha_1]$ and decrypt it using $\tau_{i,3}$, i.e., parse N_1 as (v_1, r_1) and compute $(id_1, \mathbf{0}$, add $r_s(N_2)) = v_1 \oplus H_1(\tau_{i,3}, r_1)$. Let $\alpha_2 = \mathrm{addr}_s(N_2)$.

   (c) For $j \ge 2$, decrypt node N_j as above until $\alpha_{j+1} = \mathbf{0}$.

   (d) Let $S_i = \{id_1, \ldots, id_t\}$ be the file identifiers revealed in the previous steps.

2. For the sets $S_1, \cdots, S_n$ generated in step 1, let $\mathbf{I}_W = \{id_1, \ldots, id_m\}$ be the intersection, i.e., $\mathbf{I}_W = S_1 \cap S_2 \cap \ldots \cap S_n$. Compute the proofs in the following steps:

   (a) For $1 \le i \le n$, find the leaf θ_i in α whose first element is $\tau_{i,1}$ and generate the proof t_i. The t_i includes θ_i and all the sibling nodes in the path from the leaf θ_i to the root. Let $\mathcal{T} = \{t_1, \ldots, t_n\}$.

   (b) For $1 \le i \le n$, form the polynomial:

   $$P_i = \prod_{f\in S_i - \mathbf{I}_W}(s + id(f)),$$

   then use the public parameters $(g, g^s, g^{s^2}, \ldots, g^{s^q})$ to compute the value g^{P_i}. Let $\mathcal{S} = \{g^{P_1}, \ldots, g^{P_n}\}$ be the subset witness.

   (c) Giving the polynomials $\{P_1, \ldots, P_n\}$ generated in step (b), find the polynomials $\{q_1, \ldots, q_n\}$ that satisfying $q_1P_1 + q_2P_2 + \cdots + q_nP_n = 1$. This can be done using extended Euclidean algorithm over polynomials. Let $\mathcal{C} = \{g^{q_1}, \ldots, g^{q_n}\}$ be the completeness witness.

3. Output the result $\mathbf{I}_W$ and the proof $\pi = \{\mathcal{T}, \mathcal{S}, \mathcal{C}\}$.

- **Verify**$(\mathbf{K}, \mathbf{st}, \tau_s, \mathbf{I}', \pi)$

1. Parse π as $\{\mathcal{T}, \mathcal{S}, \mathcal{C}\}$and verify these proofs in the following steps:

   (a) For each proof t_i in $\mathcal{T}$, let θ_i be the corresponding leaf node in t_i. Parse θ_i as $(\theta_{i,1}, \theta_{i,2})$, i.e., $\theta_{i,1} = F_{K_1}(w_i)$

and $\theta_{i,2} = g^{\prod_{f\in \mathbf{f}_{w_i}}(s+id(f))}$. Verify if the value $\theta_{i,1}$ equals to $\tau_{i,1}$, where $\tau_{i,1}$ is the first element of τ_i in τ_s. Then verify the proof t_i using the root st.

(b) For $1 \le i \le n$, parse the leaf node θ_i in step (a) as $(\theta_{i,1}, \theta_{i,2})$, then perform the subset condition verification by checking:

$$e\left(g^{\prod_{k=1}^{m}(s+id_k)}, g^{P_i}\right) \overset{?}{=} e\left(\theta_{i,2}, g\right),$$

where $(id_1, \ldots id_m)$ is from $\mathbf{I}'$ and g^{P_i} is element in $\mathcal{S}$.

(c) Verify the completeness condition by checking:

$$\prod_{i=1}^{n} e(g^{P_i}, g^{q_i}) \overset{?}{=} e(g, g),$$

where g^{P_i} is element in $\mathcal{S}$ and g^{q_i} is the corresponding element in $\mathcal{C}$.

2. If all the verifications succeed, then output 1; otherwise output 0.

- **Dec**($\mathbf{K}, \mathbf{c}$): Output $f = \Pi.\mathrm{Dec}_{K_4}(c)$.

- **Add/Update**($\mathbf{U} : \mathbf{K}, \delta_{\mathbf{f}}, \mathbf{f}, \mathbf{st};\ \mathbf{S} : \alpha, \gamma, \mathbf{c}$):

User:
Recover the unique sequence of words $(w_1, \ldots, w_{\#f})$ from δ_f and compute the set $\{F_{K_1}(w_i)\}_{1\le i\le \#f}$ and send to the server.
Server:

1. For $1 \le i \le \#f$, traverse the Merkel tree α and:
   (a) Find the leaf θ_i in α whose first element is $F_{K_1}(w_i)$
   (b) Let t_i be the proof in α from θ_i to the root. The proof includes the leaf θ_i, and all the sibling nodes from θ_i to the root.
2. Let $\rho = (t_1, \ldots, t_{\#f})$ and send it to the user.

User:

1. Verify the proofs in $(t_1, \ldots, t_{\#f})$ using st, if fails, output $\perp$ and terminate.
2. For $1 \le i \le \#f$,
   (a) Let θ_i be the leaf in t_i, parse θ_i as $(\theta_{i,1}, \theta_{i,2})$.
   (b) Compute the new leaf node $\theta'_i = (\theta_{i,1}, (\theta_{i,2})^{s+id(f)})$.
3. Update the root hash st using $(\theta'_1, \ldots, \theta'_{\#f})$ and the information in $(t_1, \ldots, t_{\#f})$.
4. Compute $\tau_a = (F_{K_1}(f), G_{K_2}(f), \lambda_1, \ldots \lambda_{\#f})$, where for all $1 \le i \le \#f$:

$$\lambda_i = \begin{pmatrix} \theta'_{i,1}, \theta'_{i,2}, G_{K_2}(w_i), \langle id(f), \mathbf{0}, \mathbf{0}\rangle \oplus H_1(P_{K_3}(w_i), r_i), \\ r_i, \langle \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}\rangle \oplus H_2(P_{K_3}(f), r'_i), r'_i \end{pmatrix},$$

where r_i and r'_i are random $k-$bit strings.
5. Let $c_f \leftarrow \Pi.\mathrm{Enc}_{K_4}(f)$ and send (τ_a, c_f) to the server, then output the new root st'.

Server:

1. Parse τ_a as $(\tau_1, \tau_2, \lambda_1, \ldots, \lambda_{\#f})$ and return $\perp$ if τ_1 is already in T_d.
2. For $1 \le i \le \#f$,
   (a) Find the first free location φ in A_s, second free location φ_{+1} in A_s, first free location φ' in A_d, and second free location φ'_{+1} in A_d, by computing $(\varphi, \mathbf{0}) = \mathrm{T}_s[\text{free}]$, $(\mathbf{0}, \varphi_{+1}, \varphi') = \mathrm{A}_s[\varphi]$ and $(\mathbf{0}, \varphi_{+2}, \varphi'_{+1}) = \mathrm{A}_s[\varphi_{+1}]$.
   (b) Update the search table to point to the second free entry by setting $\mathrm{T}_s[\text{free}] = (\varphi_{+1}, \mathbf{0})$.
   (c) Recover the first node N_1s address α_1 by computing $(\alpha_1, \alpha'_1) = \mathrm{T}_s[\lambda_i[1]] \oplus \lambda_i[3]$.
   (d) Parse $\mathrm{N}_1 = \mathrm{A}_s[\alpha_1]$ as (v_1, r_1), then update N_1s back pointer point to the new node by setting:
   $$\mathrm{A}_s[\alpha_1] = (v_1 \oplus \langle \mathbf{0}, \varphi, \mathbf{0}\rangle, r_1).$$
   (e) Store the new node at location φ and modify its forward pointer to N_1 by setting:
   $$\mathrm{A}_s[\varphi] = (\lambda_i[4] \oplus \langle \mathbf{0}, \mathbf{0}, \alpha_1\rangle, \lambda_i[5]).$$
   (f) Update the search table by setting:
   $$\mathrm{T}_s\,[\lambda_i[1]] = (\varphi, \varphi') \oplus \lambda_i[3].$$
   (g) Parse $\mathrm{D}_1 = \mathrm{A}_d[\alpha'_1]$ as (v'_1, r'_1), update the dual of N_1 by setting $\mathrm{A}_d[\alpha'_1] = (v'_1 \oplus \langle \mathbf{0}, \varphi', \mathbf{0}, \mathbf{0}, \varphi, \mathbf{0}\rangle, r'_1)$.
   (h) If $i < \#f$, store the dual of $\mathrm{A}_s[\varphi]$ at position φ' by setting:
   $$\mathrm{A}_d[\varphi'] = \big(\lambda_i[6] \oplus \langle \varphi'_{+1}, \mathbf{0}, \alpha'_1, \varphi, \mathbf{0}, \alpha_1\rangle, \lambda_i[7]\big).$$
   If $i = \#f$, then set:
   $$\mathrm{A}_d[\varphi'] = \big(\lambda_i[6] \oplus \langle \mathbf{0}, \mathbf{0}, \alpha'_1, \varphi, \mathbf{0}, \alpha_1\rangle, \lambda_i[7]\big).$$
   (i) If $i = 1$, then update the deletion table by setting $\mathrm{T}_d[\tau_1] = \varphi' \oplus \tau_2$.
3. Update the ciphertexts by adding c to $\mathbf{c}$.
4. Let $\theta'_i = (\lambda_i[1], \lambda_i[2])$, update the tree α by replacing the leaves $(\theta_1, \ldots, \theta_{\#f})$ with $(\theta'_1, \ldots, \theta'_{\#f})$.
5. Output $(\alpha', \gamma', \mathbf{c}')$, where α' is the updated tree.

- **Del/Update**($\mathbf{U} : \mathbf{K}, \delta_{\mathbf{f}}, f, st;\ S : \alpha, \gamma, \mathbf{c}$)

User:
Recover the unique sequence of words $(w_1, \ldots, w_{\#f})$ from δ_f and compute the set $\{F_{K_1}(w_i)\}_{1\le i\le \#f}$ and send to the server.

Server:

1. For $1 \leq i \leq \#f$, traverse the Merkel tree α and:
   (a) Find the leaf θ_i in α whose first element is $F_{K_1}(w_i)$
   (b) Let t_i be the proof in α from θ_i to the root. The proof includes the leaf θ_i, and all the sibling nodes from θ_i to the root.
2. Let $\rho = (t_1, \ldots, t_{\#f})$ and send it to the user.

User:

1. Verify the proofs in $(t_1, \ldots, t_{\#f})$ using st, if fails, output $\perp$ and terminate.
2. For $1 \leq i \leq \#f$,
   (a) Let θ_i be the leaf in t_i, parse θ_i as $(\theta_{i,1}, \theta_{i,2})$.
   (b) Compute the new leaf $\theta'_i = (\theta_{i,1}, (\theta_{i,2})^{1/(s+id(f))})$.
3. Update the root hash st using $(\theta'_1, \ldots, \theta'_{\#f})$ and the information in $(t_1, \ldots, t_{\#f})$.
4. Compute $\tau_d = (F_{K_1}(f), G_{K_2}(f), P_{K_3}(f), id(f), \theta'_1, \ldots, \theta'_{\#f})$.
5. Send τ_d to the server, then output the new root st'.

Server:

1. Parse τ_d as $(\tau_1, \tau_2, \tau_3, id, \theta'_1, \ldots, \theta'_{\#f})$.
2. Find the first node of L_f by computing $\alpha'_i = \mathrm{T}_d[\tau_1] \oplus \tau_2$.
3. While $\alpha'_i \neq \mathbf{0}$,
   (a) Parse $\mathrm{D}_i = \mathrm{A}_d[\alpha'_i]$ as (v'_i, r'_i), decrypt D_i by computing $(\alpha_1, \ldots, \alpha_6) = v'_i \oplus H_2(\tau_3, r'_i)$.
   (b) Delete D_i by setting $\mathrm{A}_d[\alpha'_i]$ to a random string.
   (c) Find address of the first free node by computing $(\varphi, \mathbf{0}) = \mathrm{T}_s[\text{free}]$.
   (d) Update the first node of the free list in the T_s point to D_is dual by setting $\mathrm{T}_s[\text{free}] = (\alpha_4, \mathbf{0})$.
   (e) Free D_is dual by setting $\mathrm{A}_s[\alpha_4] = (\mathbf{0}, \varphi, \alpha'_i)$.
   (f) Let N_{-1} be the node before D_is dual. Update N_{-1}s next pointer by setting $\mathrm{A}_s[\alpha_5] = (\beta_1, \beta_2, \beta_3 \oplus \alpha_4 \oplus \alpha_6, r_{-1})$, where $(\beta_1, \beta_2, \beta_3, r_{-1}) = \mathrm{A}_s[\alpha_5]$. Also, update the pointers of N_{-1}s dual by setting:
   $$\mathrm{A}_d[\alpha_2] = (\beta_1, \beta_2, \beta_3 \oplus \alpha'_i \oplus \alpha_3, \beta_4, \beta_5, \beta_6 \oplus \alpha_4 \oplus \alpha_6, r'_{-1}),$$
   where $(\beta_1, \ldots, \beta_6, r'_{-1}) = \mathrm{A}_d[\alpha_2]$
   (g) Let N_{+1} be the node after D_is dual. Update N_{+1}s previous pointer by setting $\mathrm{A}_s[\alpha_6] = (\beta_1, \beta_2 \oplus \alpha_4 \oplus \alpha_5, \beta_3, r_{+1})$, where $(\beta_1, \beta_2, \beta_3, r_{+1}) = \mathrm{A}_s[\alpha_6]$. Also, update N_{+1}s duals pointers by setting:
   $$\mathrm{A}_d[\alpha_3] = (\beta_1, \beta_2 \oplus \alpha'_i \oplus \alpha_2, \beta_3, \beta_4, \beta_5 \oplus \alpha_4 \oplus \alpha_5, \beta_6, r'_{+1}),$$
   where $(\beta_1, \ldots, \beta_6, r'_{+1}) = \mathrm{A}_d[\alpha_3]$
   (h) Set $\alpha'_i = \alpha_1$.
4. Remove the cipher text corresponding to id from $\mathbf{c}$.
5. Remove τ_1 from T_d.
6. Update the tree α by replacing the leaves $(\theta_1, \ldots, \theta_{\#f})$ with $(\theta'_1, \ldots, \theta'_{\#f})$.
7. Output $(\alpha', \gamma', \mathbf{c}')$, where α' is the updated tree.

6 Security analysis

In this section we give an overview of the security property of our scheme and then give the mathematic proofs in detail.

6.1 Dynamic adaptive chosen-keyword security

As mentioned in Sect. 4, most of the known efficient searchable encryption algorithms more or less leak information. The extent to which the practical security of SSE is affected by this leakage is not well understood, and depends greatly on the setting in which SSE is used.

We summarize the information that is leaked in the scheme into leakage functions to describe the chosen-keyword security feature. Our goal is to prove that, any PPT adversary can obtain no information about the data and queries, except the information in the leakage functions.

In the following, we analyze our scheme and investigate which information has been leaked during the execution of these algorithms and protocols. The formal definition will be given afterward.

In our scheme, for each word , the value $F_{K_1}(w_i)$ can be treated as a unique identifier, and we denote it by $id(w_i)$. For each file f_i, there are two identifiers, the $id(f_i)$ in the array A_s and the $F_{K_1}(f_i)$ in the table T_d. Both of them can uniquely represent a file, so for convenience, we do not distinguish between them.

Given the encrypted index $\gamma = (\mathrm{T}_s, \mathrm{A}_s, \mathrm{T}_d, \mathrm{A}_d)$, the Merkle tree α and the ciphertexts $\mathbf{c}$, the server can learn the size of A_s, the set $[id(w)]_{w\in\mathbf{w}}$ from T_s, the set $[id(f)]_{f\in\mathbf{f}}$ and the length of each file $[|f|]_{f\in\mathbf{f}}$. We denote these by $\mathcal{L}_1$, i.e.,

$$\mathcal{L}_1(\delta, \mathbf{f}) = \left(\#\mathrm{A}_s, [id(w)]_{w\in\mathbf{w}}, [id(f)]_{f\in\mathbf{f}}, [|f|]_{f\in\mathbf{f}}\right).$$

The search operation reveals to the server $id(w)$ for all $w \in W$, and the relationship between $id(w)$ and the identifiers of all files that contains w. We denote these by $\mathcal{L}_2$, i.e.,

$$\mathcal{L}_2(\delta, \mathbf{f}, W) = \left([id(f)]_{f\in\mathbf{f}_w}, id(w)\right)_{\text{for all } w\in W}.$$

In the add protocol, the server can learn the identifier of the file to be added, the length of the file, and the identifiers of the words that belong to the file. In addition, it can tell whether the word w contained in the file is a new word by checking the table T_s. We denote these by $\mathcal{L}_3$, i.e.,

$$\mathcal{L}_3(\delta, \mathbf{f}, f) = \left(id(f), \left[id(w), \mathrm{apprs}(w)\right]_{w \in \mathbf{w}_f}, |f|\right),$$

where $\mathrm{apprs}(w)$ is a one bit flag set to 1 if the word w exists in the index before the file f is added; otherwise, it is set to 0.

Similarly, in the delete protocol, the server can learn the identifier of the file to be deleted, and know the relationship between $id(f)$ and those word identifiers. In addition, for each $w \in \mathbf{w}_f$, by removing the word pair (f, w) from the list L_w, the server learns the locations of the pairs neighbors in L_w. We denote these by $\mathcal{L}_4$.

$$\mathcal{L}_4(\delta, \mathbf{f}, f) = \\ \left(id(f), \left[id(w), \mathrm{prev}(f, w), \mathrm{next}(f, w)\right]_{w \in \mathbf{w}_f}\right),$$

where $\mathrm{prev}(f, w)$ and $\mathrm{next}(f, w)$ are the file identifiers of the file before and after f in the word list L_w. For the head and the tail of the list, the corresponding value is set $\perp$ to indicate that there are no more nodes before or after this one.

Now we use the following theorem to claim that the construction in Sect. 5 is dynamic CKA2-secure in the random oracle model with the leakage functions described above.

Theorem 1 *If the Π scheme is CPA-secure, and the F, G and P are pseudo-random functions, then our scheme is $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$-secure against adaptive chosen-keyword attacks in the random oracle model.*

Proof The primary goal of providing this proof is to construct a PPT simulator $\mathcal{S}$ that can generate the simulated values in the ideal game using the information given in these leakage functions. Those simulated values should be indistinguishable from ones in the real game to any PPT adversary.

Given the information received from $\mathcal{L}_1$, the simulator could determine the length and the structure of encrypted index γ, ciphertexts $\mathbf{c}$ and tree α. Then it can use randomly chosen strings to construct these structures and produce these values as the simulated one $(\tilde{\gamma}, \tilde{\mathbf{c}}, \tilde{\alpha})$. If a PPT adversary can distinguish the tuple $(\tilde{\gamma}, \tilde{\mathbf{c}}, \tilde{\alpha})$ from $(\gamma, \mathbf{c}, \alpha)$ with non-negligible probability then it can break at least one of these properties with non-negligible probability: the CPA security of the encryption scheme; the pseudo-randomness of the PRFs and the elliptic curve discrete logarithm assumption.

Given the information received from $\mathcal{L}_2$, $\mathcal{L}_3$ and $\mathcal{L}_4$, the simulator should respond the simulated search token, the simulated add token and the simulated delete token during the adversary's queries. These steps become more complex due to the fact that simulator needs to track the dependencies between the information revealed by these queries to ensure consistency among these simulated tokens. We define additional assisting structures iA_s, iA_d, iT_s and iT_d in the simulator side to maintain consistency during updation. The simulator uses these assisted structures to record those dependencies that are revealed by $\mathcal{L}_2$, $\mathcal{L}_3$ and $\mathcal{L}_4$ in the queries, and builds internal relationship in iA_s, iA_d, iT_s and iT_d, while the values in $\tilde{\gamma} = (\tilde{\mathbf{A}}_s, \tilde{\mathbf{T}}_s, \tilde{\mathbf{A}}_d, \tilde{\mathbf{T}}_d)$ remain random. This gives the simulator the ability to respond the adversary's queries like a real user, except using those simulated values.

The explicit proof is given as follows.

Let $\mathcal{S}$ represent the polynomial time simulator. The $\mathcal{S}'$s task is to make sure for all PPT adversaries $\mathcal{A}$, the outputs of $\mathrm{Real}_{\mathcal{A}}(k)$ and $\mathrm{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ are identical, except with negligible probability. The simulator $\mathcal{S}$ generates the simulated index $\tilde{\gamma} = (\tilde{\mathbf{A}}_s, \tilde{\mathbf{T}}_s, \tilde{\mathbf{A}}_d, \tilde{\mathbf{T}}_d)$, the simulated set of ciphertexts $\tilde{\mathbf{c}}$, the simulated Merkle tree $\tilde{\alpha}$ and a sequence of tokens $(\tilde{\tau}_1, \ldots, \tilde{\tau}_q)$ in the following way:

Initialization phase

Given $\mathcal{L}_1(\delta, \mathbf{f})$ defined above, the simulator generates $K_4 \leftarrow \Pi.\mathrm{Gen}(1^k)$ and chooses $s \leftarrow \mathrm{Z}_p^*$ at random. Let iA_s, $\tilde{\mathbf{A}}'_s$, iA_d and $\tilde{\mathbf{A}}'_d$ be empty arrays of size $\#\mathrm{A}_s$, iT_s and $\tilde{\mathbf{T}}'_s$ be lookup table of size $\#\mathbf{w} + 1$, iT_d and $\tilde{\mathbf{T}}'_d$ be lookup table of size $\#\mathbf{f}$, G be an empty key table of size $\#\mathbf{f}$ used for add and delete.

Sets $\mathrm{iT}_s[\mathrm{free}]$ to $\perp$. For all $w \in \mathbf{w}$, sets $\mathrm{iT}_s[id(w)]$ to $\perp$. For all $f \in \mathbf{f}$, sets $\mathrm{iT}_d[id(f)]$ to $\perp$. Let γ_s be a bijection mapping search keys in to search keys in $\tilde{\mathbf{T}}'_s$, γ_d be a bijection mapping search keys in iT_d to search keys in $\tilde{\mathbf{T}}'_d$. For each word identifier $id(w_i)$, randomly chooses a $k-$bit string $K_{id(w_i)}$ associates with it. For each file identifier $id(f_i)$, randomly chooses a string $K_{id(w_i)}$ associates with it.

(Simulating $\tilde{\mathbf{A}}_s$) it generates the array $\tilde{\mathbf{A}}_s$ of size $\#\mathrm{A}_s$, and fills all cells with random strings. Each cell can be recognized as the form $\langle \tilde{\mathbf{N}}, \tilde{r} \rangle$ where $|\tilde{\mathbf{N}}| = 2 \log \#\mathrm{A}_s + \log \#\mathbf{f}$ and $|\tilde{r}| = k$. Copies the content of $\tilde{\mathbf{A}}_s$ to $\tilde{\mathbf{A}}'_s$. Then randomly chooses z cells in $\tilde{\mathbf{A}}_s$ and marks as free in iA_s.

(Simulating $\tilde{\mathbf{T}}_s$) it generates the lookup table $\tilde{\mathbf{T}}_s$ of size $\#\mathbf{w} + 1$, and fills all cells with random strings. Each cell can be recognized as a $k-$bit search key $\tilde{\sigma}$ along with a $2 \log \#\mathrm{A}_s$ bit string $\tilde{v}$. Copies the content of $\tilde{\mathbf{T}}'_s$ to $\tilde{\mathbf{T}}'_s$.

(Simulating $\tilde{\mathbf{A}}_d$) it generates the array $\tilde{\mathbf{A}}_d$ of size $\#\mathrm{A}_s$, and fills all cells with random strings. Each cell can be recognized as the form $\langle \tilde{\mathbf{D}}, \tilde{r} \rangle$ where $|\tilde{\mathbf{D}}| = 6 \log \#\mathrm{A}_s$ and $|\tilde{r}| = k$. Copies the content of $\tilde{\mathbf{A}}_d$ to $\tilde{\mathbf{A}}'_d$. Then randomly chooses z cells in $\tilde{\mathbf{A}}_d$ and marks as free in iA_d. It then generates a bijection δ mapping all the cells in iA_s to all the cells in iA_d, such that $\delta(\perp) = \perp$ and free cells in iA_s and iA_d are mapped to each other.

(Simulating $\tilde{\mathbf{T}}_d$) it generates the lookup table $\tilde{\mathbf{T}}_d$ of size $\#\mathbf{f}$, and fills all cells with random strings. Each cell can be recognized as a $k-$bit search key $\tilde{\sigma}$ along with a $\log \#\mathrm{A}_s$ bit string $\tilde{v}$. Copies the content of $\tilde{\mathbf{T}}_d$ to $\tilde{\mathbf{T}}'_d$.

(Simulating $\tilde{\mathbf{c}}$) for all $f \in \mathbf{f}$, computes $\tilde{c}_f = \Pi.\mathrm{Enc}_{K_4}(\mathbf{0}^{|f|})$.

(Simulating $\tilde{\alpha}$) chooses $\#\mathbf{w}$ random number $(\omega_1, \ldots, \omega_{\#\mathbf{w}})$ from and computes $(g^{\omega_1}, \ldots, g^{\omega_{\#\mathbf{w}}})$. Then for all $w_i \in \mathbf{w}$, simulates the leaves by forming $\tilde{\theta}_{w_i} = \langle \gamma_s(id(w_i)), g^{\omega_i} \rangle$. Let $\tilde{\alpha}$ be the Merkle tree constructed using the leaves $\tilde{\mathcal{L}} = \{\tilde{\theta}_w\}_{w \in \mathbf{w}}$.

Query for search token

Given $\mathcal{L}_2(\delta, \mathbf{f}, W)$ defined above, for each w_i in W, the simulator needs to consider the following two cases:

Case 1 For the search key $id(w_i)$, the value $\mathrm{iT}_s[id((w_i)] = (\perp, \perp)$. This means the $id(w_i)$ has never appeared in previous queries, or there was no file attached to $id(w_i)$ before. Then the simulator needs to create a new list for $id(w_i)$. It randomly chooses $\#\mathbf{f}_{w_i}$ unused and non-free cells in and marks them with $id(w_i)$. It then stores $(id_1, \ldots id_{\#\mathbf{f}_{w_i}})$ in these cells, and updates each cells next pointer to form a list, according to the order given in $\mathcal{L}_2$. The lists last cells next pointer is set to $\perp$. Let α_1 be the address of the first cell in the list, the simulator updates iT_s by setting $\mathrm{iT}_s[id(w_i)] = \langle \alpha_1, \delta(\alpha_1) \rangle$. If $\mathrm{iT}_s[id(w_i)] = (\alpha_1, \delta(\alpha_1))$, then sets $\mathrm{iT}_s[id(w_i)] = (\perp, \perp)$.

Case 2 For the search key $id(w_i)$, the value $\mathrm{iT}_s[id((w_i)] = (\alpha_1, \delta(\alpha_1))$.This means there is already a list and has some file identifiers in it. Then the simulator needs to traverse the list from the cell located in α_1, and examine these file identifiers to find out the identifiers that in the set $(id_1, \ldots id_{\#\mathbf{f}_{w_i}})$ but not in the list. It then augments the list to the length $\#\mathbf{f}_{w_i}$ by randomly choosing unused and non-free cells in and storing the missing identifiers and pointers in these sells. It marks the new cells with $id(w_i)$.

For each w_i in W, by creating: $\tilde{\tau}_i = (\gamma_s(id(w_i)), \tilde{\mathbf{T}}'_s[\gamma_s(id(w_i))] \oplus \mathrm{iT}_s(id(w_i)), K_{id(w_i)})$, the simulator finally generates the token $\tilde{\tau}_s = (\tilde{\tau}_1, \ldots \tilde{\tau}_n)$.

Query for add token and ciphertext

Given $\mathcal{L}_3(\delta, \mathbf{f}, f)$ defined above, apprs(w) is set to 1 if the word w exists in the index before the file f is added; otherwise, it is set to 0.

If $id(f) \in \mathrm{iT}_d$, it means the file has already existed in the table. According to the algorithm, the simulator must return $\perp$ and terminate the simulation.

For the circumstances $id(f)$ is a new one, the simulator first checks the consistency of its structures in the following step.

For all $w_i \in \mathbf{w}_f$, checks if $\mathrm{iT}_s[id(w_i)] = (\perp, \perp)$ and apprs$(w_i) = 1$. If so, it means the simulators internal data structure has not been set properly. To fix this, the simulator chooses a random unused and non-free cell location α_1 in iA_s, stores $id(f)$ and $id(w_i)$ in this cell, and sets $\mathrm{iT}_s[id(w_i)] = \langle \alpha_1, \delta(\alpha_1) \rangle$.

After the checking the two values above, the simulator forms the set $\{\gamma_s(id(w_1)), \ldots, \gamma_s(id(w_{\#f}))\}$ and sends it to the adversary. The adversary is supposed to response the MHT proof $\rho = (t_1, \ldots, t_{\#f})$ to the simulator. The first element of the leaf node in each t_i should equal to the key $\gamma_s(id(w_i))$. If the verification of the proof ρ failed, the simulator returns $\perp$ and terminates the simulation.

The simulator then randomly chooses a $k-$bit string $K_{id(f)}$. If $id(f)$ is an entry in G or v is a random $\log \#\mathrm{A}_s$ bit string if $id(f)$ is not in G, it generates $v = \mathrm{G}[id(f)]$. And for all $w \in \mathbf{w}_f$, it generates $\lambda_i = (\gamma_s(id(w_i)), \theta'_{w_i,2}, \tilde{\mathbf{T}}'_s[\gamma_s(id(w_i))] \oplus \mathrm{iT}_s(id(w_i)), u_i, r_i, u'_i, r'_i)$, where u_i, r_i, u'_i, r'_i are random strings with the length of, respectively, $(2\log \#\mathrm{A}_s + \log \#\mathbf{f})$-bit, $6 \log \#\mathrm{A}_s$-bit, k-bit and k-bit. Let $\theta_{w_i} = (\theta_{w_i,1}, \theta_{w_i,2})$ be the leaf node in t_i, then $\theta'_{w_i,2} = (\theta_{w_i,2})^{s+id(f)}$.

Then, it returns the token and ciphertexts: $\tilde{\tau}_a = (\gamma_d(id(f)), v, \lambda_1, \ldots, \lambda_{\#\mathbf{w}_f})$, $\tilde{c}_f = \mathrm{Enc}_{K_4}(\mathbf{0}^{|f|})$.

After returning the token, the simulator needs to update its internal structures. For all $w_i \in \mathbf{w}_f$:

- Gets α_1 by inquiring $\mathrm{iT}_s[id(w_i)] = \langle \alpha_1, \delta(\alpha_1) \rangle$, despite α_1 is a valid address or $\perp$. Let φ and φ_{+1} be the first and second free location in iA_s.
- If $\alpha_1 \neq \perp$, updates $\tilde{\mathbf{A}}'_s(\alpha_1)$ by setting $\tilde{\mathbf{A}}'_s(\alpha_1) = (\tilde{\mathbf{N}} \oplus \langle \mathbf{0}, \varphi, \mathbf{0} \rangle, \tilde{r})$.
- Updates $\tilde{\mathbf{A}}'_s(\varphi)$ by setting $\tilde{\mathbf{A}}'_s(\varphi) = (\tilde{\mathbf{N}} \oplus \langle \mathbf{0}, \mathbf{0}, \alpha_1 \rangle, \tilde{r})$, updates $\tilde{\mathbf{A}}'_d[\delta(\varphi)]$ by setting $\tilde{\mathbf{A}}'_d[\delta(\varphi)] = (\tilde{\mathbf{D}} \oplus \langle \delta(\varphi_{+1}), \mathbf{0}, \delta(\alpha_1), \varphi, \mathbf{0}, \alpha_1 \rangle, \tilde{r})$.
- Updates the next pointer of the cell at location φ in iA_s point to α_1.
- Stores $id(f)$ and $id(w_i)$ in the cell at location φ in iA_s, and tags it as non-free cell.
- Sets $\mathrm{iT}_s[id(w_i)] = \langle \varphi, \delta(\varphi) \rangle$.

After those updating above, the simulator needs to create the list of $id(f)$ in iA_d in the following steps:

- Finds all the duals of the cells used above using the map δ. These cells should be free cells.
- Generates the list by updating each cells next pointer, according to the order given in $\mathcal{L}_3$.
- Marks these cells as used in iA_d with $id(f)$, and tags as non-free cells.
- Stores the corresponding word identifier $id(w_i)$ in each of these cells.
- Let h be the head of the list. Sets $\mathrm{iT}_d[id(f)] = h$, and updates $\tilde{\mathbf{T}}'_d[\gamma_d(id(f))]$ by setting: $\tilde{\mathbf{T}}'_d[\gamma_d(id(f))] = v \oplus h$.

Query for delete token

Given: $\mathcal{L}_4(\delta, \mathbf{f}, f)$ defined above, the simulator sets the corresponding value for the head and the tail of the list to $\perp$ to indicate that there are no more nodes before or after this one.

For all $w_i \in \mathbf{w}_f$, the simulator first checks the consistency of its internal structures as follows:

- It gets the word list from $\mathrm{iT}_s[id(w_i)]$. If $\mathrm{iT}_s[id(w_i)] = (\perp, \perp)$, the list has not been initialized yet. It chooses an unused and non-free cell in iA_s at random and marks it with $id(w_i)$ and $id(f)$. Let φ be the new location of the cell. It sets $\mathrm{iT}_s[id(w_i)] = (\varphi, \delta(\varphi))$.
- If $\mathrm{iT}_s[id(w_i)] \neq (\perp, \perp)$, it means there is already a list for w_i. The simulator searches the list for a cell stores $id(f)$. If no such cell exists, it chooses an unused and non-free cell in iA_s at random and marks it with $id(w_i)$ and $id(f)$. It then inserts this new cell into proper position of the list according to the $\mathrm{prev}(f, w)$ and $\mathrm{next}(f, w)$ given in $\mathcal{L}_4$.
- If $\mathrm{iT}_d[id(f)] = \perp$, the simulator needs to create a list for $id(f)$ by finding the duals of the cells found above, and modifying the pointers to form the list. The order of the cells in the list respects the order giving by $\mathcal{L}_4$. It then sets $\mathrm{iT}_d[id(f)]$ to be the location of the head of the list.

Then, the simulator forms the set $\{\gamma_s(id(w_1)), \ldots, \gamma_s(id(w_{\#f}))\}$ and sends it to the adversary. The adversary is supposed to response the MHT proof $\rho = (t_1, \ldots, t_{\#f})$ to the simulator. The first element of the leaf node in each t_i should equal to the key $\gamma_s(id(w_i))$. If the verification of the proof ρ failed, the simulator returns $\perp$ and terminates the simulation.

The simulator then returns the token:

$$\tilde{\tau}_d = \begin{pmatrix} \gamma_d\left(id(f)\right), \tilde{\mathbf{T}}'_d\left[\gamma_d\left(id(f)\right)\right] \oplus \mathrm{iT}_d\left[id(f)\right], \\ K_{id(f)}, id(f), \theta'_{w_1}, \ldots, \theta'_{w_{\#f}} \end{pmatrix},$$

where for all $w_i \in \mathbf{w}_f$, $\theta'_{w_i} = (\theta_{w_i,1}, (\theta_{w_i,2})^{1/(s+id(f))})$ is the modified leaf node to the original one $\theta_{w_i} = (\theta_{w_i,1}, \theta_{w_i,2})$ in t_i.

After returning the token, the simulator sets $\mathrm{G}[id(f)] = \tilde{\mathbf{T}}'_d[\gamma_d\,(id(f))] \oplus \mathrm{iT}_d[id(f)]$. Then it updates its data structures as follows:

- Marks the cells in the list $id(f)$ as free cells in iA_d, and their duals in iA_s too. When frees the cells in iA_s, updates their neighbors pointers to point to each other. Modify the pointers in iT_s if necessary.
- Removes the search key $id(f)$ from iT_d, and merges the newly freed cells in iA_s and iA_d into the free list, according to the order given in $\mathcal{L}_4$.

Analysis We now claim that no polynomial-time adversary $\mathcal{A}$ can distinguish between $\mathrm{Real}_{\mathcal{A}}(1^k)$ and $\mathrm{Ideal}_{\mathcal{A},\mathcal{S}}(1^k)$, except with negligible probability $\mathrm{negl}(1^k)$. We argue that this is not possible by stating that every output of the simulated experiment $\mathrm{Ideal}_{\mathcal{A},\mathcal{S}}(1^k)$ is indistinguishable from its corresponding elements in $\mathrm{Real}_{\mathcal{A}}(1^k)$.

- ($\mathrm{A}_s, \mathrm{T}_s, \mathrm{A}_d, \mathrm{T}_d$ and $\tilde{\mathbf{A}}_s, \tilde{\mathbf{T}}_s, \tilde{\mathbf{A}}_d, \tilde{\mathbf{T}}_d$)

Recall that each cell in $\tilde{\mathbf{A}}_s$ can be recognized as the form $\langle \tilde{\mathbf{N}}, \tilde{r}\rangle$ where $|\tilde{\mathbf{N}}| = 2 \log \#\mathrm{A}_s + \log \#\mathbf{f}$ and $|\tilde{r}| = k$. The cell in A_s is $\mathrm{N}_i = (\langle id_i, \mathrm{addr}_s(\mathrm{N}_{i-1}), \mathrm{addr}_s((\mathrm{N}_{i+1})\rangle \oplus H_1(P_{K_3}(w)r_i), r_i)$, due to the pseudo-randomness of P and the random oracle H_1, all PPT adversaries cannot distinguish $\tilde{\mathbf{A}}_s$ from A_s. Similarly, it cannot distinguish $\mathrm{T}_s, \mathrm{A}_d, \mathrm{T}_d$ with $\tilde{\mathbf{T}}_s, \tilde{\mathbf{A}}_d, \tilde{\mathbf{T}}_d$ if the pseudo-randomness of F, G and P holds. It means the adversary can distinguish the real index $\mathrm{A}_s, \mathrm{T}_s, \mathrm{A}_d, \mathrm{T}_d$ from the simulated index $\tilde{\mathbf{A}}_s, \tilde{\mathbf{T}}_s, \tilde{\mathbf{A}}_d, \tilde{\mathbf{T}}_d$. Therefore, the probability ε_1 is negligible.

- (α and $\tilde{\alpha}$)

Based on the elliptic curve discrete logarithm assumption and the pseudo-randomness of F, any PPT adversary $\mathcal{A}$ cannot distinguish the real leaf nodes $\mathcal{L}$ from the simulated one $\tilde{\mathcal{L}}$, therefore cannot distinguish the tree $\tilde{\alpha}$ from α, since they are generated by these leaves.

- (θ_w and $\tilde{\theta}_w$)

Because the pseudo-randomness of F holds, any PPT adversary cannot distinguish the random bits from the output of PRF F. So it cannot distinguish the random bits $\gamma_s\ (id(w_i))$ with $F_{K_1}(w_i)$. In addition, due to the discrete logarithm assumptions, any PPT adversary cannot can distinguish g^{ω_i} with $g^{\prod_{f\in \mathbf{f}_{w_i}}(s+id(f))}$. As we know, the real leaf nodes from the simulated one are in the forms of:

$$\begin{cases} \tilde{\mathcal{L}} = \{\tilde{\theta}_w\}_{w\in\mathbf{w}} = \{(\tilde{\theta}_{w,1}, \tilde{\theta}_{w,2})\}_{w\in\mathbf{w}} \\ \mathcal{L} = \{\theta_w\}_{w\in\mathbf{w}} = \{(\theta_{w,1}, \theta_{w,2})\}_{w\in\mathbf{w}} \end{cases}$$

So $\mathcal{A}$ cannot distinguish $\tilde{\mathcal{L}}$ from $\mathcal{L}$. The tree $\tilde{\alpha}$ and α are built by the collision-resistant hash function H_3 of the leaf nodes, so the adversary cannot distinguish the tree $\tilde{\alpha}$ and α.

- ($\mathbf{c}$ and $\tilde{\mathbf{c}}$)

Recall that $\tilde{c}$ is Π encryption, which is CPA-secure. Since, all PPT adversaries $\mathcal{A}$, the CPA security of Π will guarantee that all PPT adversaries $\mathcal{A}$ can distinguish ($\tilde{\mathbf{c}}$ and $\mathbf{c}$) with negligible probability. Because the Π encryption is proved to be CPA-secure, so the ciphertexts it produces do not reveal any partial information about the plaintext. So any PPT adversary $\mathcal{A}$ cannot distinguish the ciphertexts that generated by two different inputs using the Π encryption. As we know, $\tilde{\mathbf{c}}$ and $\mathbf{c}$ are ciphertexts in the forms of:

$$\begin{cases} \mathbf{c} = (c_1, \ldots, c_{\#\mathbf{f}}),\ \text{in which } c_i = \Pi.\mathrm{Enc}_{K_4}(f_i) \\ \tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_{\#\mathbf{f}}),\ \text{in which } \tilde{c}_i = \Pi.\mathrm{Enc}_{K_4}(\mathbf{0}^{|f|}) \end{cases},$$

so $\mathcal{A}$ cannot distinguish $\tilde{\mathbf{c}}$ from $\mathbf{c}$.

- (τ_s and $\tilde{\tau}_s$)

Recall that τ_s and $\tilde{\tau}_s$ are in the forms of:

$$\begin{cases} \tau_i = (F_{K_1}(w_i), G_{K_2}(w_i), P_{K_3}(w_i)) \\ \tilde{\tau}_i = (\gamma_s(id(w_i)), \mathbf{T}'_s[\gamma_s(id(w_i))] \oplus \mathrm{iT}_s(id(w_i)), K_{id(w_i)}) \end{cases}$$

because the pseudo-randomness of F, G, P holds, any PPT adversary cannot distinguish the random bits from the output of PRF F, G, P. So it cannot distinguish τ_s and $\tilde{\tau}_s$, so as $\tau_s = (\tau_1, \dots \tau_n)$ and $\tau_s = (\tilde{\tau}_1, \dots \tilde{\tau}_n)$.

- $(\tau_a, \tau_d$ and $\tilde{\tau}_a, \tilde{\tau}_d)$

In the same way, based on the elliptic curve discrete logarithm assumption and the pseudo-randomness of F, G, P, any PPT adversary cannot distinguish the random bits from the output of PRF F, G, P. So it cannot distinguish $(\tau_a, \tau_d$ and $\tilde{\tau}_a, \tilde{\tau}_d)$.

To sum up, we have the conclusion that for all PPT adversaries $\mathcal{A}$, the output of $\mathrm{Real}_{\mathcal{A}}(1^k)$ and $\mathrm{Ideal}_{\mathcal{A},\mathcal{S}}(1^k)$ are identical, except with negligible probability $\mathrm{negl}(1^k)$:

$$|\Pr \mathrm{Real}_{\mathcal{A}}(1^k) = 1] - \Pr[\mathrm{Ideal}_{\mathcal{A},\mathcal{S}}(1^k) = 1]| \le \mathrm{negl}(1^k).$$

Therefore our scheme is $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$ -secure against adaptive chosen-keyword attacks in random oracle model.

6.2 Unforgeability

The method to give the proof for unforgeability is straightforward and needs no additional explanations.

Theorem 2 *If H_3 is collision-resistant hash function and the bilinear q-SDH assumption holds then our scheme is unforgeable.*

Proof The main idea to give the proof is that, if there exists a PPT adversary $\mathcal{A}$ such that $\mathrm{Forge}_{\mathcal{A}}(1^k) = 1$, then there exist a PPT simulator $\mathcal{S}$ that breaks at least one of the assumptions:

- The collision-resistance property of H_3,
- Bilinear q-SDH assumption.

During the game, the simulator $\mathcal{S}$ interacts with $\mathcal{A}$ using real algorithm. Assume after q times queries, $\mathcal{A}$ outputs a set of file identifiers $\mathbf{I}' \neq \mathbf{I}_W$ and a valid proof π. This means the proof $\pi = \{\mathcal{T}, \mathcal{S}, \mathcal{C}\}$ he produces under query W passes all three steps of the verification phase. We categorize the forgery into three types:

- **Type I:** For some word $w_i \in W$, the adversary outputs a different leaf value $\tilde{\theta}_{w_i}$ in Merkle tree proof $\tilde{t}_i$ and passes the verification step 1.
- **Type II:** For some word $w_i \in W, \mathbf{I}' \not\subset S_i$. The adversary gives the simulator the real accumulation value in the proof t_i, and outputs a subset witness $\tilde{g}^{P_i}$ that passes the verification step 2.
- **Type III:** The set $\mathbf{I}'$ is a proper subset of $\mathbf{I}_W$. The adversary gives the simulator the real $\mathcal{S} = \{g^{P_1}, \dots, g^{P_n}\}$, and outputs a completeness witness $\widehat{\mathcal{C}}$ which passes the verification step 3.

It is clear that if $\mathbf{I}' \neq \mathbf{I}_W$ and proof π is valid then one of the above mentioned forgeries must occur. Next we show that the simulator $\mathcal{S}$ can use **type I** forgery to break the collision-resistance property of H_3, and use **type II** or **III** forgeries to break the bilinear q-SDH assumption.

(A) The collision-resistance property of H_3

The hash function H_3 is collision-resistance if it is difficult for all PPT adversaries to find two different messages m_1 and m_2, such that $H_3(m_1) = H_3(m_2)$.

First, given the hash function H_3, the simulator $\mathcal{S}$ interacts with the adversary $\mathcal{A}$ according to the game $\mathrm{Forge}_{\mathcal{A}}(1^k)$.

If $\mathcal{A}$ wins the game and the **type I** forgery occurs, that means for some word $w_i \in W$, $\mathcal{A}$ outputs a different leaf value $\tilde{\theta}_{w_i}$ in Merkle tree proof $\tilde{t}_i$.

Then, the simulator $\mathcal{S}$ verifies the value $\mathcal{A}$ outputs, which passes the verification step 1. Let $\tilde{\theta}_{w_i} = (\tilde{\theta}_{w_i}, \tilde{\theta}_{w_i})$. Passing the verification step 1 means the following two conditions hold:

- The search key $F_{K_1}(w_i) = \tilde{\theta}_{w_i,1}$.
- The tree verification using the leaf $\tilde{\theta}_{w_i}$ succeeds.

According to the verification step 1, the adversary $\mathcal{A}$ may only forge the $\tilde{\theta}_{w_i,2}$. Then passing the Merkle tree verification implies that the adversary is able to find the collision of H_3, because it can generate the same root with the modified leaf.

(B) Bilinear q-SDH assumption

Given the simulator $\mathcal{S}$ an instance of bilinear q-SDH problem: $(p, \mathbf{G}, \mathcal{G}, e, g)$ and a $(q+1)$-tuple $(g, g^s, \dots, g^{s^q})$. $\mathcal{S}$ interacts with the adversary $\mathcal{A}$ in the following way.

First, since $\mathcal{S}$ doesn't know the value s in the given bilinear q-SDH instance, it needs to reconstruct the following algorithms which related to s in the game $\mathrm{Forge}_{\mathcal{A}}(1^k)$:

- **Gen**: the simulator $\mathcal{S}$ directly uses $(g, g^s, \dots, g^{s^q})$ as the public parameters without knowing s, and sends them to the adversary.
- **Setup**: for leaf nodes: $\theta_w = \langle F_{K_1}(w), g^{\prod_{f \in \mathbf{f}_w}(s+id(f)}\rangle$, the simulator $\mathcal{S}$ computes the value $g^{\prod_{f \in \mathbf{f}_w}(s+id(f))}$ using $(g, g^s, \dots, g^{s^q})$. It is worth mentioning that, the simulator $\mathcal{S}$ needs to construct an extra auxiliary data structure $\mathcal{N}$. It stores for each leaf nodes θ_w the polynomial: $n_w = \prod_{f \in \mathbf{f}_w}(s + id(f))$, which is used to form the add/delete tokens later.
- **Add/Update**: Simulator $\mathcal{S}$ cannot directly compute the value of τ_a in Add/Update protocol. In the Add/Update protocols users step 2(b), when computing the value of the new leaf node $\theta_i{}'$ in τ_a, it first finds $\mathcal{N}$ to find the polynomials n_i that equals $\theta_{i,2}$, then computes the

value $g^{n_i \cdot (s+id(f))}$ using $(g, g^s, \ldots, g^{s^q})$. The value $\theta'_i = (\theta_{i,1}, g^{n_i \cdot (s+id(f))})$ is the updated leaf node.

- **Del/Update**: Similarly, in the Del/Update protocol, $\mathcal{S}$ finds the n_i in $\mathcal{N}$ and removes the factor $s + id(f)$ from n_i and then computes the value $g^{n_i/(s+id(f))}$ using $(g, g^s, \ldots, g^{s^q})$ and gets a new leaf node $\theta'_i = (\theta_{i,1}, g^{n_i/(s+id(f))})$.

In the above modified game, the values that related to s are computed in a new way. However, it produces same output as it was produced by earlier version of algorithm. So in the view of adversary $\mathcal{A}$, these values are still valid, it cannot distinguish this game with the original one.

If $\mathcal{A}$ wins the game, and the following two types of forgeries occur, then the simulator $\mathcal{S}$ may solve the given bilinear q-SDH instance.

1. If the **type II** forgery occurs, i.e., the adversary $\mathcal{A}$ outputs a set $\mathbf{I}' = \{x_1, \ldots, x_m\}$ and a witness $\widehat{g}^{P_i}$ for some $w_i \in W$. Let $S_i = \{id_1, \ldots, id_q\}$ be the file identifier set related to the word w_i, then $g^{(s+id_1)(s+id_2)\cdots(s+id_q)}$ is the corresponding accumulation value. Since $\mathbf{I}' \not\subset S_i$, there exists some $1 \leq j \leq m$, such that $x_j \notin S_i$. This means in the equation: $e(g^{\prod_{x\in\mathbf{I}'}(s+x)}, \tilde{g}^{P_i}) = e(g^{(s+id_1)} g^(s + id_2) \cdots (s + id_q), g)$, $(s + x_j)$ does not divide $(s + id_1)(s + id_2) \cdots (s + id_q)$.
   Therefore there exists polynomial $Q(s)$ of degree $q - 1$ and constant $c \neq 0$, such that: $(s + id_1)(s + id_2) \cdots (s + id_q) = Q(s)(s + x_j) + c$. Then the simulator $\mathcal{S}$ has $e(g, \tilde{g}^{P_i})^{(s+x_j)\prod_{x\in\mathbf{I}'\wedge x\neq x_j}(s+x)} = e(g, g)^{Q(s)(s+x_j)+c}$. After transformation, it can finally have
   $$e(g, g)^{\frac{1}{s+x_j}} = (e(g, \tilde{g}^{P_i})^{\prod_{x\in\mathbf{I}'\wedge x\neq x_j}(s+x)} e(g, g)^{-Q(s)})^{1/c}.$$
   This means the simulator $\mathcal{S}$ can solve the instance of bilinear q-SDH problem in polynomial time.
2. If the **type III** forgery occurs, i.e., the adversary $\mathcal{A}$ outputs a set $\mathbf{I}' = \{x_1, \ldots, x_m\}$ and the completeness witness $\widehat{\mathcal{C}}$. Since the set $\mathbf{I}'$ is the proper subset of $\mathbf{I}_W$, there exists at least one common factor in polynomials $P_1, \ldots, P_n$. We use $(s + x)$ to denote the factor, where $x \notin \mathbf{I}'$.

These values can pass the verification step 3 means the following holds: $\prod_{i=1}^n e(g^{P_i}, g^{q_i}) = e(g, g)$. Extract $(s + x)$ from each P_i by computing $g^{P'_i} = (g^{P_i})^{1/(s+x)}$, then $\prod_{i=1}^n e(g^{P_i}, g^{q_i}) = (\prod_{i=1}^n e(g^{P'_i}, g^{q_i})^{s+x} = e(g, g)$. Thus the simulator $\mathcal{S}$ can easily form the solution of the instance of bilinear q-SDH problem by computing:

$$e(g, g)^{1/(s+x)} = \prod_{i=1}^n e(g^{P'_i}, g^{q_i}).$$

This means the simulator $\mathcal{S}$ can also solve the instance of bilinear q-SDH problem in polynomial time.

The above analyses show that, if the adversary $\mathcal{A}$ could successfully forge a proof, it must have the ability to break at least one of these assumptions above. Therefore we have the conclusion that for all PPT adversaries $\mathcal{A}$, the probability $\Pr[\text{Forge}_{\mathcal{A}}(1^k) = 1] \leq \text{negl}(1^k)$, where $\text{negl}(1^k)$ is a negligible function with input 1^k. Thus our scheme is unforgeable.

7 Experiment result

In this section, we describe the implementation of our conjunctive keyword searchable encryption scheme and discuss the results of experimentation regarding its practical performance.

We implement our scheme in C++ under the 256 bit system security parameter. The experiments were run on linux ubuntu installed on a rotational disk. We chose AES-CBC-256, SHA256, and HMAC with SHA256 in the OpenSSL library for the symmetric encryption, the random oracles, and the pseudo-randomness functions. We use PBC library to compute the group operations including the pairing and the point exponentiation calculation. We select the type A pairings in PBC to realize the symmetric pairing, which is based on the curve over the field F_q for some prime $q = 3 \bmod 4$. The group order is 256 bits and the order of its base field is 512 bits. We use the NTL library for efficient FFT and Euclidean algorithms over polynomials.

The dataset was selected from real word. We used Enron emails dataset at https://www.cs.cmu.edu/ enron/, the August 21, 2009 version for evaluation. We chose a subset of 1000 files randomly from those mails as file collection. Each separate word in the file served as the keyword. We chose the words that appear in most files as keywords, and performed conjunctive keyword search from 1 to 5 keywords. As a comparison, we used the scheme in [14] to perform multiple times single-keyword search. Due to the conjunctive keyword search is unsupported in [14], the server need not to prove the correctness of the intersection of the searching results, and the verifications in [14] were executed at the user side, so the proof phase and verification phase in [11] is uncompareble with ours.

In order to validate the practicality of our conjunctive keyword searchable encryption scheme, we implemented the scheme in a client–server architecture and deployed the server end to a remote server. The remote server has an Intel Core i7-2600 Quad-Core Processor 3.4 GHz and 8 GB of RAM. The cloud server stores not only the set of file ciphertexts $\mathbf{c}$,but also the encrypted index γ and the Merkle tree authenticated structure α,after the Setup algorithm. The stotage overhead of the set of file ciphertexts $\mathbf{c}$ is up to the size of the file sets and the encryption algorithm $\Pi.enc$. For the

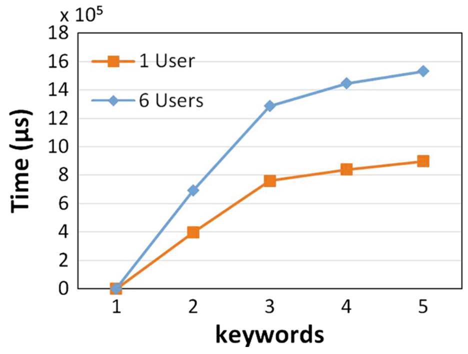

Fig. 4 Search Time affected by keywords'number for 1 and 6 users

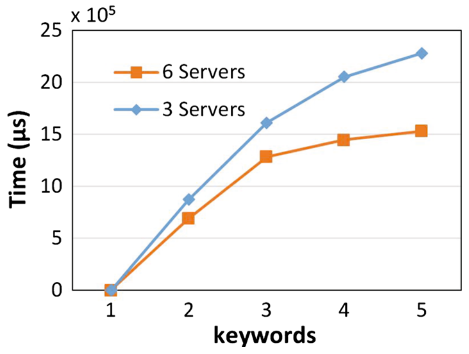

Fig. 5 Search time affected by keywords'number for 3 and 6 servers

encrypted index γ, the storage overhead is mainly determined by the size of the files and the amounts of the keywords. And the size of the Merkle tree authenticated structure α is only determined by the size of the files. The clinet in our system is simulated by several computers. Each computer can simulate one clinet to interact with the server. The clinet stores the users secret keys and the root value st of the Merkle tree authenticated structure α.

Furthermore, we implemented a job allocation mechanism in the server end that acts as the master server, and used threads to simulate different node servers that do the actual search jobs. As a result, our experiment environment simulates a real-world cloud service: The clients only communicate with the master server, and the master server allocates the search jobs dynamically to many node servers.

Under the experiment environment described above, we conducted the following experiments. During all the experiments, the network communication time is not recorded since it is highly depended on the network connection between the client and the server.

In the first experiment, we tested the search time (including proof generation) required to perform conjunctive keyword search queries with different number of keywords for both 1 and 6 users, as illustrated in Fig. 4. We used 6 servers (threads) for this experiment. When testing with 6 users, all the 6 clients send the search queries to the master server at the same time, and the time is recorded until every search query is finished.

Moreover, we focused on how different number of servers affects the performance of the system, as shown in Fig. 5. We divided the experiment into two parts. Firstly, we used 3 servers to perform conjunctive keyword search queries with different number of keywords for 6 users, and the total time cost is recorded. Then we repeated the experiment with 6 servers, and compared the results.

The communication overhead is affected by the number of files and the size of the proof. While searching for n keywords out of the t total distinct keywords that are indexed in the system, the total size is $O(m + n \log t)$, where m is the file number in the search result.

While taking into account the file size, we refer to the size of $id(f)$, instead of the size of file itself, since the user only needs the set of $id(f)$ to verify the result. We split the communication size since the result consists of two parts: the files set, and the proof. In Fig. 6 we show how the size is affected by the number of keywords in a search. As the number of keywords increased, the intersection size becomes smaller, and may eventually come to zero. In this case, the main factor affecting the size is the proof, which is $O(n \log t)$ to the number of keywords.

7.1 Search and prove overheads

We then measured the time it takes for the server to search and to compute the proof. Figures 7 and 8 illustrate the time it takes to search and prove versus number of keywords. Let Nbe the total size of the keywords inverted lists. Theoretically, the asymptotic running time at the server is $O(N\log^2 N \log\log N)$ [21]. In practice, the critical computation at the server is the power operation for group elements, which is executed to construct the subset and completeness witnesses.

Its worth mentioning that, the algorithms can be further optimized using implementation techniques which can greatly reduce the prove time at the server side, such as the multi-thread execution and the precomputed P_i and g^{P_i}. However, in order to honestly reflect the original efficiency of the algorithm, we hadn't use any optimize techniques, just simply let the server do all the computation in single thread each time.

7.2 Verification overhead

The verifications asymptotic running time on the user side is $O(m + n \log t)$. Figure 9 illustrates the time it takes to verify all the proofs versus number of keywords. Compares to the

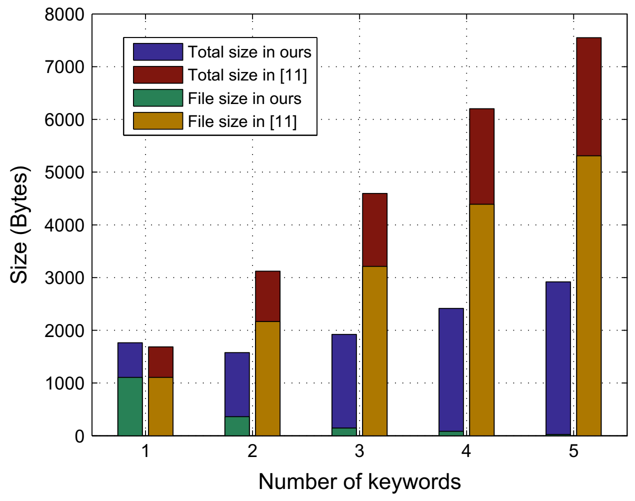

Fig. 6 Communication time affected by keywords' number

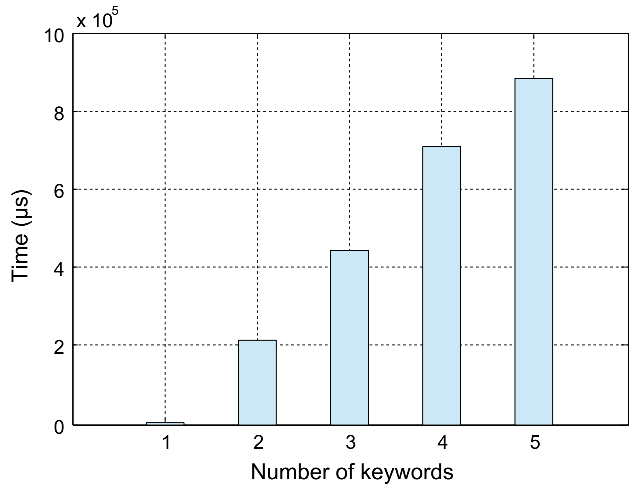

Fig. 8 Prove time affected by keywords' number

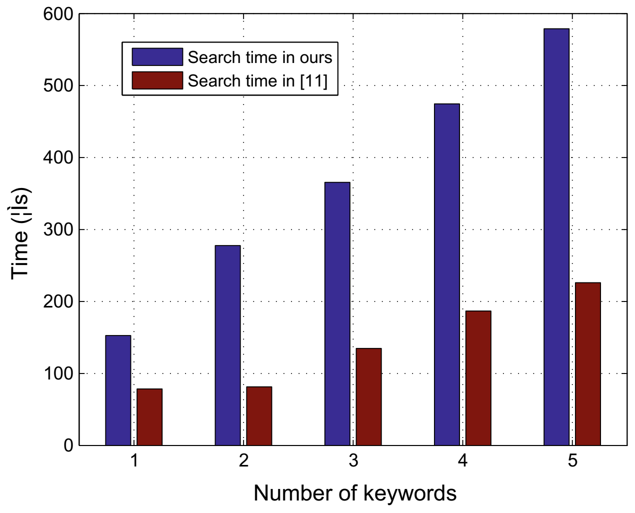

Fig. 7 Search time affected by keywords' number

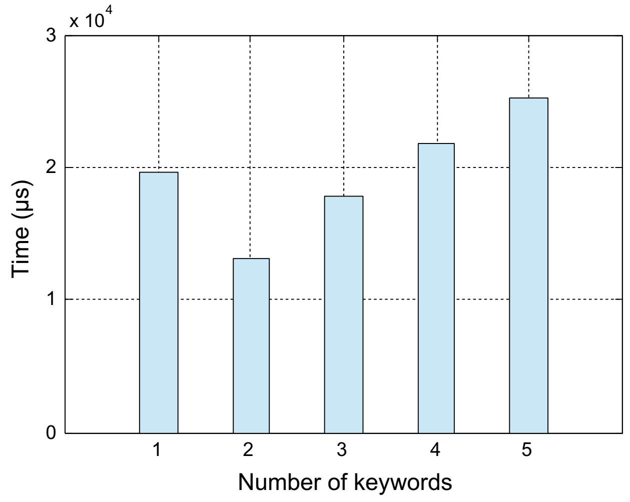

Fig. 9 Total verification time on user side

search and prove operation at the server side, the computation at the user side is relatively fast.

The most expensive part in the verification algorithm is the bilinear pairing. While searching for n keywords, user needs to compute $n + 1$ pairings to verify the results correctness. This step could also be further optimized while the search result is empty. In this case, user only needs to check whether each g^{P_i} equals $\theta_{i,2}$ in the verification step 1(b) and do single pairing operation in step 1(c) to check the completeness condition.

8 Conclusion

Searchable encryption is an important cryptographic primitive for cloud storage environment. It is well motivated by the popularity of cloud storage services. At the same time, the authentication methods utilized for the search results verification is a significant supplement that makes the search more reliable and it would greatly promote the development of cloud storage service.

In this paper, we described a searchable encryption scheme which supports conjunctive keyword search and authentication. The scheme can greatly reduce the communication cost during the search. We demonstrated that, taking into account the security challenges in the cloud storage, our scheme can withstand the chosen-keyword attack carried out by the adaptive adversaries. Proposed scheme also prevents the result from being maliciously altered by those adversaries.

In the future, we will perform a detailed analysis of the security aspects in this paper and investigate the feasibility of designing improved security model to enhance the schemes security features. Moreover, we will give consideration to the authenticate techniques to achieve more efficiency to meet practical needs.

Acknowledgements This work was supported in part by the National Natural Science Foundation of China under Grant Nos. 61772127, 61472184 and 61321491, the National Science and Technology Major Project under Grant No. 2013ZX03002006, the Liaoning Province Science and Technology Projects under Grant No. 2013217004, the Liaoning Province Doctor Startup Fund under Grant No. 20141012, the Fundamental Research Funds for the Central Universities under Grant No. N130317002, the Shenyang Province Science and Technology Projects under Grant No. F14-231-1-08.

References

1. Cachin, C., Keidar, I., Shraer, A.: Trusting the cloud. ACM Sigact News **40**(2), 81–86 (2009)
2. Kamara, S., Lauter, K.: Cryptographic cloud storage. Financ. Cryptogr. Data Secur. **6054**, 136–149 (2010)
3. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data, security and privacy. In: Proceedings of 2000 IEEE Symposium on IEEE, pp. 44–55 (2000)
4. Goh, E.J.: Secure indexes. In: IACR Cryptology ePrint Archive, p. 216 (2003)
5. Chang, Y.C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) Applied Cryptography and Network Security. ACNS 2005. Lecture Notes in Computer Science, vol. 3531, pp. 442–455. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11496137_30
6. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, pp. 79–88, ACM (2006)
7. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM Conference on Computer and Communications Securitypp, pp. 965–976, ACM (2012)
8. Cash, D. Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H. Rosu, M.C., Steiner, M.: Dynamic searchable encryption in very large databases: data structures and implementation. In: Proceedings of NDSS (2014)
9. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: Proceedings of NDSS (2014)
10. Golle, P., Staddon, J., Waters, B.: Secure conjunctive keyword search over encrypted data. In: Jakobsson, M., Yung, M., Zhou, J. (eds.) Applied Cryptography and Network Security. ACNS 2004. Lecture Notes in Computer Science, vol. 3089, pp. 31–45. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24852-1_3
11. Kamara, S., Papamanthou, C., Roeder, T.: CS2: a searchable cryptographic cloud storage system. TechReport MSR-TR-2011-58, Microsoft Research (2011)
12. Byun, J.W., Lee, D.H., Lim, J.: Efficient conjunctive keyword search on encrypted data storage system. In: Atzen, A.S., Lioy, A. (eds.) Public Key Infrastructure. EuroPKI 2006. Lecture Notes in Computer Science, vol. 4043, pp. 184–196. Springer, Berlin (2006). https://doi.org/10.1007/11774716
13. 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)
14. Ballard, L., Kamara, S., Monrose, F.: Achieving efficient conjunctive keyword searches over encrypted data. In: Qing, S., Mao, W., López, J., Wang, G. (eds.) Information and Communications Security. ICICS 2005. Lecture Notes in Computer Science, vol. 3783, pp. 414–426. Springer, Berlin (2005). https://doi.org/10.1007/11602897_35
15. Kurosawa, K., Ohtaki, Y.: UC-Secure Searchable Symmetric Encryption, Financial Cryptography and Data Security, pp. 285–298. Springer, Berlin (2012)
16. Boneh, D., Di Crescenzo, G., Ostrovsky, R., Persiano, G.: Public key encryption with keyword search. In: Cachin, C., Camenisch, J.L. (eds.) Advances in Cryptology—EUROCRYPT 2004. Eurocrypt 2004. Lecture Notes in Computer Science, vol. 3027, pp. 506–522. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-24676-3_30
17. Gentry, C.: Practical identity-based encryption without random oracles. In: Vaudenay, S. (ed.) Advances in Cryptology—EUROCRYPT 2006. Eurocrypt 2006. Lecture Notes in Computer Science, vol. 4004, pp. 445–464. Springer, Berlin (2006). https://doi.org/10.1007/11761679_27
18. Baek, J., Safavi-Naini, R., Susilo, W.: Public Key Encryption with Keyword Search Revisited. In: Gervasi, O., Murgante, B., Laganà, A., Taniar, D., Mun, Y., Gavrilova, M.L. (eds.) Computational Science and Its Applications—ICCSA 2008. ICCSA 2008. Lecture Notes in Computer Science, vol. 5072, pp. 1249–1259. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-69839-5_96
19. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press, Boca Raton (2014)
20. Merkle, R.C.: A Certified Digital Signature. In: Brassard, G. (ed.) Advances in Cryptology—CRYPTO’ 89 Proceedings. CRYPTO 1989. Lecture Notes in Computer Science, vol. 435, pp. 218–238. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_21
21. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Optimal Verification of Operations on Dynamic Sets. In: Rogaway, P. (ed.) Advances in Cryptology—CRYPTO 2011. CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 91–110. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-22792-9_6
22. Stefanov, E., Van Dijk, M., Shi, E. et al.: Path ORAM: an extremely simple oblivious RAM protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications security, pp. 299–310 (2013)