Efficient Forward and Backward Private Searchable Symmetric Encryption for Multiple Data Sources

Lin Mei, Chungen Xu[✉], and Lin Li

School of Science, Nanjing University of Science and Technology, Nanjing 210094, China
xuchung@njust.edu.cn

Abstract. Searchable symmetric encryption (SSE) has been widely applied in the encrypted database for keyword queries. Although SSE is powerful and feature-rich, it is always plagued by information leaks. Some recent researches have pointed out that forward and backward privacy which disallows leakage from update operations should be the basic requirement for a secure SSE scheme. However, most existing forward and backward private SSE schemes only consider the single data source model which is not practical in the IoT scenario (e.g., data are often separately distributed over multiple devices). Considering the above issues, this paper proposes an efficient forward and backward private SSE scheme for multiple data sources (FBSSE-MDS). As far as we know, FBSSE-MDS is the first efficient SSE scheme which supports both forward privacy and backward privacy BP-II (the second level of backward privacy) in the scene of multiple data sources. Finally, we implement our scheme and compare its performance with two other related schemes. The experiment results show that our scheme is highly efficient.

Keywords: Internet of Things · Searchable symmetric encryption · Forward privacy · Backward privacy · Multiple data sources

1 Introduction

Internet of Things (IoT) enables different devices to achieve convenient and efficient connections, during which tremendous data are collected and transferred via the internet. With the sharp increase of the massive data, outsourcing data to the IoT cloud has recently become prevalent. Despite the benefits of cloud storage, such as low cost and ubiquitous access, data privacy is a major concern. To dispel user's concern, encrypting data before uploading it to the untrusted cloud server is a straightforward solution. However, encryption hinders the usability

The authors would like to thank the support from the National Natural Science Foundation of China (No: 62072240), the National Key Research and Development Program of China (No. 2020YFB1804604).

of data, which results in common retrieval methods such as the keyword search fails to be directly executed over ciphertexts. To solve this problem, searchable symmetric encryption (SSE) was introduced in 2000 [16]. It allows a client to retrieve the outsourced encrypted files containing a certain keyword by submitting a cryptographically generated token. The original SSE schemes only provided secure search over static database, which seriously restrict the applicability due to the lack of update functionality. Dynamic searchable symmetric encryption (DSSE) was formally proposed by Kamara et al. [12] to allow users to perform update operations on the outsourced database.

At the core of designing SSE schemes is to improve efficiency while ensuring the security at the same time. The researches of the former goal is focused on storage requirements, bandwidth or latency. The latter faces more challenged issues since tremendous works have uncovered devastating and fairly generic attacks against many SSE schemes. For example, deterministic encryption used in SSE makes it easy for the malicious server to observe repeated queries and other information. These leakages typically include the search pattern that reveals which search queries refer to the same keyword as well as the access pattern that reveals which files are returned for a query. Generally, these leakages could be eliminated by using oblivious RAM (ORAM) [7]. However, applying ORAM brings heavy computational overhead and bandwidth cost. To gain a more practical counter measure, Islam et al. [9] proposed the first database padding approach and be further improved by Xu et al. [22].

Compared with SSE, DSSE introduces two additional privacy concerns, owing to the added functionality. The first is that newly updated files can be related to previous search results. The second is that search queries can leak matching files after they have been deleted. In 2016, Zhang [23] gave a more powerful attack named file injection attack, which shows how the leakages in DSSE can be exploited to reveal a considerable amount of information in practice. In this attack, 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 easily recover the keyword of a query.

The work in [23] underlines the importance of forward privacy (FP) when constructing DSSE schemes, which requires that the update (addition and deletion) operations cannot be linked to previous search queries and further ensures that the scheme can resist file injection attacks. The initial work that achieve FP is proposed by Chang and Mitzenmacher, which is designed by utilizing pseudo-random functions [4]. However, in their scheme the size of search query grows linearly in the number of updates, which means there exists a threshold for update, and the communication cost for the search operation will become unacceptably high once the threshold is exceeded. In 2016, Bost [1] creatively proposed an efficient forward secure SSE scheme named $\sum o\varphi o\varsigma$, which only relies on trapdoor permutations. Since then, several schemes have been proposed to achieve FP using different cryptographic primitives [6, 13, 17, 18].

Regarding data deletion, most past schemes cannot resist the server from learning that the new document has a keyword user searched for in the past

[1,6,13,17,18]. To hide the relationship between the deleted files and the query, Bost [2] introduced a formal definition of backward privacy (BP) with three different types of leakage ordered from most to least secure (from BP-I to BP-III). Besides, Bost [2] provided four backward-private constructions that achieve different privacy/efficiency trade-offs. Later, Chamani et al. [3] proposed an enhanced forward and backward secure scheme named MITRA, which offers backward-privacy Type-II.

As far as we know, most existing forward and backward private SSE schemes only consider the single data source model, that is to say, suppose that the searchable index can be directly built by the single data source. This assumption only makes sense when data files are extremely lightweight and stored centrally, which is not realistic in IoT. For example, industry IoT, as a specific application scenario of IoT, always involves multiple data sources (e.g., interconnected sensors, instruments and other devices networked together with computers' industrial applications). It is not rational for a company to centralize all the data and then build a searchable index for secure query. In order to address the mentioned issues, Liu et al. [15] proposed the notion of Multi-Data-Source (MDS) SSE, which allows each data source to build a local index individually and enables the storage provider to merge all local indexes into a global index afterwards.

Contributions. We propose an efficient and secure scheme FBSSE-MDS, which achieves both forward privacy and BP-II (the second level of backward privacy) in the scenario of multiple data sources. To the best of our knowledge, FBSSE-MDS is the first scheme which achieve all above properties. Also, we give the strict security proof of our scheme to show that our scheme is forward private and BP-II private. Through theoretical analysis, our scheme is demonstrated to own rich functionalities and high security while maintaining high efficiency. We implement our scheme and other related schemes using the Java programming language. The experimental results show that our scheme achieves better balance between security and efficiency.

Related Works: SSE was first introduced by Song et al. [16], with a scheme whose search time is linear in the number of documents. To improve efficiency, Curtmola et al. [5] gave the first index-based SSE constructions to achieve sublinear search time. Since then, many works has been done to enrich the functionality [12,14] and improve the security [1–3,6,13,17,18]. Among others, Kamara et al. [12] proposed the first dynamic SSE to support sublinear search with update operations, but it leaks the hashes of the unique keywords contained in the updated documents. Kamara and Papamanthou [11] later improved their construction by increasing the space complexity. Recently, a series of DSSE schemes have been proposed to offer varieties of functionalities [10] and improve efficiency [17] and security [18].

In 2014, Stefanov et al. Stefanov2014 first formalized the notion of FP for DSSE scheme. In 2016, Zhang et al. [23] gave a formalized definition of a very strong attack, named file injection attack. This attack can easily recover the keyword of a query by injecting only a small number of files in a DSSE scheme. Since then, several schemes have been proposed to achieve forward privacy using different cryptographic primitives, including Sophos [1] (uses trapdoor permutation (TDP)), Diana [2] (uses Constrained Pseudorandom Function (CPRF)), Dual [6] (uses keyed hash function), FSSE [21] (uses keyed-block chains), SGX-SE [20] (uses intel SGX) and VFSSE [8] (uses blockchain).

Very recently, Bost et al. [2] introduced a formal definition for backward privacy with three different types of leakage ordered from most to least secure (from BP-I to BP-III). Bost et al. [2] provided four backward private (and forward private) constructions that achieve different privacy/efficiency trade-offs. They first described a simple and generic method to transform a forward private SSE scheme to a backward private SSE scheme at the cost of an extra roundtrip per search query. Moneta and Fides are two instantiations of this method, while the security level of the latter is BP-II. Then they proposed two BP-III schemes named Dianadel and Janus that rely on puncturable cryptographic primitives to achieve better results, but increasing the amount of information leaked. Sun et al. [19] proposed a backward-secure SSE scheme from symmetric puncturable encryption. Compared to Janus (the first non-interactive backward-secure SSE scheme), the proposed construction in [19] is proved to be more practical through implementation.

1.1 Organization

The remainder of this paper is organized as follows. In Sect. 2, we state the system model, threat model and design goals of our scheme. In Sect. 3, we describe the cryptographic background for our construction. In Sect. 4, we introduce the details of our proposed scheme FBSSE-MDS. Then we carry out the security analysis and performance evaluation in Sect. 5 and Sect. 6, respectively. Finally, we conclude the paper in Sect. 7.

2 Problem Statement

2.1 System Model

As shown in Fig. 1, there are four roles in a FBSSE-MDS system: (1) data sources, denoted as DS, who own various collections of data files. (2) data user, denoted as DU, who issues search queries for interested keywords. (3) trusted authority, denoted as TA, who stores some valuable information to help DU and DS perform search or update operation. (4) cloud server, who stores encrypted data files and responses DU's search queries and DS's update operations. Note that the roles of DU and DS are interchangeable, which means an authorized DU who shares the secret key with DS is assumed to be a DS as well or any DS can be called DU when he performs a search query.

Figure 1 illustrates the architecture of our proposed FBSSE-MDS system. Firstly, DS encrypt data files and build searchable indexes before data outsourcing. Upon receiving all indexes from DS, server merges them into one searchable index. Then DU can query the encrypted data by generating a search token, which will later be submitted to the server. Finally, the server searches the index and returns the identities of data files containing searched keywords.

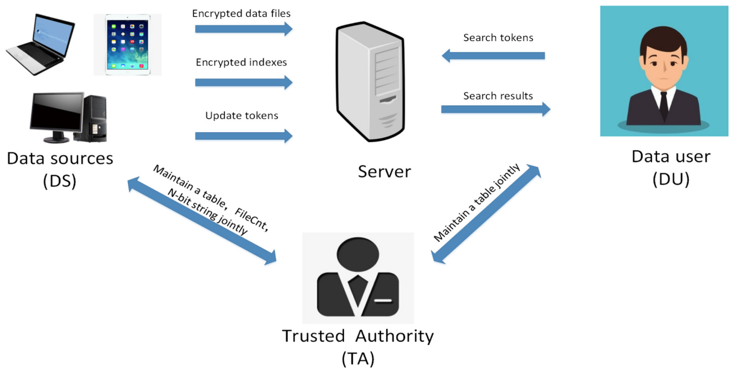


Fig. 1. System model

2.2 Threat Model

We treat the server as the adversary, who behaves "honest-but-curious". As its name implies, the server will follow all the operations defined in the FBSSE-MDS system model, while trying to deduce private information about the original data or searched keywords. Meanwhile, we suppose that DS, DU and TA are fully trusted, that is to say, there is no collusion among DS, Du and TA. Following most of the existing settings, the key is assumed to be transferred via a secure channel between DS and DU.

2.3 Design Goals

The proposed model should accomplish the following goals:

- **Providing forward privacy.** The server cannot violate the privacy of newly updated files by utilizing previously received search tokens. Each search token is associated with a state that is renewed every time files are updated. Therefore, the disclosure of the old token will not pose a threat to the updated files.
- **Providing BP-II backward privacy.** As for the deletion operation, the server cannot learn which deletion cancels which addition. In particular, the identity of files will be hidden by XOR operation. It ensures that FBSSE-MDS achieves BP-II backward privacy.
- **Supporting multiple data sources.** The server can merges indexes built by different *DS* that are indistinguishable into one index before performing search operations. Afterwards, the server can search over only one index rather than k indexes, which greatly enhances the efficiency.

3 Preliminaries

Forward privacy and backward privacy are two SSE properties that aim to control what information is leaked by dynamic schemes in relation to updates. Informally, a scheme is forward private if it is not possible to relate an update to previous search operations. This is particularly essential in practice, e.g., to hide whether an addition is about a new keyword or a pre-existing one (which may have been previously searched for).

Definition 1 (Forward Privacy). An $\mathcal{L}$-adaptively-secure SSE scheme that supports addition/deletion of a single keyword is forward private iff the update leakage function $\mathcal{L}^{Update}$ can be written as:

$$\mathcal{L}^{Update}(op, w, ind) = \mathcal{L}^{'}(op, ind),$$

where $\mathcal{L}^{'}$ is a stateless function, op is insertion or deletion, and ind is a file identifier.

An SSE scheme satisfies backward privacy if after deleting a document ind matching keyword w, the server cannot reveal the deleted document ind from the subsequent search of keyword w. In 2017, Bost et al. [2] have defined backward privacy at three levels: BP-I, BP-II and BP-III. The definitions of them can be described as follows.

Definition 2 (BP-I). A $\mathcal{L}$-adaptively-secure SSE scheme is insertion pattern revealing backward-private iff leakage functions $\mathcal{L}$ can be written as:

$$\mathcal{L}^{Update}(op, w, ind) = \mathcal{L}^{'}(op),$$

$$\mathcal{L}^{Search}(w) = \mathcal{L}^{''}(TimeDB(w)),$$

where $\mathcal{L}^{'}$ and $\mathcal{L}^{''}$ are stateless and $|TimeDB(w)| = a_w$ for a_w is a constant.

Definition 3 (BP-II). A $\mathcal{L}$-adaptively-secure SSE scheme is update pattern revealing backward-private iff leakage functions $\mathcal{L}$ can be written as:

$$\mathcal{L}^{Update}(op, w, ind) = \mathcal{L}^{'}(op, w), \mathcal{L}^{Search}(w) = \mathcal{L}^{''}(TimeDB(w), Updates(w)),$$

where $\mathcal{L}^{'}$ and $\mathcal{L}^{''}$ are stateless and $|TimeDB(w)| = a_w$ for a_w is a constant.

Definition 4 (BP-III). A $\mathcal{L}$-adaptively-secure SSE scheme is weakly backward-private iff leakage functions $\mathcal{L}$ can be written as:

$$\mathcal{L}^{Update}(op, w, ind) = \mathcal{L}^{'}(op, w),$$

$$\mathcal{L}^{Search}(w) = \mathcal{L}^{''}(TimeDB(w), DelHist(w)),$$

where $\mathcal{L}^{'}$ and $\mathcal{L}^{''}$ are stateless and $|TimeDB(w)| = a_w$ for a_w is a constant.

4 Our Construction

This section mainly introduces the specific structure of FBSSE-MDS scheme, which includes three protocols: setup, update and search. Among them, setup and update are the protocols running between multiple DS, TA and servers,

```
$DS_p (1 \leq p \leq k)$:
1: Randomly select $\mathcal{K}$ from $\{0,1\}^\lambda$
2: Initialize an empty collection $\mathcal{L}_p$
3: Initalize an empty list $O$
4: $FileCnt \leftarrow$ online $FileCnt$
5: for $i = FileCnt + 1$ to do
6:     $FileCnt + |\mathcal{D}_p|$
7:     $c_i \leftarrow ENC(\mathcal{K}, f_i)$
8:     $ID(f_i) \leftarrow PRP^{'}(\mathcal{K}, i)$
9:     $O[i - FileCnt] \leftarrow ID(f_i)$
10:    $\mathcal{L}_p \leftarrow \mathcal{L}_p \cup \{(c_i, ID(f_i))\}$
11: end for
12: Send $\mathcal{L}_p$ to the server via anonymous communication
13: $FileCnt \leftarrow FileCnt + |\mathcal{D}_p|$
14: Initialize an array $A$ of length $|\mathcal{W}|$
15: Initialize a counter $ctr \leftarrow 1$
16: $K_{permute} \leftarrow PRF(\mathcal{K}, 0)$
17: for $w \in \mathcal{W}$ do
18:     Initialize a counter $j \leftarrow 1$
19:     Initialize an N-bit zero string $\gamma$
20:     for $f \in \mathcal{D}_i$ do
21:         if $w \in f$ then
22:             $\gamma[\mathcal{F}[j]] \leftarrow 1$
23:         end if
24:         $j \leftarrow j + 1$
25:     end for
26:     $t_w \leftarrow PRF(\mathcal{K}, w), \alpha \leftarrow H_1(t_w||0)$
27:     $\beta \leftarrow PRF(\mathcal{K}, w||0)$, $s_{id} \leftarrow PRF^{'}(\beta, id)$
28:     $\gamma \leftarrow \gamma \oplus s_{id}$, $\delta \leftarrow H_2(t_w||0) \oplus \perp$
29:     $A[PRP(K_{permute}, ctr)] \leftarrow (\alpha, \gamma, \delta)$
30:     $ctr \leftarrow ctr + 1$
31: end for
32: Send $A$ to the server.

Server:
1: Initialize an empty dictionary $\mathcal{I}$.
2: for $i = 1$ to $k$ do
3:     Parse $A[i]$ as $(\alpha_{i,1}, \gamma_{i,1}, \delta_{i,1})$,
4:     $(\alpha_{i,2}, \gamma_{i,2}, \delta_{i,2}), ...(\alpha_{i,|\mathcal{W}|}, \gamma_{i,|\mathcal{W}|}, \delta_{i,|\mathcal{W}|})$
5: end for
6: for $j = 1$ to $|\mathcal{W}|$ do
7:     $\alpha \leftarrow \alpha_{1,j}$ ( note that $\alpha_{1,j} = ... = \alpha_{k,j}$)
8:     $\delta \leftarrow \delta_{1,j}$ ( note that $\delta_{1,j} = ... = \delta_{k,j}$)
9:     $\gamma \leftarrow \bigoplus_{i=1}^{k} \gamma_{i,j}$
10: end for
11: $\mathcal{I}[\alpha] \leftarrow (\gamma, \delta)$
12: $Z \leftarrow Z \cup \mathcal{L}_p$
```

Fig. 2. Setup protocol

while search is the protocol running between DU, TA and server. Now we give the detailed descriptions of three protocols as follows.

Setup. During the Setup protocol, each data source DS_p encrypts his data files and builds searchable indexes. The server receives encrypted data files and indexes from each DS_p and merges all indexes.

- DS_p: Take the security parameter λ and an empty collection $\mathcal{L}_p$, an empty list O, the total number of data files currently stored on the server $FileCnt$ as inputs. Let $PRP^{'} : \{0,1\}^\lambda \times \{1, ..., N\} \rightarrow \{1, ..., N\}$ be a pseudo-random permutation, and $H_1 : \{0,1\}^* \rightarrow \{0,1\}^\lambda$, $H_2 : \{0,1\}^* \rightarrow \{0,1\}^N$ be two hash functions, and $PRF : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$, $PRF^{'} : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^N$ be two pseudo-random functions. Run the first part of Setup protocol (as shown in Fig. 2) to compute the indexes.
- Server: Run the second part of Setup protocol (as shown in Fig. 2) to merge the received indexes into one index and store it.

Update. During the Update protocol, DS_p needs to submit encrypted data files and update tokens to the server, then the server updates index and stores encrypted data files in cipher collection.

- DS_p: Assuming the DS_p wants to update the files in set ID_{add} and ID_{del}, he runs line 1 to 16 of the first part in Update protocol to generate the ciphertexts and identity set (as shown in Fig. 3). Furthermore, in order to

```
DSp(1 ≤ p ≤ k):
1: Initialize an empty array $ID_{up}$ whose size is $|ID_{add}| + |ID_{del}|$
2: if $ID_{add} = \emptyset$ then
3:     Padding $Cipher$ with random elements
4: else
5:     for $j = 1$ to $|ID_{add}|$ do
6:         $c_j \leftarrow ENC(\mathcal{K}, f_j)$
7:         $ID(f_j) \leftarrow PRF^{'}(\mathcal{K}, FileCnt + j)$
8:         $Cipher \leftarrow Cipher \cup (c_j, ID(f_j))$
9:         $ID_{up}[j] \leftarrow ID(f_j)$
10:    end for
11: end if
12: if $ID_{del} \neq \emptyset$ then
13:     for $i = |ID_{add}|+1$ to $|ID_{add}|+|ID_{del}|$ do
14:         $ID_{up}[j] \leftarrow ID_{del}[i - |ID_{add}|]$
15:     end for
16: end if
17: Initialize a empty array $F$ whose size is $|\mathcal{W}_{up}|$
18: Initialize an empty collection $\mathcal{UT}_{\mathcal{W}_{up}}$
19: for $i = 1$ to $|\mathcal{W}_{up}|$ do
20:     $(UpdCnt, st_c) \leftarrow \mathcal{T}(\mathcal{W}_{up}[i])$
21:     $st_{c+1} \xleftarrow{\$} \{0,1\}^{\lambda}$, $t_w \leftarrow PRF(\mathcal{K}, w)$
22:     $\alpha \leftarrow H_1(t_w||st_{c+1})$
23:     $\delta \leftarrow H_2(t_w||st_{c+1}) \oplus st_c$
24:     Initialize a N-bit string $\gamma_{up}$ to 0
25:     for $j = 1$ to $|ID_{up}|$ do
26:         if the data file corresponding to $ID_{up}[j]$ contains keyword $\mathcal{W}_{up}[i]$ then
27:             set the $ID_{up}[j]$-th bit of $\gamma_{up}$ to 1
28:         end if
29:     end for
30:     $UpdCnt \leftarrow UpdCnt + 1$
31:     $\gamma \leftarrow \gamma_{up} \oplus PRF(\mathcal{K}, \mathcal{W}_{up}[i]||UpdCnt)$
32:     $\mathcal{UT}_{\mathcal{W}_{up}} \leftarrow \mathcal{UT}_{\mathcal{W}_{up}} \cup \{(\alpha, \gamma, \delta)\}$
33:     $\mathcal{T}(\mathcal{W}_{up}[i]) \leftarrow (UpdCnt, st_{c+1})$
34:     $FileCnt \leftarrow FileCnt + |D_{add}|$
35: end for
36: Send $(Cipher, \mathcal{UT}_{\mathcal{W}_{up}})$ to the server

server:
1: for $i = 1$ to $|\mathcal{W}_{up}|$ do
2:     $ID_{up}[j] \leftarrow ID_{del}[i - |\mathcal{D}_{add}|]$
3:     Parse $\mathcal{UT}_{\mathcal{W}_{up}}$ as $(\alpha_1, \gamma_1, \delta_1), \ldots,$
4: $(\alpha_{|\mathcal{W}_{up}|}, \gamma_{|\mathcal{W}_{up}|}, \delta_{|\mathcal{W}_{up}|})$
5:     $\mathcal{I}[\alpha_i] \leftarrow (\gamma_i, \delta_i)$
6:     $Z \leftarrow Z \cup Cipher$
7: end for
```

Fig. 3. Update protocol

ensure the searchability of the updated files, he computes the update tokens as line 17 to 35 of the first part in Update protocol. Finally, he sends them to the server.

– Server: Run the second part of Update protocol (as shown in Fig. 3) to update the ciphertexts and update tokens.

Search. During the Search protocol, DU issues the search token and then the server returns search results.

– *DU*: Assuming the *DU* wants to search the files containing keyword w, he runs line 1 to 9 of the first part in Search protocol to generate the search token (as shown in Fig. 4). Then, he sends it to the server.
– Server: After receiving the search token, the server runs the second part of Search protocol (as shown in Fig. 4) to obtain the result set and returns it to the DU.

5 Security Analysis

In this section, we analyze the security of our FBSSE-MDS scheme. Our scheme can achieve forward privacy and backward privacy BP-II. We first define the leakage functions in our scheme as follows:

```
DU:
1: Initialize an empty collection $\mathcal{ST}_w$
2: $(UpdCnt, st_c) \leftarrow \mathcal{T}[w]$
3: $t_w \leftarrow PRF(\mathcal{K}, w)$
4: $\beta_0 \leftarrow PRF(\mathcal{K}, w||0)$,
5: $s \leftarrow \bigoplus_{id=1}^{k} PRF'(\beta_0, id)$
6: for $i = 1$ to $UpdCnt$ do
7:     $\beta_i \leftarrow PRF(\mathcal{K}, w||i)$
8:     $s \leftarrow s \oplus \beta_i$
9: end for
10: Send $\mathcal{ST}_w = (t_w, st_c, s)$ to the server
server:
1: Initial an empty collection $ID(w)$, a $N$-bit
   string $\gamma'$ whose all bits are 0
2: Parse $\mathcal{ST}_w$ as $(t_w, st_c, s)$
3: while $st_c \neq \perp$ do
4:     $\alpha \leftarrow H_1(t_w||st_c)$
5:     $(\gamma, \delta) \leftarrow \mathcal{I}[\alpha]$
6:     $\gamma' \leftarrow \gamma' \oplus \gamma$
7:     $st_{c-1} \leftarrow H_2(t_w||st_c) \oplus \delta$
8:     $st_c \leftarrow st_{c-1}$
9: end while
10: $\gamma' \leftarrow \gamma' \oplus s$
11: for $i = 1$ to $N$ do
12:     if $\gamma'[i] = 1$ then
13:         $ID(w) \leftarrow ID(w) \cup \{i\}$
14:     end if
15: end for
16: Send $ID(w)$ to $DU$
```

Fig. 4. Search protocol

$$\mathcal{L}^{Setup}(\{\mathcal{D}_j\}_{1\leq j\leq k}, \mathcal{W}) = (k, n, N, \mathcal{W}), \mathcal{L}^{Update}(op, w, ind) = \perp$$
$$\mathcal{L}^{Search}(w) = \mathcal{L}'(\mathbf{TimeDB}(w), \mathbf{Updates}(w))$$

where $k, n, N, \mathcal{W}$ stands for the number of data sources, the total number of data files for initialization, the maximum number of update operations and the keyword universe respectively.

We are now ready to state the following theorem regarding the security of FBSSE-MDS.

Theorem 1. *If $H_1, H_2, PRF, PRF', PRP, PRP', SKE$ are secure cryptographic primitives, then our scheme FBSSE-MDS is an adaptively-secure SSE scheme with $\mathcal{L} = (\mathcal{L}^{Setup}, \mathcal{L}^{Update}, \mathcal{L}^{Search})$.*

Proof. For the adversary $\mathcal{A}$, challenger C and simulator S, we define the following two experiments: $\mathbf{Real}_{\mathcal{A},\mathcal{C}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},S}(\lambda)$.

$\mathbf{Real}_{\mathcal{A},\mathcal{C}}(\lambda)$: the challenger C generates a secret key $\mathcal{K} = KeyGen(1^\lambda)$. $\mathcal{A}$ chooses k collections of data files $D_1, ..., D_k$ containing n data files in total and a keyword universe $\mathcal{W}$. $\mathcal{A}$ receives $\{c_j\}_{1\leq j\leq n}, \{ID_j\}_{1\leq j\leq n}, \{\mathcal{I}_j\}_{1\leq j\leq k}$ such that c_j is an encrypted data file, ID_j is the identity of file c_j and $\mathcal{I}_j$ is the local index built by DS_j. Then $\mathcal{A}$ merges all indexes $\{\mathcal{I}_j\}_{1\leq j\leq k}$. Afterwards, $\mathcal{A}$ makes a polynomial number of adaptive queries. For each queried keyword w, $\mathcal{A}$ receives a search token $\mathcal{ST}_w$ or update token $\mathcal{UT}_w$ from the challenger C. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.

$\mathbf{Ideal}_{\mathcal{A},S}(\lambda)$: $\mathcal{A}$ chooses k collections of data files $D_1, ..., D_k$ containing n data files in total and a keyword universe $\mathcal{W}$. Given $\mathcal{L}^{Setup}(\{\mathcal{D}_j\}_{1\leq j\leq k}, \mathcal{W})$, S simulates and sends $\{c'_j\}_{1\leq j\leq n}$, $\{ID'_j\}_{1\leq j\leq n}$, $\{\mathcal{I}'_j\}_{1\leq j\leq k}$ to $\mathcal{A}$ and then $\mathcal{A}$ merge them. Afterwards, $\mathcal{A}$ makes a polynomial number of adaptive queries. For each queried keyword w, S receives $\mathcal{L}^{Query}(w)$ or $\mathcal{L}^{Update}(op, w, ind)$. Meanwhile, S

simulates and sends a search token $\mathcal{ST}'_w$ or update token $\mathcal{UT}'_{\mathcal{W}_{up}}$ to $\mathcal{A}$. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.

Then we prove that $\mathcal{A}$ cannot distinguish between the experiments $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},S}(\lambda)$. In other words, $\mathcal{A}$ cannot distinguish between $\{c_j\}_{1\leq j\leq n}$, $\{ID_j\}_{1\leq j\leq n}, \{\mathcal{I}_j\}_{1\leq j\leq k}$, $\mathcal{ST}_w$, $\mathcal{UT}_w$ and $\{c'_j\}_{1\leq j\leq n}$, $\{ID'_j\}_{1\leq j\leq n}$,$\{\mathcal{I}'_j\}_{1\leq j\leq k}$, $\mathcal{ST}'_w, \mathcal{UT}'_w$. In the following proof, when we say "indistinguishable" or "cannot distinguish" we mean the advantage in distinguishing two variables is limited by $negl(\lambda)$.

Simulating $\{c'_j\}_{1\leq j\leq n}$: For $1 \leq j \leq n$, S randomly selects a bit string c'_j of length $|c_j|$. As SKE is a secure cryptographic primitive, $\{c_j\}_{1\leq j\leq n}$ and $\{c'_j\}_{1\leq j\leq n}$ are indistinguishable to $\mathcal{A}$.

Simulating $\{ID'_j\}_{1\leq j\leq n}$: For $1 \leq j \leq n$, S simply set $ID'_j = ID_j$.

Simulating $\{\mathcal{I}'_j\}_{1\leq j\leq k}$: S initializes k arrays $\{\mathcal{I}'_j\}_{1\leq j\leq k}$ of size $|\mathcal{W}|$. For each $\{\mathcal{I}'_j\}_{1\leq j\leq k}$, S randomly selects $|\mathcal{W}|$ strings $\{\alpha_j\}_{1\leq j\leq |\mathcal{W}|}$ of length λ, $|\mathcal{W}|$ bit strings $\{\gamma'_{c,j}\}_{1\leq c\leq k, 1\leq j\leq |\mathcal{W}|}$ of length N and $|\mathcal{W}|$ strings $\{\delta_j\}_{1\leq j\leq |\mathcal{W}|}$ of length λ. S sets $\mathcal{I}'_c[i] = (\alpha'_i, \gamma'_{c,i}, \delta'_i)$ for $1 \leq c \leq k, 1 \leq i \leq |\mathcal{W}|$. As H_1 is a secure hash function, $\{\mathcal{I}_j\}_{1\leq j\leq k}$ and $\{\mathcal{I}'_j\}_{1\leq j\leq k}$ are indistinguishable to $\mathcal{A}$.

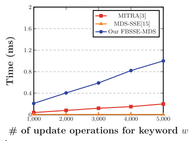
Simulating $\mathcal{UT}'_{\mathcal{W}_{up}}$: We model two hash functions H_1 and H_2 as random oracles. During an update query, for update keyword w, S gets the state (t_w, st_c) from TA, chooses string st_{c+1} randomly from $\{0,1\}^\lambda$ and then updates state (t_w, st_{c+1}). Afterwards, S gets string str_1 by sampling at random from $\{0,1\}^\lambda$ and stores $(t_w||st_{c+1}, str_1)$ in H_1. Similarly, S gets string str_2 by sampling at random from $\{0,1\}^N$, stores $(t_w||st_{c+1}, str_2)$ in H_2 and computes $\delta = str_2 \oplus st_c$. Then S chooses string γ randomly from $\{0,1\}^N$. Let i be the timestamp of the update. S stores entry $I(i) = (str_1, \gamma, \delta)$. Then S sends $\mathcal{UT}'_{\mathcal{W}_{up}} = \{(str_{1,j}, \gamma_j, \delta_j)\}_{1\leq j\leq |\mathcal{W}_{up}|}$ to $\mathcal{A}$.

Simulating $\mathcal{ST}'_w$: During a search, S receives leakage functions $\mathbf{TimeDB}(w)$ and $\mathbf{Updates}(w)$. He/she then infers from $\mathbf{Updates}(w)$ the timestamps of previous updates related to the searched keyword w, denoted by $J = (i_1, ..., i_{up})$. Afterwards, he/she infers from $\mathbf{TimeDB}(w)$ the set of file-identities that currently contain the searched keyword w and generates an N-bit string γ corresponding to the set(for each ID in the set, S sets ID-th bit to 1). Then he/she computes the XOR result s among γ and all strings that stored in $I(i_j)$ for $j = 1, ..., upd$. At last, S gets local state (t_w, st_{c+1}) and sends search token $\mathcal{ST}'_w = (t_w, st_{c+1}, s)$ to $\mathcal{A}$. In such a way, S simulates correct search/update tokens which have the same search/update results as in the experiment $\mathbf{Real}_{\mathcal{A},\mathcal{C}}(\lambda)$. Therefore, $\mathcal{A}$ cannot distinguish between $\mathcal{ST}_w$, $\mathcal{UT}_{\mathcal{W}_{up}}$ and $\mathcal{ST}'_w, \mathcal{UT}'_{\mathcal{W}_{up}}$.

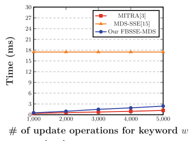
In summary, S cannot distinguish between the view in $\mathbf{Real}_{\mathcal{A},\mathcal{C}}(\lambda)$ and the view in $\mathbf{Ideal}_{\mathcal{A},S}(\lambda)$. Thus we have $|Pr[\mathbf{Real}_{\mathcal{A},\mathcal{C}}(\lambda) = 1] - Pr[\mathbf{Ideal}_{\mathcal{A},S}(\lambda) = 1]| \leq negl(\lambda)$.

6 Performance Evaluation

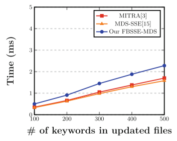
In this section we implemented our scheme, MITRA [3] and MDS-SSE [15] using Java 11. All test programs were performed on an Intel(R) Core(TM) i7-9750H 2.60 GHz computer with 8GB RAM running Windows 10. Each data point in the figures is an average of 50 executions. In all tests, we used a keyword universe of 5000 common English words and set N of FBSSE-MDS and MDS-SSE to 100000. We are interested in measuring execution time for search and update operations in MITRA, MDS-SSE and our FBEES-MDS scheme.



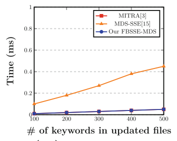
(a) search token generation



(b) search



(c) update token generation



(d) update

Fig. 5. Experiment results

Figure 5(a) and Fig. 5(b) reports the execution time for search token generation and search operation for different numbers of update operations in three schemes. In two tests, we vary the number of update operations from 1000 to 5000. As can be seen from Fig. 5(a), for MDS-SSE, the execution time of search token generation is independent of search results. The reason is that the result extraction is merely to scan a bit string and find all positions where the bit is 1. Such operations are extremely efficient. For MITRA and FBSSE-MDS, the execution time of search token generation increases when more search results are returned. This is because they need to perform a_w times hash computation while FBSSE-MDS needs to perform $(a_w + k)$ times XOR operation in addition. We can see from Fig. 5(a) that when the number of update times for w is 5000, the execution time of FBSSE-MDS is below 1.2 ms. It is efficient in the application.

In terms of execution time in search, as we can see from Fig. 5(b), for MDS-SSE, it is independent of the number of update operations for w, and related to N which is the maximum number for update operations. In this test, as we have explained before, we set N to 100000 and the execution time of MDS-SSE is 17.25 ms. As for MITRA and FBSSE-MDS, the time is linear with the number of update operations for w. The reason is that for each update operation, there is a tuple inserted to index. When 5000 results are returned, the time is about 1.5 ms in MITRA and about 2.4 ms in FBSSE-MDS. There is much difference in search time between MDS-SSE and our FBSSE-MDS.

Figure 5(c) and Fig. 5(d) presents the execution time for update token generation and update operation for different numbers of keywords contained in the updated file. In the test, we removed the file encryption time from the execution time for update token generation. Obviously, the execution time for update token generation in three schemes is linear with keywords contained in update files. For MDS-SSE, $|W_{up}|$ times XOR operation are performed in the update token generation. While for MITRA, $|W_{up}|$ times hash computation and $|W_{up}|$ times XOR operation are performed. As for our FBSSE-MDS, it needs to perform $2|W_{up}|$ times hash computation and generates $|W_{up}|$ strings of length λ in addition. As can be seen from Fig. 5(c), when the number of keywords in updated files is 500, the time is below 2.5 ms and there is little difference between three schemes.

In terms of execution time in update, as we can see from Fig. 5(d), for all three schemes, the time is linear with $|W_{up}|$ which is the number of keywords in updated files. But there is a difference between three schemes. For MITRA and FBSSE-MDS, the update operation is just inserting tuples (α, γ, δ) into the index. While for MDS-SSE, $|W_{up}|$ times connection operation for strings is performed. As can be seen from Fig. 5(d), when the number of keywords in updated files is 500, the time is about 0.05 ms in MITRA and FBSSE-MDS while the time is about 0.45 ms in MDS-SSE.

7 Conclusion

Motivated by the universal phenomenon in data outsourcing that user's data is often separately distributed, we propose a FBSSE-MDS scheme in the scenario of multiple data sources which also provides both forward privacy and BP-II backward privacy to limit the leakage to the server when data sources perform update operation. Compared to MITRA which is the fastest existing scheme achieving both forward privacy and backward privacy, our FBSSE-MDS scheme supports multiple data sources and reduces one round of interaction while maintains the high efficiency, forward privacy and backward privacy. Compared to MDS-SSE which is an efficient SSE scheme for multiple data sources, our FBSSE-MDS scheme achieves both forward and BP-II backward privacy in addition. Experimental results show that FBSSE-MDS is highly efficient and practical.

References

1. Bost, R.: $\sum o\varphi o\varsigma$: forward secure searchable encryption. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, pp. 1143–1154. ACM (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, CCS 2017, Dallas, TX, USA, pp. 1465–1482. ACM (2017)
3. Chamani, J.G., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, pp. 1038–1055. ACM (2018)
4. Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005). https://doi.org/10.1007/11496137_30

5. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, pp. 79–88. ACM (2006)
6. Etemad, M., Küpçü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. PoPETs **2018**(1), 5–20 (2018)
7. Garg, S., Mohassel, P., Papamanthou, C.: TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9816, pp. 563–592. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53015-3_20
8. Guo, Y., Zhang, C., Jia, X.: Verifiable and forward-secure encrypted search using blockchain techniques. In: 2020 IEEE International Conference on Communications, ICC 2020, Dublin, Ireland, pp. 1–7. IEEE (2020)
9. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA. The Internet Society (2012)
10. Kamara, S., Moataz, T.: Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10212, pp. 94–124. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56617-7_4
11. 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
12. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: The ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, pp. 965–976. ACM (2012)
13. Kim, K.S., Kim, M., Lee, D., Park, J.H., Kim, W.: Forward secure dynamic searchable symmetric encryption with efficient updates. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, pp. 1449–1463. ACM (2017)
14. Li, L., Xu, C., Yu, X., Dou, B., Zuo, C.: Searchable encryption with access control on keywords in multi-user setting. J. Cyber Secur. **2**(1), 9–23 (2020)
15. Liu, C., Zhu, L., Chen, J.: Efficient searchable symmetric encryption for storing multiple source dynamic social data on cloud. J. Netw. Comput. Appl. **86**, 3–14 (2017)
16. Song, D.X., Wagner, D.A., Perrig, A.: Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, 14–17 May 2000, pp. 44–55. IEEE Computer Society (2000)
17. Song, X., Dong, C., Yuan, D., Xu, Q., Zhao, M.: Forward private searchable symmetric encryption with optimized I/O efficiency. IEEE Trans. Dependable Secur. Comput. **17**(5), 912–927 (2020)
18. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA. The Internet Society (2014)
19. Sun, S., et al.: Practical backward-secure searchable encryption from symmetric puncturable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, pp. 763–780. ACM (2018)

20. Vo, V., Lai, S., Yuan, X., Sun, S., Nepal, S., Liu, J.K.: Accelerating forward and backward private searchable encryption using trusted execution. CoRR abs/2001.03743 (2020)
21. Wei, Y., Lv, S., Guo, X., Liu, Z., Huang, Y., Li, B.: FSSE: forward secure searchable encryption with keyed-block chains. Inf. Sci. **500**, 113–126 (2019)
22. Xu, L., Yuan, X., Wang, C., Wang, Q., Xu, C.: Hardening database padding for searchable encryption. In: 2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, pp. 2503–2511. IEEE (2019)
23. 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, Austin, TX, USA, pp. 707–720. USENIX Association (2016)