Forward Secure Searchable Encryption with Conjunctive-Keyword Supporting Multi-user

Zhongyi Liu, Chungen Xu(✉), and Zhigang Yao

School of Science, Nanjing University of Science and Technology, Nanjing, China
ZhongyiLiu950217@outlook.com, xuchung@njust.edu.cn,
zhigangyaocrypto@outlook.com

Abstract. Searchable symmetric encryption (SSE) enables users to efficiently search ciphertext in the cloud and ensures the security of encrypted data. Recent works show that forward security is an important property in dynamic SSE. Many forward secure searchable symmetric encryption (FSSE) schemes supporting single-keyword search have been proposed. Only a few SSE schemes can satisfy the forward security and support conjunctive keyword search at the same time, which are realized by adopting inefficient or complicated cryptography tools. Very recently, Hu proposed a novel construction to achieve conjunctive-keyword search, that is, using inner-product encryption (IPE) to design a conjunctive-keyword FSSE scheme. However, IPE scheme is a conceptually complex and low efficient scheme. In this paper, we use a more efficient cryptographic tool, asymmetric scalar-product-preserving encryption (ASPE), to design an efficient and secure conjunctive-keyword FSSE scheme. To improve practicality, we design our scheme to support multi-user setting. Our scheme achieves sub-linear efficiency, and can easily be used in any single-keyword FSSE scheme to obtain a conjunctive-keyword FSSE scheme supporting multi-user. Compared with the current conjunctive-keyword FSSE scheme, our scheme has a better update and search efficiency.

Keywords: Forward security · Conjunctive-keyword search · Multi-user

1 Introduction

With the development of network technology, cloud computing technology has been widely used by companies and individuals. Using cloud storage service, users can outsource large amounts of data to the cloud. For data security, users need to encrypt their data before uploading. Searchable symmetric encryption

This work was supported in part by the Fundamental Research Funds for the Central Universities (No. 30918012204).

D. Wang et al. (Eds.): SPNCE 2020, LNICST 344, pp. 423–440, 2021.
https://doi.org/10.1007/978-3-030-66922-5_29

(SSE) [13] provides keyword search function while ensuring the security of cloud data. In an SSE scheme, the user encrypts files and (ind, keywords) pairs, then sends the encrypted data to the server; When the user wants to search the files containing keyword w, he generates a search token using the secret key and sends it to the server; After receiving the search token, it runs the match operation to obtain the files containing the keyword w, and then sends those files to the user. Finally, the user will decrypt the received data to get the files that he wants. Dynamic searchable symmetric encryption (DSSE) is more practical than SSE. It can support dynamic update operations. The user can add new files to the server or delete an old file in the server.

Some basic leakages are inevitable for SSE schemes as they support keyword search over encrypted data. Curtmola [4] proposed the definition of the leakage of access pattern and search pattern. DSSE scheme leaks more information than SSE scheme. In DSSE scheme, an adversary can inject files containing some special keywords into the server's database; Then the adversary can use old search queries to search those injected files and then obtain some useful information from the search result. This kind of attack is called file injection attack (FIA) [19]. The FSSE schemes can well resist file injection attack. Because in a FSSE scheme, old queries can not match the new file which not queried before. Bost [1] proposed a new idea for constructing FSSE schemes. Only trapdoor permutation is used to construct the FSSE scheme in Bost's scheme. In order to improve the practicability of FSSE scheme, some FSSE schemes [7,16] that support conjunctive-keyword search have been proposed. These schemes can search files containing multiple keywords. Most of the existing SSE schemes that support boolean queries leak the *result pattern* information [3]. This leakage allows the server to know which files contain part of the search keywords.

1.1 Our Contributions

The main contributions of this paper are as follows:

- We construct an efficient and secure FSSE scheme that supports conjunctive-keyword query and multi-user. Compared with the best current conjunctive-keyword FSSE scheme, our scheme has a better update and query efficiency. Our scheme can hide the number of keywords in a search query.
- Our scheme proves the feasibility of ASPE in constructing conjunctive-keyword FSSE scheme.
- We give the security analysis and detailed efficiency description of our scheme. We also implement our scheme using Java programming language to test its practicality.

1.2 Related Work

Song et al. [13] proposed the first SSE scheme, which searching over the ciphertext instead of using the index table. So its search complexity is linear with the number of documents. After that, in order to improve search efficiency, many

schemes using an index table [2,3,5,8,9,15] have been proposed. Goh et al. [5] proposed the first index-based SSE scheme. Schemes using index table will greatly improve search efficiency. In addition, researchers designed some SSE schemes with more properties, and hence, is more practical. Kamara et al. [9] proposed a DSSE scheme that can achieve sub-linear search efficiency. To support conjunctive-keyword search, Golle et al. [6] first proposed conjunctive-keyword search scheme. But their scheme are not very efficient. It has linear complexity in the whole number of documents. After that, Cash et al. [3] proposed the first SSE scheme that can achieve sub-linear search efficiency and support boolean queries. Lai [11] proposed a scheme that solves the Keyword-Pair Result Pattern leakage of OXT and achieves almost the same efficiency as OXT. But Lai's scheme needs one more round communication in search protocol. Zhang et al. [19] gave a formalized definition of a very strong attack, named file injection attack. This attack can easily recover the keyword of a query. Stefanov et al. [15] first formalized the notion of forward security for SSE scheme. After Bost [1] proposed a very creative way to design a practical FSSE scheme, many practical FSSE schemes were successively proposed. Song et al. [14] improved efficiency based on Bost's scheme using pseudorandom permutation. After that, Zhang et al. [20] proposed a more efficient way to construct FSSE scheme than Song's scheme [14]. According to Zhang's idea, we can use only hash function to construct a FSSE scheme. Recently, a few FSSE schemes support conjunctive-keyword search [7,16] were proposed. But the cryptography tools they used are not efficient enough.

1.3 Organization

The rest of this paper is organized as follows. In Sect. 2, we describe the system model, threat model and design goals of our system. The notations and cryptographic primitives used in this paper are introduced in Sect. 3. In Sect. 4, we give the detailed structure of our scheme and describe how to deploy our scheme in a real scenario. We analyze the security of our scheme and compare the complexity with existing schemes and give the experimental result in Sect. 5 and 6, respectively. Finally, we give a brief conclusion in Sect. 7.

2 Problem Statement

In this section, we will introduce the system model, threat model and design goals of our system.

2.1 System Model

Figure 1 records an overview of our system model. There are three parties: server, private key generator and user. User can be both data owner and data user. These parties can be described as follows:

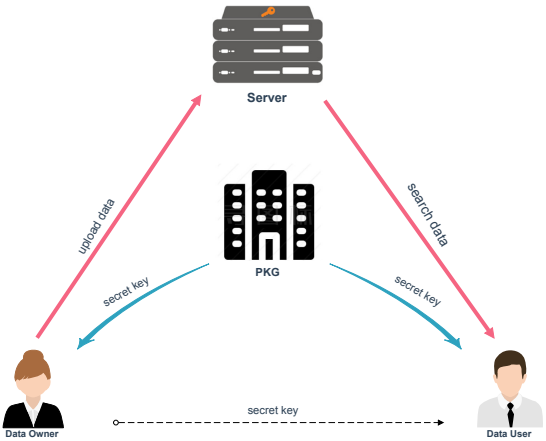

Fig. 1. System model

- **Server:** The server is a semi-honest provider of cloud storage services. It stores encrypted data sent by data owner and responses search operation honestly. But it will learn and try to get some plain text information
- **PKG:** PKG is a trusted center. It first generates system parameters, including the public parameter pp and the master key msk. PKG then uses the system parameters to generate private keys for each user. PKG will not leak his master key to anyone.
- **User:** All users are authenticated. They can both be data owner and data user. They can send their file to the server and search the files in the server. Users can communicate with each other for data sharing.

How the System Works. First, PKG generates the system parameters pp and msk and predefine the keyword space W. PKG then responds to the registration request of the authenticated user and generates a key for him. After receiving the key, the user generates a random number by himself and uses both the key from PKG and the random number as his secret key. After getting the secret key, the user can encrypt his data and then send the ciphertext to the server. When a user wants to search his own data containing keywords $W_{search} = \{w_1, \cdots, w_l\}$, he can use his secret key to generate a search token, and then sends the search token to the server. After receiving the search token, the server runs match operation using the token and return the result to the user. When a user U_1 wants to search the data of another user U_2, U_1 first sends a request to U_2, U_2 then share his secret key and a state st_c with U_1. U_1 can generate search token using the secret key and state, and then search files in the server.

2.2 Threat Model

The server in our system is considered semi-honest, that is, it will perform every operation honestly, but it will try to learn information about the ciphertext from

each operation. The private key generator PKG is considered credible. It will not leak any information about system parameters and any user's secret key to other parties. Only an authenticated person can register as a user in the system. And each user will not leak his secret key to unauthenticated parties.

2.3 Design Goals

We design our system with the following goals:

- **Forward Privacy:** Newly added files would not cause previous queries to leak information.
- **Update and Search Efficiency:** Our scheme is more efficient than other existing conjunctive-keyword FSSE schemes both in update and search efficiency.
- **Scalability:** The method we use to construct our system can be easily applied to other single-keyword FSSE schemes.

3 Preliminaries

In this section, we present some notations and cryptographic primitives used in this paper. Table 1 is part of the notations we use in this paper.

Table 1. Notations and descriptions

Notation	Description
λ	Security parameter
N	The number of total keywords allowed by the system
$\mathcal{Z}_p$	A finite field, where p is a big prime integer
$\mathbb{G}_1, \mathbb{G}_t$	Two cyclic groups with prime order p
pp	The public parameter of PKG
msk	The master key of PKG
W	A predefined ordered keyword space
H_i	i-th hash function
F_i	i-th pseudorandom function
F_p	A pseudorandom function with range $\mathbb{Z}_p$
$\lvert \cdot \rvert$	Cardinality of set
Σ, T	Two maps
st_i	The ith state
ind	A document index
U_{id}	One user with identity id
$x \xleftarrow{\$} X$	Uniformly sample an element from a set X
$negl(\lambda)$	The negligible function in the security parameter λ
$PRPV(key, \boldsymbol{v})$	Randomly rearrange the vector's elements
op	The update operation, where $op \in \{add, del\}$
W_{search}	All keywords in a search query
W_{ind}	All keywords in a file ind

3.1 Bilinear Maps

Definition 1. *Let $\mathbb{G}_1$ and $\mathbb{G}_t$ be two cyclic groups with the same prime order p. Let g be a generator of group $\mathbb{G}_1$. $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_t$ is a map from $\mathbb{G}_1$ to $\mathbb{G}_t$. e is a bilinear map if it has the following three properties:*

1. *Bilinearity: $\forall g_1, g_2 \in \mathbb{G}_1$ and $a, b \in \mathbb{Z}_p$, we have $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$.*
2. *Non-degeneracy: If g is a generator of $\mathbb{G}_1$, then $e(g, g) / \neq 1$, where 1 is the unity element of $\mathbb{G}_t$.*
3. *Computability: $\forall g_1, g_2 \in \mathbb{G}_1, e(g_1, g_2)$ can be computed in polynomial time.*

3.2 Dynamic Symmetric Searchable Encryption

We briefly introduce DSSE based on [1]. A database $DB = (ind_i, W_i)_{i=1}^{D}$ is a tuple of index/keywords pairs with $ind_i \in \{0,1\}^l$, $W_i \in \{0,1\}^*$, where D is the number of documents in DB. $W = \bigcup_{i=1}^{D}$ denote the total number of keywords in database DB. Let $N = \sum_{i=1}^{D} |W_i|$ be the number of document/keyword pairs. $DB(w)$ is the set of the document containing keyword w. Dynamic searchable encryption can be denoted by $\Pi = (Setup, Search, Update)$. Among them, $Setup$ is an algorithm, $Search$ and $Update$ are two protocols between server and client. Π can be described as follows:

- $Setup(DB)$ is used to initialize and start a system. It takes as input a plaintext database DB, and outputs (EDB, K, σ). EDB is the encrypted database, K is the secret key and σ is client's state.
- $Search(EDB, K, q, \sigma)$ is a protocol between the server and the user. The user first uses his secret key K, state σ and a search query $q = \{w_1, \cdots, w_l\}$ to generate a search token, then he sends the token to the server. Server runs the match algorithm which takes as input search token and its encrypted database EDB, then sends back the search result to the client.
- $Update(EDB, K, \sigma, op, in) = (Update_C(K, \sigma, op, in), Update_S(EDB))$ is a protocol between the user and the server. $Update_C$ is run by the user. The $Update_C$ takes as input the key K, state σ, operation op and input in parsed as the index ind and a set of keywords W_{ind} and output a update token. The $Update_S$ is run by the server and takes as input EDB. The operation op is taken from the set $\{add, del\}$, meaning the addition and deletion operations of a document/keyword pair.

3.3 Security Definition

It is difficult to obtain efficiency when design SSE schemes achieving high security. Therefore, when designing a SSE scheme, some information will be appropriately allowed to leak to get higher efficiency. Let $\mathcal{L} = \{\mathcal{L}_{Setup}, \mathcal{L}_{Update}, \mathcal{L}_{Search}\}$ be a set of predefined leakage functions. We can call a SSE scheme is secure if no more information is leaked than $\mathcal{L}$. Let $\Pi = (Setup, Update, Search)$ be a dynamic searchable symmetric encryption scheme, $\mathcal{A}$ be an adversary, λ be a security parameter. We can define the security of DSSE scheme by these two experiments [14]:

- $\mathbf{Real}_{\mathcal{A}}^{\Pi}(\lambda)$: The adversary $\mathcal{A}$ chooses database DB, then the experiment runs $Setup(DB)$ and returns EDB to $\mathcal{A}$. $\mathcal{A}$ adaptively chooses and executes a list of query $\{q_i\}$. If q_i is a search query, then the experiment runs $Search(EDB, K, q_i, \sigma)$ and returns the result. If q_i is an update query, the experiment answers the query by running $Update(EDB, K, \sigma, op, in)$. Finally, the adversary $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.
- $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\Pi}(\lambda)$: The adversary $\mathcal{A}$ chooses a database DB, then the experiment runs $\mathcal{S}(\mathcal{L}_{Setup}(DB))$ with the leakage $\mathcal{L}_{Setup}(DB)$ and returns EDB to $\mathcal{A}$. Then the adversary adaptively chooses and executes a list of query $\{q_i\}$. If q_i is a search query, then the experiment runs $\mathcal{S}(\mathcal{L}_{Search}(q_i))$ with leakage $\mathcal{L}_{Search}(q_i)$ and returns the result. If q_i is an update query, the experiment runs $\mathcal{S}(\mathcal{L}_{Update}(q_i))$ with leakage $\mathcal{L}_{Update}(q_i)$. Finally, the adversary $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.

Definition 2. *Π is called $\mathcal{L}$-adaptively-secure SSE scheme, if for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that:*

$$Pr(\mathbf{Real}_{\mathcal{A}}^{\Pi}(\lambda) = 1) - Pr(\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\Pi}(\lambda) = 1) \leq negl(\lambda)$$

where $negl(\lambda)$ is a negligible function.

Definition 3 *(Forward privacy [14]). A $\mathcal{L}$-adaptively-secure SSE scheme Π is forward secure if the update leakage function $\mathcal{L}_{Update}$ can be written as*

$$\mathcal{L}_{Update}(q_i) = (i, op_i, ind_i)$$

where $q_i = (ind_i, w_i, op_i)$ is an update query, op_i is an update operation, and ind_i is a document index. This definition means that an update query leaks no information about the update keyword.

3.4 Result Pattern Leakage

Result Pattern (RP) Leakage [11] leakage is a kind of leakage in SSE protocols, i.e. the files matching one search token. In conjunctive-keyword setting, the ideal RP leakage is the files matching all the keywords in one search operation, i.e. *the Whole Result Pattern (WRP)*. But most existing conjunctive-keyword search SSE scheme leak more information than WRP. For example, if user wants to search the files containing keywords $w_1, \cdots, w_t$. In OXT, there are three types of RP leakage:

- Single Keyword Result Pattern (SKRP), i.e. the server can know $DB(w_1)$
- Keyword-Pair Result Pattern (KPRP), i.e. the server can know $DB(w_1, w_i), 2 \leq i \leq t$
- Multiple Keyword Cross-Query Intersection Result Pattern(IP), i.e. the server can know $DB(w_1, \{w_i\})$ where $\{w_i\} \subseteq \{w_2, \cdots, w_t\}$

3.5 Asymmetric Scalar-Product-Preserving Encryption [17]

An ASPE scheme can be denoted by $\Pi = \{Setup, Encrypt, KeyGen, Decrypt\}$. Π can be described as follows:

- **Setup(λ).** This algorithm takes as input secure parameter λ and outputs the secret key K.
- **Encrypt($\boldsymbol{v}$, K).** This algorithm takes as input the message $\boldsymbol{v}$, a vector, and the secret key K, and outputs the ciphertext $\boldsymbol{c}$.
- **KeyGen($\boldsymbol{q}$, K).** This algorithm takes as input the query vector $\boldsymbol{q}$ and the secret key K, and outputs the search key $\boldsymbol{d}$.
- **Decrypt($\boldsymbol{c}$, $\boldsymbol{d}$).** This algorithm takes as input the ciphertext $\boldsymbol{c}$ and the search key $\boldsymbol{d}$ and outputs the inner product of $\boldsymbol{v}$ and $\boldsymbol{q}$.

We call Π is a ASPE scheme if and only if Π satisfies the following conditions:

1. $\boldsymbol{v} \cdot \boldsymbol{q} = Decrypt(\boldsymbol{c}, \boldsymbol{d})$ for any message vector $\boldsymbol{v}$ and any query vector $\boldsymbol{q}$
2. $\boldsymbol{v}_i \cdot \boldsymbol{v}_j \neq Decrypt(\boldsymbol{c}_i, \boldsymbol{c}_j)$ for any message vector $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$.

4 Our Scheme

In this section, we give the construction of our scheme. We design our scheme based on [7] and [20] which are conjunctive-keyword FSSE scheme and verifiable FSSE scheme supporting single-keyword search, respectively. We use a more efficient cryptographic tool to integrate these two schemes and then construct our more efficient conjunctive-keyword FSSE scheme.

4.1 Construction

Let $\mathbb{G}_1$, $\mathbb{G}_t$ be two cyclic groups with the same prime order p. g is a generator of $\mathbb{G}_1$. Let $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_t$ be a bilinear map. Let N be the cardinality of predefined ordered keyword space W, i.e. $N = |W|$. $PRPV(k, \boldsymbol{v})$ is a pseudo-random permutation. It can randomly reorder the elements of the vector $\boldsymbol{v}$, and its reordering rules are determined by k. We use the ASPE scheme proposed in [17] to construct our scheme. We express op with a bit length, i.e. $op \in \{add, del\} = \{1, 0\}$. The details of our scheme are shown in the Algorithm 1. The scheme contains four parts: *Setup*, *Derive*, *Update* and *Search*. They can be described as follows:

- $Setup(1^\lambda)$. This algorithm is run by PKG to initialize and start the system. First, PKG predefine an ordered keyword space W. With the cyclic groups $\mathbb{G}_1$, $\mathbb{G}_t$ and bilinear map e, PKG first randomly samples $(s, a) \xleftarrow{\$} \mathbb{Z}_p$ as its master key msk. Then PKG generates the public parameter $pp = \{N, p, g, g_1, H_1, H_2, H_3, F_p, F_1, F_2\}$ where $H_1, H_3 : \{0,1\}^* \rightarrow \{0,1\}^\lambda$, $H_2 : \{0,1\}^* \rightarrow \{0,1\}^{2\lambda+1}$, $F_i : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ and $F_p : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \mathbb{Z}_p$. Finally, PKG publishes the public parameter pp, and keeps the master key msk.

- $Derive(msk, U_{id})$. This is a protocol between the user and the PKG. First, an authenticated user U_{id} sends a registration request to PKG. After receiving the request, the PKG randomly generates two elements $(x, y) \xleftarrow{\$} \mathbb{Z}_p$ and computes $k_1 = g^x, k_2 = g^y$, and then sends (k_1, k_2) to U_{id}. After receiving the key (k_1, k_2), U_{id} randomly generates an integer $k_3 \xleftarrow{\$} \mathbb{Z}_p$. Finally, U_{id} keeps $sk_{id} = (k_1, k_2, k_3)$ as his secret key.
- $Update(ind, (w_{t_1}, w_{t_2}, \cdots, w_{t_l}), sk_{id}, \Sigma, T, op)$. This is a protocol between the user and the server where Σ is a map saved by the user to store user's latest local status, and T is a map saved by the server to store the encrypted keyword/file pairs. When a user U_{id} wants to update (add or delete) a file ind containing keywords $(w_{t_1}, w_{t_2}, \cdots, w_{t_l})$, he first generates the key k_s and then runs the $ASPE.Setup(seed)$ to get a secret key k_a for the ASPE scheme. Next, the user needs to initialize a vector $\boldsymbol{v}$ (line 4 to 8 in update protocol) where t_i represents the position of the keyword w_{t_i} in the keyword space W. Note that we set $\boldsymbol{v}[N+1] = -1$ and $\boldsymbol{v}[N+2] = 1$. Next, the user randomly shuffles the elements of the vector $\boldsymbol{v}$ by running $PRPV(k_s, \boldsymbol{v})$ and then encrypts the vector $\boldsymbol{v'}$ by running $ASPE.Encrypt(k_a, \boldsymbol{v'})$. Then he can generate a node for the chain corresponding to each keyword $w_i \in W_{ind}$. For each keyword $w_i \in W_{ind}$, the user needs to compute t_w, e and u first. Note that if $st_c = \Sigma[w_i]$ is $\perp$, we set $e = H_2(t_w||st_1) \oplus (\perp ||op||ind)$, otherwise we set $e = H_2(t_w||st_{c+1}) \oplus (st_c||op||ind)$. The symbol $\perp$ is used to mark the beginning of the current chain. After getting t_w, e, u, the user generates a key k_p to shuffle the elements of ciphertext vector $\boldsymbol{cv}$ and then hides $\boldsymbol{cv'}$ by doing $\boldsymbol{cv'} \oplus mc$. Finally, the user sends the update token $u, e, \boldsymbol{mcv}$ to the server. After receiving the update token, the server stores it into the encrypted database T.
- $Search((w_{t_1} \wedge w_{t_2} \wedge \cdots \wedge w_{t_m}), sk_{id}, \Sigma, T)$. This is a protocol between the user and the server. When a user U_{id} wants to search the documents containing keywords $W_{search} = (w_{t_1}, w_{t_2}, \cdots, w_{t_m})$, he first searches for the st_c corresponding to w_{t_1} in map Σ. If there is no state st_c, i.e. $st_c = \perp$, the user can sure that there is no file containing the keyword w_{t_1} in the server. If $st_c \neq \perp$, the user generates k_s, k_p, k_a and a vector $\boldsymbol{q}$ like the update protocol. The difference is we set $\boldsymbol{q}[N+1] = m$ and $\boldsymbol{q}[N+2] = F_p(t_w, st_c)$ where $m = |W_{search}|$. Then the user shuffles the elements of the vector $\boldsymbol{q}$ and generates a search key by running $ASPE.KeyGen(k_a, \boldsymbol{q'})$. After shuffling the search key $\boldsymbol{cq}$ using k_p, the user sends the search token $t_w, st_c, \boldsymbol{cq'}$ to the server. After receiving the search token, the server uses t_w and st_c to traverse the encrypted keyword/file pair chain corresponding to w_{t_1}. For each node in the chain (representing an update operation), the server can get e and $\boldsymbol{mcv}$. Then the server can get (st_{c-1}, op, ind) from e where st_{c-1} is the previous state of keyword w_{t_1}. The server determines whether the update is performing an add operation or a delete operation through the op. If the current node is an add operation, the server calculates the ciphertext vector $\boldsymbol{cv'}$ and runs $ASPE.Decrypt(\boldsymbol{cv'}, \boldsymbol{cq'})$ to get $st_c{}'$ which is the inner product of $\boldsymbol{v}$ and $\boldsymbol{q}$. If $st_c{}' = F_p(t_w, st_c)$, then the server can determine that current file ind contains all the keywords in

W_{search} and then puts this file into the result set R. If the current node is a delete operation, the server checks whether the current file is included in the result R, and deletes if it contains. Finally, the server sends the result set R to the user.

Correctness. First, as long as k_p is the same, we have the following equation: $ASPE.Decrypt(\boldsymbol{cv}', \boldsymbol{cq}') = ASPE.Decrypt(\boldsymbol{cv}, \boldsymbol{cq})$. Because if we use the same rules to shuffle the elements in two vectors, this does not change their inner product. Second, in the search protocol, the server can determine the current file *ind* contains all the keywords in W_{search} by $st_c{}' = F_p(t_w, st_c)$. Because if the file *ind* contains all the keywords in W_{search}, then $W_{search} \subset W_{ind}$. So there are m elements in the same position in $\boldsymbol{v}$ and $\boldsymbol{q}$ equal to 1. So the inner product of $\boldsymbol{v}$ and $\boldsymbol{q}$ is $m + (-m) + F_p(t_w, st_c) = F_p(t_w, st_c)$. If the file *ind* does not contain all the keywords in W_{search}, then there are less than m elements in the same position in $\boldsymbol{v}$ and $\boldsymbol{q}$ equal to 1. So the inner product of $\boldsymbol{v}$ and $\boldsymbol{q}$ is $m' + (-m) + F_p(t_w, st_c) \neq F_p(t_w, st_c)$, where $m' < m$.

4.2 How to Deploy

How to Support Multi-user. The basic idea we use to support multi-user is based on [18] which design a lightweight secret sharing method to share the secret key with another user. In our scheme, all keys can be generated by the key k_s. If a user U_1 wants to search his data stored in the cloud, he can generate the key k_s^1 using his secret key sk_1. With the k_s^1, he can generate the search token.

In the multi-user setting, let U_1 and U_2 be two different users with secret key $sk_1 = (g^{x_1}, g^{y_1}, k_3^1)$ and $sk_2 = (g^{x_2}, g^{y_2}, k_3^2)$. If user U_2 wants to search U_1's documents containing keywords $\{w_{t_1}, w_{t_2}, \cdots, w_{t_m}\}$, he needs U_1's key k_s^1 and the state $st_c = \Sigma[w_{t_1}]$. For the key k_s^1, U_2 can submit a request to U_1, then U_1 responds with $(g_1^{k_3^1}, g^{k_3^1})$. Then U_2 can calculate k_s^1, because of the following equation:

$$(e(g, k_2^1)/e(g_1, k_1^1))^{k_3^1} = e(g, g)^{k_3^1(y_1 - ax_1)} = e(g, g)^{k_3^1 s} = k_s^1$$
$$e(g^{k_3^1}, g^{y_2})/e(g_1^{k_3^1}, g^{x_2}) = e(g, g)^{k_3^1(y_2 - ax_2)} = e(g, g)^{k_3^1 s} = k_s^1$$

For st_c, U_1 can share it with U_2 through a public channel as follows: After getting the key k_s^1, U_2 generates a secret key of a symmetric encryption algorithm using k_s^1. U_1 also can generate the same secret key using his own key k_s^1. Then U_2 and U_1 can communicate securely.

How to Compress Vector Dimension. The disadvantage of our scheme 1 is that we need to predefined the keyword space W. This condition limits the practicality of our scheme. In addition, the efficiency of the ASPE scheme decreases as the keyword space expands, i.e. the dimension expands. In Algorithm 1, there are three points: the keyword space W can be sorted, the position of each keyword can be determined, and each document contains few keywords. So using a vector with dimension $|W|$ to record all the keywords' position in keywords

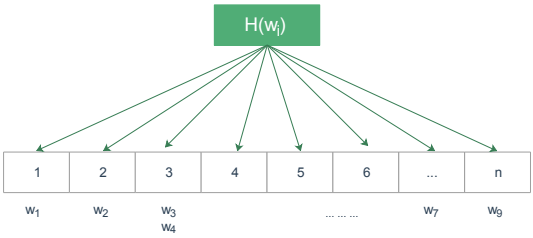

Fig. 2. Vector compression algorithm

space is extremely wasteful, because most elements of this vector is set to 0. We can save storage and computing resources by compressing this vector. The compression method is shown in Fig. 2. All we need to do is select a vector with a smaller dimension and then map the keywords to the elements of this vector. This compressing algorithm is a probabilistic algorithm. If you can accept errors (probably return a few extra files) with a small probability, you can use this method to compress the vector dimensions without losing too much accuracy.

5 Security Analysis

Informally, the security of our scheme depends on the security of the hash function, pseudorandom function and ASPE scheme. To support conjunctive-keyword search, we add an extra ciphertext into the update token, and then the server can do conjunctive-keyword search on the encrypted database. So as long as this extra ciphertext does not leak any information about the keywords of the newly added file, our scheme can guarantee the forward security. Let $W_{ind} = \{w_{t_1}, \cdots, w_{t_l}\}$ be the keywords in a file ind, and $W_{search} = \{w_{t_1}, \cdots, w_{t_m}\}$ be the keywords in a search query. For a search query q, the user always uses w_{t_1} to get t_w and st_c. We call this kind of keyword as $stag$. For a search query q, let $stag = Stag(q)$. Let $Hist = \{(DB_i, q_i)\}_{i=0}^{Q}$, where q_i is an update query or search query, DB_i is a snapshot of the database DB. For a search query W_{search}, let $W_s = \{w_i | w_i \in W_{search}, w_i \in W_{ind}\}$. If the ASPE scheme in [17] is secure, the leakage of our scheme can be described as follows:

1. Search Pattern:

$$sp(w_{t_1}) = \{i | Stag(q_i) = w_{t_1}, q_i \in Hist\}$$

2. Update History:

$$uh(w_{t_1}) = \{(i, op, ind_i) | q_i \ is \ a \ search \ query, Stag(q_i) = w_{t_1}\}$$

3. Whole Result Pattern:

$$WRP(W_{search}) = \{ind_i | W_{search} \subseteq W_{ind_i}, q_i \ is \ a \ update \ query\}$$

We can get the following theorem:

Theorem 1 *Let H_1, H_2, H_3 be three secure hash functions, and F, F_p be two secure pseudorandom function. We define the leakage function $\mathcal{L} = (\mathcal{L}_{setup}, \mathcal{L}_{search}, \mathcal{L}_{Update})$ as:*

$$\begin{aligned} \mathcal{L}_{setup} &= \perp \\ \mathcal{L}_{search}(W_{search}) &= (sp(w_{t_1}), uh(w_{t_1}), WRP(W_{search})) \\ \mathcal{L}_{Update}(i, op_i, W_{ind_i}, ind_i) &= (i, op_i, ind_i) \end{aligned}$$

Then the scheme 1 is is a $\mathcal{L}$-adaptively-secure dynamic SSE with forward privacy.

On conjunctive-keyword and single-user setting, our scheme has a similar structure to Hu's scheme [7], which has proven to be $\mathcal{L}$-adaptively-secure. So the security of our scheme on the single-user setting is based on the security of ASPE scheme. Detailed proof process proves that the scheme is indistinguishable from a simulated scheme, i.e. the security definition described in Sect. 2.

Unlike the cryptography tool used in Hu's scheme, the ASPE scheme is conceptually simple and efficient, but it doesn't have a high-security level. In fact, ASPE is proved to be not a CPA-secure scheme [12]. But this does not affect the forward security of our scheme. Because in update protocol, we masked $\boldsymbol{cv}'$ with mc, and mc is associated with st_{c+1}. Without the latest status st_{c+1}, the server can not get the correct ciphertext $\boldsymbol{cv}'$. That guarantees the forward security of our scheme. If the adversary breaks the ASPE scheme, he can only get the vectors $\boldsymbol{v}'$ and $\boldsymbol{q}'$ in a old update and search query, where $\boldsymbol{v}'$ and $\boldsymbol{q}'$ are vectors reordered by $PRPV(\cdot, \cdot)$. So the adversary cannot obtain the plaintext of the keywords W_{ind} and W_{search}. But the RP leakage of this scheme will include the SKRP, KPRP and IP leakage.

6 Efficiency Analysis and Performance Evaluation

In this section, we are going to compare the complexity of our scheme with the enhanced scheme in [7] and the FOXT-B in [16], and code programs to test the efficiency of our scheme.

6.1 Efficiency Comparison

The notations used in this subsection are shown in Table 2.

The Table 3 shows the comparison result of communication complexity. And the Table 4 shows the comparison result of computational complexity.

6.2 Experiment Results

In this subsection, we use a set of data to test the efficiency of search and update protocol in our scheme. We run the program on a personal computer with Windows10 OS, Intel(R) Core(TM) i7-9750H CPU 2.60 GHz, and 4 GB of

Table 2. Notations and description

Notation	Description
l	The number of keywords in one file, i.e. $l = \|W_{ind}\|$
m	The number of keywords in one search query, i.e. $m = \|W_{search}\|$
N	The size of the predefined keyword space, i.e. $N = \|W\|$
c	The average size of $DB(w)$
$\|\mathbb{Z}_p\|$	The bit size of the element of $\mathbb{Z}_p$
$\|\mathbb{G}_1\|$	The bit size of the element of $\mathbb{G}_1$
P	The computation cost of a bilinear pairing operation
E	The computation cost of an exponentiation operation in cyclic group $\mathbb{G}_1$ and $\mathbb{G}_t$
H	The computation cost of hash function and pseudorandom function
$A.S$	The computation cost of $ASPE.Setup$
$A.E$	The computation cost of $ASPE.Encrypt$
$A.K$	The computation cost of $ASPE.KeyGen$
$A.D$	The computation cost of $ASPE.Decrypt$
π	The computation cost of trapdoor permutation function
$I.E$	The computation cost of $IPE.Encrypt$
$I.K$	The computation cost of $IPE.KeyGen$
$I.D$	The computation cost of $IPE.Decrypt$

Table 3. Comparison of communication complexity

Scheme	Update token	Search token
Hu's scheme	$(N+1)\cdot\|\mathbb{G}_1\|+2\lambda$	$2(\lambda+\|\mathbb{Z}_p\|)+(N+1)\|\mathbb{G}_1\|$
FOXT-B	$2\lambda+\|\mathbb{Z}_p\|+\|\mathbb{G}_1\|$	$2\lambda+\mathbb{Z}_p+c(m-1)\|\mathbb{G}_1\|$
Our scheme	$3\lambda+1+2(N+2)\cdot\|\mathbb{Z}_p\|$	$2\lambda+2(N+2)\cdot\|\mathbb{Z}_p\|$

As we know, $|\mathbb{G}_1|$ is much larger than $\mathbb{Z}_p$. According to the result of this table, if N is not small, the size of update token in our scheme is small than Hu's schemes and large than FOXT-B, and the size of search token in our scheme is small than Hu's schemes and FOXT-B.

DDR4 RAM. We use the JPBC library for Java and choose the type A pairing to implement the pairing function. Let $\lambda = 256$ be the security parameter. We choose SM3 as hash functions H_i and let $F_i(key, data) = Hash(key||data)$. According to the complexity analysis in Subsect. 6.1, N, l and c are the main factors affecting the efficiency of update and search protocol. So we test how the efficiency change with these variables. The results are described as follows.

Table 4. Comparison of computational complexity

Scheme	Update.client	Search.client	Search.server
Hu's scheme	$l(4H + \pi) + I.E$	$H + I.K$	$c(3H + \pi + I.D)$
FOXT-B	$l(5H + \pi + E)$	$H + c(m - 1)E$	$c(H + m \cdot E + \pi)$
Our scheme	$2P + E + 5l \cdot H + A.S + A.E$	$2P + E + 3H + A.K$	$c(4H + A.D)$

According to the result of this table, our scheme is more efficient than FOXT-B and Hu's enhanced scheme. Because ASPE scheme is much more efficient than IPE scheme, and we do not need to compute a lot of exponentiation operation in the cyclic group. The efficiency comparison between ASPE and IPE are shown in the Table 5.

Table 5. The efficiency of IPE and ASPE

N	Client side in search		Server side in search		Client side in update	
	I.K	A.K	I.D	A.D	I.E	A.E
5	0.8 ms	0.0003 ms	9.9 ms	0.00008 ms	2.6 ms	0.00037 ms
10	1.2 ms	0.0007 ms	24.1 ms	0.00016 ms	4.5 ms	0.00081 ms
30	2.4 ms	0.0045 ms	67.1 ms	0.00023 ms	12.5 ms	0.00514 ms
50	4.0 ms	0.0117 ms	110.2 ms	0.00030 ms	20.7 ms	0.01343 ms
100	9.8 ms	0.0446 ms	217.8 ms	0.00058 ms	43.2 ms	0.05144 ms
250	40.9 ms	0.2624 ms	540.9 ms	0.00128 ms	124.4 ms	0.03192 ms
500	140.6 ms	1.0726 ms	1100 ms	0.00229 ms	310.5 ms	1.86808 ms
750	303.7 ms	2.3842 ms	1600 ms	0.00334 ms	555.9 ms	4.46054 ms

The testing result of IPE is come from paper [10]. This comparison is reasonable because the operating environment in [10] is better than ours.

Figure 3 shows how the efficiency of the update protocol changes with the variables l and N. In these two subgraph, we let $N = 100$ for the left one and $l = 10$ for the right one, respectively. Note that the second curve is close to a straight line, but it is not, because of the computation complexity of ASPE.Encrypt is $\mathcal{O}(n^2)$. This result satisfies the complexity analysis in the Table 4. Figure 4 shows how the efficiency of the search protocol changes with the variables c and N. In these two subgraph, we let $N = 100$ for the left one and $c = 100$ for the right one, respectively. The second curve also is close to a straight line, but it is not, because of the computation complexity of ASPE.KeyGen is $\mathcal{O}(n^2)$. This result also satisfies the complexity analysis in the Table 4.

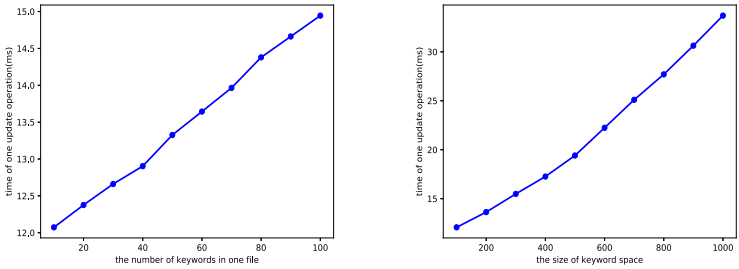

Fig. 3. Efficiency of update protocol

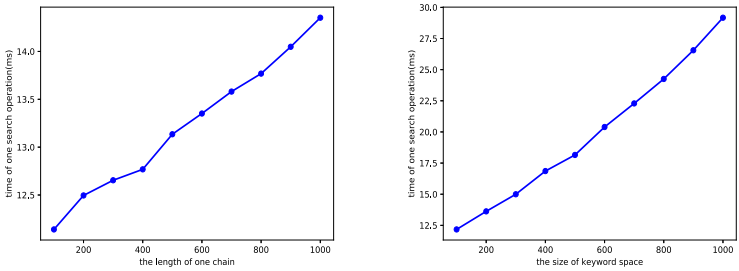

Fig. 4. Efficiency of search protocol

These test results prove that the efficiency of update protocol is linear with l, and search protocol is linear with c. Both the efficiency of update and search protocol is linear with N^2. However, due to the fast running speed of ASPE, if N grows not very large, the growth of N has little effect on the efficiency of update and search protocol.

Algorithm 1. Conjunctive-keyword FSSE using ASPE Scheme

$Setup(1^\lambda)$

$(s, a) \xleftarrow{\$} \mathbb{Z}_p$
$g_1 = g^a$
$msk = \{s, a\}$
$pp = \{N, p, g, g_1, H_1, H_2, H_3, F_p, F_1, F_2\}$
PKG keeps msk as master key

$Derive(msk, U_{id})$

PKG:
$(x, y) \xleftarrow{\$} \mathbb{Z}_p$, where x, y satisfy $y = ax + s(mod\ p)$
$k_1 = g^x, k_2 = g^y$
send k_1, k_2 to the user
Client:
$k_3 \xleftarrow{\$} \mathbb{Z}_p$
the user keeps $sk_{id} = (k_1, k_2, k_3)$

$Update(ind, (w_{t_1}, w_{t_2}, \cdots, w_{t_l}), sk_{id}, \Sigma, T)$

Client:
$k_s = (e(g, k_2)/e(g_1, k_1))^{k_3}$
$seed = H_1(k_s)$
$k_a = ASPE.Setup(seed)$
$\boldsymbol{v} = \mathbf{0}$ where $\boldsymbol{v} \in \mathbb{Z}_p^{N+2}$
for $i \in \{t_1, \cdots, t_l, N+2\}$ **do**
  $\boldsymbol{v}[i] = 1$
end for
$\boldsymbol{v}[N+1] = -1$
$\boldsymbol{v}' = PRPV(k_s, \boldsymbol{v})$
$\boldsymbol{cv} = ASPE.Encrypt(k_a, \boldsymbol{v}')$
for $i \in \{t_1, \cdots, t_l\}$ **do**
  $t_w = F_1(k_s, w_i)$
  $st_c = \Sigma[w_i]$
  if $st_c = \perp$ **then**
    $st_1 \xleftarrow{\$} \{0,1\}^\lambda$
    $e = H_2(t_w||st_1) \oplus (\perp ||op||ind)$
  else
    $st_{c+1} \xleftarrow{\$} \{0,1\}^\lambda$
    $e = H_2(t_w||st_{c+1}) \oplus (st_c||op||ind)$
  end if
  $\Sigma[w_i] = st_{c+1}$
  $u = H_3(t_w||st_{c+1})$
  $k_p = F_2(k_s, w_i)$
  $\boldsymbol{cv}' = PRPV(k_p, \boldsymbol{cv})$
  $mc = F_p(st_{c+1}, t_w)$
  $\boldsymbol{mcv} = \boldsymbol{cv}' \oplus mc$
  send $(u, e, \boldsymbol{mcv})$ to the server
end for
Server:
$T[u] = (e, \boldsymbol{mcv})$

$Search((w_{t_1} \wedge w_{t_2} \wedge \cdots \wedge w_{t_m}), sk_{id}, \Sigma, T)$

Client:
$st_c = \Sigma[w_{t_1}]$
if $st_c = \perp$ **then**
  return ϕ
end if
$k_s = (e(g, k_2)/e(g_1, k_1))^{k_3}$
$k_p = F_2(k_s, w_{t_1})$
$seed = H_1(k_s)$
$k_a = ASPE.Setup(seed)$
$t_w = F_1(k_s, w_{t_1})$
$\boldsymbol{q} = \mathbf{0}$ where $\boldsymbol{q} \in \mathbb{Z}_p^{N+2}$
for $i \in \{t_1, \cdots, t_m\}$ **do**
  $\boldsymbol{q}[i] = 1$
end for
$\boldsymbol{q}[N+1] = m,\ \boldsymbol{q}[N+2] = F_p(t_w, st_c)$
$\boldsymbol{q}' = PRPV(k_s, \boldsymbol{q})$
$\boldsymbol{cq} = ASPE.KeyGen(k_a, \boldsymbol{q}')$
$\boldsymbol{cq}' = PRPV(k_p, \boldsymbol{cq})$
send $(t_w, st_c, \boldsymbol{cq}')$ to the server
Server:
initialize two empty set R, Δ
while $st_c \neq \perp$ **do**
  $u = H_3(t_w||st_c)$
  $(e, \boldsymbol{mcv}) = T[u]$
  $(st_{c-1}, op, ind) = H_2(t_w||st_c) \oplus e$
  if $op = del$ **then**
    $\Delta = \Delta \cup \{ind\}$
  else
    if $ind \in \Delta$ **then**
      $\Delta = \Delta \setminus \{ind\}$
    else
      $mc = F_p(st_c, t_w)$
      $\boldsymbol{cv}' = \boldsymbol{mcv} \oplus mc$
      $st_c' = ASPE.Decrypt(\boldsymbol{cv}', \boldsymbol{cq}')$
      if $st_c' = F_p(t_w, st_c)$ **then**
        $R = R \cup \{ind\}$
      end if
    end if
  end if
end while
send R to the client

7 Conclusion

In this paper, we analyze the feasibility of using the ASPE scheme to support multi-keyword search in FSSE scheme and construct a conjunctive-keyword FSSE scheme supporting multi-user. We analyze the security and efficiency of our scheme and how to deploy our scheme in the real scenario. Compared with existing conjunctive-keyword FSSE scheme, our scheme is more efficient and practical. For future work, we will try to design a more efficient, lower communication and storage overhead conjunctive-keyword FSSE scheme using Shamir threshold scheme.

References

1. Bost, R.: Σ οφος forward secure searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1143–1154 (2016)
2. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1465–1482 (2017)
3. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for Boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_20
4. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. J. Comput. Secur. **19**(5), 895–934 (2011)
5. Goh, E.J., et al.: Secure indexes. IACR Cryptol. ePrint Arch. **2003**, 216 (2003)
6. Golle, P., Staddon, J., Waters, B.: Secure conjunctive keyword search over encrypted data. In: Jakobsson, M., Yung, M., Zhou, J. (eds.) ACNS 2004. LNCS, vol. 3089, pp. 31–45. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24852-1_3
7. Hu, C., et al.: Forward secure conjunctive-keyword searchable encryption. IEEE Access **7**, 35035–35048 (2019)
8. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258–274. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_22
9. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 965–976 (2012)
10. Kim, S., Lewi, K., Mandal, A., Montgomery, H., Roy, A., Wu, D.J.: Function-hiding inner product encryption is practical. In: Catalano, D., De Prisco, R. (eds.) SCN 2018. LNCS, vol. 11035, pp. 544–562. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98113-0_29
11. Lai, S., et al.: Result pattern hiding searchable encryption for conjunctive queries. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 745–762 (2018)

12. Lin, W., Wang, K., Zhang, Z., Chen, H.: Revisiting security risks of asymmetric scalar product preserving encryption and its variants. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 1116–1125. IEEE (2017)
13. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceeding 2000 IEEE Symposium on Security and Privacy, S&P 2000, pp. 44–55. IEEE (2000)
14. Song, X., Dong, C., Yuan, D., Xu, Q., Zhao, M.: Forward private searchable symmetric encryption with optimized I/O efficiency. IEEE Trans. Depend. Secure Comput. **17**, 912–927 (2018)
15. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. NDSS **71**, 72–75 (2014)
16. Wang, Y., Wang, J., Sun, S., Miao, M., Chen, X.: Toward forward secure SSE supporting conjunctive keyword search. IEEE Access **7**, 142762–142772 (2019)
17. Wong, W.K., Cheung, D.W.L., Kao, B., Mamoulis, N.: Secure KNN computation on encrypted databases. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 139–152 (2009)
18. Xu, L., Xu, C., Liu, Z., Wang, Y., Wang, J.: Enabling comparable search over encrypted data for IoT with privacy-preserving. CMC-Comput. Mater. Continua **109**(2), 537–554 (2019)
19. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of file-injection attacks on searchable encryption. In: 25th {USENIX} Security Symposium, {USENIX} Security 2016, pp. 707–720 (2016)
20. Zhang, Z., Wang, J., Wang, Y., Su, Y., Chen, X.: Towards efficient verifiable forward secure searchable symmetric encryption. In: Sako, K., Schneider, S., Ryan, P.Y.A. (eds.) ESORICS 2019. LNCS, vol. 11736, pp. 304–321. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29962-0_15