Dynamic Searchable Symmetric Encryption with Forward and Stronger Backward Privacy

Cong Zuo[1,2], Shi-Feng Sun[1,2,*], Joseph K. Liu[1,*], Jun Shao[3], and Josef Pieprzyk[2,4]

[1] Faculty of Information Technology, Monash University, Clayton, 3168, Australia
{cong.zuo1,shifeng.sun,joseph.liu}@monash.edu
[2] Data61, CSIRO, Melbourne/Sydney, Australia
josef.pieprzyk@data61.csiro.au
[3] School of Computer and Information Engineering, Zhejiang Gongshang University, Hangzhou 310018, Zhejiang, China
chn.junshao@gmail.com
[4] Institute of Computer Science, Polish Academy of Sciences, 01-248 Warsaw, Poland

Abstract. Dynamic Searchable Symmetric Encryption (DSSE) enables a client to perform updates and searches on encrypted data which makes it very useful in practice. To protect DSSE from the leakage of updates (leading to break query or data privacy), two new security notions, forward and backward privacy, have been proposed recently. Although extensive attention has been paid to forward privacy, this is not the case for backward privacy. Backward privacy, first formally introduced by Bost et al., is classified into three types from weak to strong, exactly Type-III to Type-I. To the best of our knowledge, however, no practical DSSE schemes without trusted hardware (e.g. SGX) have been proposed so far, in terms of the strong backward privacy and constant roundtrips between the client and the server.

In this work, we present a new DSSE scheme by leveraging simple symmetric encryption with homomorphic addition and bitmap index. The new scheme can achieve both forward and backward privacy with one roundtrip. In particular, the backward privacy we achieve in our scheme (denoted by Type-I$^-$) is somewhat stronger than Type-I. Moreover, our scheme is very practical as it involves only lightweight cryptographic operations. To make it scalable for supporting billions of files, we further extend it to a multi-block setting. Finally, we give the corresponding security proofs and experimental evaluation which demonstrate both security and practicality of our schemes, respectively.

Keywords: Dynamic Searchable Symmetric Encryption · Forward Privacy · Backward Privacy

1 Introduction

Cloud storage solutions become increasingly popular and economically attractive for users who need to handle large volumes of data. To protect the data stored

*Corresponding author.

on the cloud, users normally encrypt the data before sending it to the cloud. Unfortunately, encryption destroys the natural structure of data and consequently, data needs to be decrypted before processing. To solve this dilemma, searchable symmetric encryption (SSE) has been proposed [6, 8, 16]. SSE not only protects confidentiality of data but also permits searching over encrypted data without a need for decryption. Furthermore, SSE is much more efficient compared to other cryptographic techniques such as oblivious RAM (ORAM) that attract a punishing computational overhead [20, 11].

Early SSE solutions were designed for a static setting, i.e., an encrypted database cannot be updated. This feature of SSE severely restricts their applications. To overcome this limitation and make SSE practical, dynamic searchable symmetric encryption (DSSE) was proposed (see [5, 13]). DSSE allows both searching and updating. However, security analysis becomes more complicated as an adversary can observe the behavior of the database during the updates (addition and deletion of data). For instance, an adversary can find out if an added/deleted file contains previously searched keywords. Cash et al. [4] argued that updates can leak information about the contents of database as well as about search queries and keywords involved. For example, file-injection attacks can reveal user queries by adding to a database a small number of carefully designed files [21].

Consequently, two new security notions called forward and backward privacy were proposed to deal with the leakages mentioned above. They were informally introduced by Stefanov et al. in 2014 [17]. Roughly speaking, for any adversary who may continuously observe the interactions between the server and the client, forward privacy is satisfied if addition of new files does not leak any information about previously queried keywords. In a similar vein, backward privacy holds if files that previously added and later deleted do not leak "too much" information within any period that two search queries on the same keyword happened[1]. Bost [2] formally defined forward privacy and designed a forward-private DSSE scheme, which is resistant against file-injection attacks [21]. The scheme has been extended by Zuo et al. [23] so it supports range queries. In contrast, backward privacy attracted less attention. Recently, Bost et al. [3] defined three variants of backward privacy in order from strong to weak. They are:

- Type-I – backward privacy with insertion pattern. Given a keyword w and a time interval between two search queries on w, then Type-I leaks information about when new files containing w were inserted and the total number of updates on w.
- Type-II – backward privacy with update pattern. Apart from the leakages of Type-I, it additionally leaks when all updates (including deletion) related to w occurred.
- Type-III – weak backward privacy. It leaks information of Type-II and it also leaks exactly when a previous addition has been canceled by which deletion.

[1] The files are leaked if the second search query is issued after the files are added but before they are deleted. This is unavoidable, since the adversary can easily tell the difference of the search results before and after the same search query.

For example, assume that a query has the following form {time, operation, (keyword, file)}. Given the following queries: $\{1, search, w\}$, $\{2, add, (w, f_1)\}$, $\{3, add, (w, f_2)\}$, $\{4, add, (w, f_3)\}$, $\{5, del, (w, f_2)\}$ and $\{6, search, w\}$. Then after time 6, Type-I leaks that there are 4 updates, the files f_1 and f_3 match the keyword w and these two files were added at time 2 and 4, respectively. Type-II additionally leaks time 3 and 5 when the updates related to keyword w occurred. Type-III leaks also the fact that the addition at time 3 has been canceled by the deletion at time 5[2].

Bost et al. [3] gave several constructions with different security/efficiency trade-offs. Their `FIDES` scheme achieves Type-II backward privacy. Their schemes $\texttt{DIANA}_{del}$ and `Janus` provide better performance at the expense of security (they are Type-III backward-private). Their scheme `MONETA`, which is based on the recent TWORAM construction of Garg et al. [11], achieves Type-I backward privacy. Ghareh Chamani et al. [12], however, argued that the `MONETA` scheme is highly impractical due to the fact that it is based on TWORAM, and it serves mostly as a theoretical result for the feasibility of Type-I schemes. Sun et al. [19] proposed a new DSSE scheme named `Janus++`. It is more efficient than `Janus` as it is based on symmetric puncturable encryption. `Janus++` can only achieve the same security level as `Janus` (Type-III).

Very recently, Ghareh Chamani et al. [12] designed three DSSE schemes. The first scheme `MITRA` achieves Type-II backward privacy and it is based on symmetric key encryption getting better performance than `FIDES` [3]. The second scheme `ORION` achieves Type-I backward privacy. It requires $O(logN)$ rounds of interaction and applies ORAM [20], where N is the total number of keyword/file-identifier pairs. The third design is `HORUS`. The number of interactions is reduced to $O(d_w)$ at the expense of lower security guarantees (Type-III backward privacy), where d_w is the number of deleted entries for w. Zuo et al. [23] also constructed two DSSE schemes supporting range queries. Their first scheme achieves forward privacy. Their second scheme (called `SchemeB`) uses bit string representation and the Paillier cryptosystem which achieves backward privacy. However, they did not provide any formal analysis for the backward privacy of their scheme. To the best of our knowledge, no practical DSSE schemes achieve both the high-level backward privacy and constant interactions between the client and the server.

Our Contributions. In this paper, we propose an efficient DSSE scheme (named `FB-DSSE`) with a stronger backward privacy (denoted as Type-I^- backward privacy) and one roundtrip (without considering the retrieval of actual files), which also achieves forward privacy. This scheme is based on a bitmap index and a simple symmetric encryption with homomorphic addition. Later, we extend it to a multi-block setting (named `MB-FB-DSSE`). Table 1 compares our schemes

[2] In this example, there is only one addition/deletion pair. For Type-II, the server knows which addition has been canceled by which deletion easily. However, there may have many addition/deletion pairs, then the server cannot know which deletion cancels which addition.

(FB-DSSE and MB-FB-DSSE) with other designs supporting backward privacy. In particular, our contributions are as follows:

Table 1: Comparison with previous works

Scheme	Roundtrips bet. Client and Server	Client Storage	Forward Privacy	Backward Privacy	Without ORAM
FIDES [3]	2	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-II	✓
$\texttt{DIANA}_{del}$ [3]	2	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-III	✓
Janus [3]	1	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-III	✓
Janus++ [19]	1	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-III	✓
MITRA [12]	2	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-II	✓
HORUS [12]	$O(logd_w)$	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-III	✗
SchemeB [23]	2	$O(2\|\mathbf{W}\|log\|D\|)$	✗	Unknown	✓
MONETA [3]	3	$O(1)$	✓	Type-I	✗
ORION [12]	$O(logN)$	$O(1)$	✓	Type-I	✗
Our schemes	1	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-I$^-$	✓

N is the number of keyword/file-identifier pairs, d_w is the number of deleted entries for keyword w. $|\mathbf{W}|$ is the collection of distinct keywords, $|D|$ is the total number of files.

- We formally introduce a new type of backward privacy, named Type-I$^-$ backward privacy. It does not leak the insertion time of each matched files which is somewhat stronger than Type-I. More precisely, for a query with a keyword w, it only leaks the number of previous updates associated with w, time when these updates happened, and files that currently match w. Type-I$^-$ leaks no information about when each file was inserted. For our example, Type-I$^-$ only leaks that time 2, 3, 4, 5 are updates and f_1, f_3 currently matching keyword w[3]. Although it is not clear the impact of leaking the insertion time in practice, it is believed that the less information the scheme leaks, the higher security it guarantees, since the leakage might be leveraged by the adversary to launch some potential attacks.
- We design a Type-I$^-$ backward-private DSSE FB-DSSE by leveraging the bitmap index and the simple symmetric encryption with homomorphic addition. FB-DSSE also achieves forward privacy, which is based on the framework of [2]. In the scheme, we achieve forward privacy through a new technique which deploys symmetric primitive instead of the public primitive [2] (one-way trapdoor permutation), which makes our scheme more efficient.
- To support an even larger number of files with improved efficiency, we extend our first scheme to the multi-block setting. We call it MB-FB-DSSE. In our experimental analysis, for the MB-FB-DSSE scheme with 1 billion files, the search and update time are 5.84s and 46.41ms, respectively, where the number of blocks is 10^3 and the bit length of each block is 10^6. For the same

[3] Note that, it does not leak the insertion time of f_1 and f_3.

number of files, the FB-DSSE scheme consumes 9.07s for search and 125.23ms for update (note that, bit length is 10^9). Finally, the security analyses are given to show that our schemes are forward and Type-I$^-$ backward private.

Remark: We note that our scheme does not leak the insertion time of each matched file for a search query w, but leaks the update time for each update. Therefore, it achieves somewhat stronger security than Type-I, but strictly stronger security than Type-II. In the the conference version [24], we claim that it achieves a stronger security than Type-I, named Type-I$^-$. In this full version, we correct the inaccurate definition for Type-I$^-$ in Section 3.4 of the conference version [24] and the corresponding example given in "**Our Contributions**". In general, to eliminate the update time of each update for a search query w, it is always required to use some sort of ORAM technique [20, 12] to touch every item in a database at the expense of low efficiency. In this work, we focus on achieving stronger backward privacy without using ORAM.

1.1 Related Works

Song et al. [16] showed how to perform keyword search over encrypted data using symmetric encryption. To search for a keyword w, the server compares every encrypted keyword in a file with a token (issued by the client). The search time is linear with the number of keyword/file-identifier pairs, which is not efficient. Later, Curtmola et al. [8] designed an efficient SSE based on inverted index, which achieves sub-linear search time. The authors also quantified the leakage of an SSE and gave a formal security definition for SSE. Cash et al. [6] proposed a highly scalable SSE, which supports large databases. Following this work, many SSE schemes have been proposed addressing different aspects. For example, Sun et al. [18] focused on the usage of SSE in a multi-client setting. Zuo et al. [22] proposed an SSE scheme, which supports more general Boolean queries. To support database updates, dynamic SSE schemes are introduced in [13, 5, 2, 3].

Early schemes have been designed under the assumption that the encrypted database is static, i.e., it cannot be updated. Dynamic SSE schemes were introduced in [13, 5]. For DSSE schemes, it is assumed that an encrypted database can be updated, i.e. new files can be added and some existing files can be removed. However, a dynamic nature of databases brings new security problems. Two security notions, namely, forward and backward privacy, have been informally introduced in [17]. Further works concentrate on refinements of the privacy notions for DSSE schemes [2, 3, 12, 19].

There is also a line of investigation that concentrates on the design of SSE schemes that can handle a richer (complex) queries. Cash et al. [6] proposed an SSE scheme that handles Boolean queries. Faber et al. [9] extended the scheme so it can handle more complex queries about ranges, substrings and wild cards. A majority of forward private DSSE schemes support single keyword queries only. Zuo et al. [23] proposed a forward private DSSE scheme supporting range queries. Recently, SGX has been used to instantiate hardware-based SSE. We refer readers to [10, 1] for more details.

1.2 Organization

The remaining sections of this paper are organized as follows. In Section 2, we give the necessary background information and describe building blocks that are used in this paper. In Section 3, we define DSSE and its security notions. In Section 4, we present our DSSE schemes. Their security and experimental analyses are given in Section 5 and Section 6, respectively. Finally, Section 7 concludes this work.

2 Preliminaries

In this paper, λ denotes the security parameter, $||$ stands for the concatenation and $|m|$ denotes the bit length of m. We use bitmap index[4] to represent file identifiers [15]. More precisely, there is a bit string bs of the length ℓ, where ℓ is the maximum number of files that a scheme can support. The i-th bit of bs is set to 1 if there exists file f_i, and 0 otherwise. Fig. 1 illustrates an instance for 6 files, i.e. $\ell = 6$. Assume that there exists file f_2 and f_3 (see Fig. 1.(a)). If we want to add file f_1, we need to generate the bit string $2^1 = 000010$ and add it to the original bit string (see Fig. 1.(b)). Now, if we want to delete file f_2, we need to generate the bit string $-2^2 = -4 = -000100$. As our index computation is done modulo 2^6, we can convert $-4 = 2^6 - 2^2 = 60 \pmod{2^6}$, which is 111100 in binary. The string 111100 is added to the original bit string (see Fig. 1.(c)). Note that manipulation on bitmap indexes for addition and deletion can be done by modulo addition. In other words, the bitmap index can be (homomorphicly) encrypted and updated (to reflect addition or deletion of files) using encryption with homomorphic property.

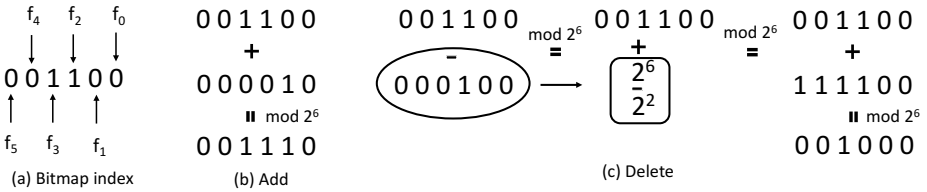


Fig. 1: An example of our bitmap index

2.1 Simple Symmetric Encryption with Homomorphic Addition

A simple symmetric encryption with homomorphic addition Π [7] consists of the following four algorithms `Setup`, `Enc`, `Dec` and `Add` as described below:

[4] A special kind of data structure which has been widely used in database community.

- $n \leftarrow \texttt{Setup}(1^\lambda)$: For the security parameter λ, it outputs a public parameter n, where $n = 2^\ell$ is the message space and ℓ is the maximum number of files a scheme can support.
- $c \leftarrow \texttt{Enc}(sk, m, n)$: For a message m, the public parameter n and a random secret key sk $(0 \leq sk < n)$, it computes a ciphertext $c = sk + m \bmod n$, where m is the message $0 \leq m < n$. Note that, for every encryption, the secret key sk needs to be stored, and it can only be used once.
- $m \leftarrow \texttt{Dec}(sk, c, n)$: For the ciphertext c, the public parameter n and the secret key sk, it recovers the message $m = c - sk \bmod n$.
- $\hat{c} \leftarrow \texttt{Add}(c_0, c_1, n)$: For two ciphertexts c_0, c_1 and the public parameter n, it computes $\hat{c} = c_0 + c_1 \bmod n$, where $c_0 \leftarrow \texttt{Enc}(sk_0, m_0, n)$, $c_1 \leftarrow \texttt{Enc}(sk_1, m_1, n)$, $n \leftarrow \texttt{Setup}(1^\lambda)$ and $0 \leq sk_0, sk_1 < n$.

Correctness. For the correctness of this scheme, it is required that sum of two ciphertexts $\hat{c} = c_0 + c_1 \bmod n$ decrypts to $m_0 + m_1 \bmod n$ under the $\hat{sk} = sk_0 + sk_1 \bmod n$ or in other words

$$\texttt{Dec}(\hat{sk}, \hat{c}, n) = \hat{c} - \hat{sk} \bmod n = m_0 + m_1 \bmod n.$$

It is easy to check that this requirement holds.

Remark. For the encryption and decryption algorithms of Π, the secret key sk can only be used once.

Perfectly Security [7]. We say Π is perfectly secure if for any PPT adversary $\mathcal{A}$, their advantage in the perfectly-security game is negligible or

$$\mathbf{Adv}^{\texttt{PS}}_{\Pi,\mathcal{A}}(\lambda) = |\Pr[\mathcal{A}(\texttt{Enc}(sk, m_0, n)) = 1] - \Pr[\mathcal{A}(\texttt{Enc}(sk, m_1, n)) = 1]| \leq \epsilon,$$

where $n \leftarrow \texttt{Setup}(1^\lambda)$, the secret key sk $(0 \leq sk < n)$ is kept secret and $\mathcal{A}$ chooses m_0, m_1 s.t. $0 \leq m_0, m_1 < n$.

2.2 Notations

Notations used in the work are given in Table 2.

3 DSSE Definition and Security Model

A database $\texttt{DB}$ is a list of file-identifier/keyword-set pairs, which is denoted as $\texttt{DB} = (f_i, \mathbf{W}_i)_{i=1}^{\ell}$. The file identifier is $f_i \in \{0,1\}^\lambda$, $\mathbf{W}_i \subseteq \{0,1\}^*$ is a set of all keywords contained in the file f_i and ℓ is the total number of files in $\texttt{DB}$. We also denote $\mathbf{W} = \cup_{i=1}^{\ell} \mathbf{W}_i$ as all keywords in $\texttt{DB}$. We identify $\mathbf{W}$ as a collection of all distinct keywords that occur in $\texttt{DB}$. Note that, $|\mathbf{W}|$ is the total number of keywords and $N = \sum_{i=1}^{\ell} |\mathbf{W}_i|$ is denoted as the total number of file-identifier/keyword pairs. A set of files that satisfy a query q is denoted by $\texttt{DB}(q)$.

Table 2: Notations (used in our schemes)

$\mathtt{DB}$	A database
λ	The security parameter
ST_c	The current search token for a keyword w
$\mathtt{EDB}$	The encrypted database $\mathtt{EDB}$ which is a map
F	A secure PRF
$\mathbf{W}$	The set of all keywords of the database $\mathtt{DB}$
$\mathbf{CT}$	A map stores the current search token ST_c and counter c for every keyword in $\mathbf{W}$
f_i	The i-th file
bs	The bit string which is used to represent the existence of files
ℓ	The length of bs
e	The encrypted bit string
Sum_e	The sum of the encrypted bit strings
sk	The one time secret key
Sum_{sk}	The sum of the one time secret keys
B	The number of blocks
$\mathbf{bs}$	The bit string array with length B
$\mathbf{e}$	The encrypted bit string array with length B
$\mathbf{Sum_e}$	The sum of the encrypted bit string arrays with length B
$\mathbf{sk}$	The one time secret key array with length B
$\mathbf{Sum_{sk}}$	The sum of the one time secret key arrays with length B

Note that, in this paper, we use bitmap index to represent the file identifiers. For a search query q, the result is a bit string bs, which represents a list of file identifiers in $\mathtt{DB}(q)$. For an update query u, a bit string bs is used to update a list of file identifiers. Moreover, we isolate the actual files from the metadata (e.g. file identifiers). We only focus on the search of the metadata. The ways we can retrieve the encrypted files are not described in this paper.

3.1 DSSE Definition

A DSSE scheme consists of an algorithm **Setup** and two protocols **Search** and **Update** that are executed between a client and a server. They are described as follows:

- $(\mathtt{EDB}, \sigma) \leftarrow \mathbf{Setup}(1^\lambda, \mathtt{DB})$: For a security parameter λ and a database $\mathtt{DB}$, the algorithm outputs a pair: an encrypted database $\mathtt{EDB}$ and a state σ. $\mathtt{EDB}$ is stored by the server and σ is kept by the client.
- $(\mathcal{I}, \perp) \leftarrow \mathbf{Search}(q, \sigma; \mathtt{EDB})$: For a state σ, the client issues a query q and interacts with the server who holds $\mathtt{EDB}$. At the end of the protocol, the client outputs a set of file identifiers $\mathcal{I}$ that match q and the server outputs nothing.
- $(\sigma', \mathtt{EDB}') \leftarrow \mathbf{Update}(\sigma, op, in; \mathtt{EDB})$: For a state σ, the operation $op \in \{add, del\}$ and a collection of $in = (f, \mathbf{w})$ pairs, the client requests the server (who holds $\mathtt{EDB}$) to update database by adding/deleting files specified by the

collection in. Finally, the protocol returns an updated state σ' to the client and an updated encrypted database $\texttt{EDB}'$ to the server.

Remark. There are two variants of result model for SSE schemes. In the first one (considered in the work [6]), the server returns encrypted file identifiers $\mathcal{I}$ so the client needs to decrypt them. In the second one (studied in the work [2]), the server returns the file identifiers to the client directly. In our work, we consider the first variant, where the protocol returns encrypted file identifiers.

3.2 Security Model

DSSE security is modeled by interaction between the Real and Ideal worlds called DSSEREAL and DSSEIDEAL, respectively. The behavior of DSSEREAL is exactly the same as the original DSSE. However, DSSEIDEAL reflects a behavior of a simulator $\mathcal{S}$, which takes the leakage of the original DSSE as input. The leakage is defined by a function $\mathcal{L} = (\mathcal{L}^{Setup}, \mathcal{L}^{Search}, \mathcal{L}^{Update})$, which details what information the adversary $\mathcal{A}$ can learn during execution of the **Setup** algorithm, **Search** and **Update** protocols.

If the adversary $\mathcal{A}$ can distinguish DSSEREAL from DSSEIDEAL with a negligible advantage, the information leakage is limited to $\mathcal{L}$ only. More formally, we consider the following security game. The adversary $\mathcal{A}$ interacts with one of the two worlds DSSEREAL or DSSEIDEAL and would like to guess it.

- $\texttt{DSSEREAL}_{\mathcal{A}}(\lambda)$: First **Setup**($\lambda$, DB) is run and the adversary gets EDB. $\mathcal{A}$ performs search queries q (or update queries (op, in)). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0, 1\}$.
- $\texttt{DSSEIDEAL}_{\mathcal{A},\mathcal{S}}(\lambda)$: Simulator $\mathcal{S}$ with the input $\mathcal{L}^{Setup}(\lambda, \texttt{DB})$ is executed. For search queries q (or update queries (op, in)) generated by the adversary $\mathcal{A}$, the simulator $\mathcal{S}$ replies by using the leakage function $\mathcal{L}^{Search}(q)$ (or $\mathcal{L}^{Update}(op, in)$). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0, 1\}$.

Definition 1. *Given a DSSE scheme and the security game described above. The scheme is $\mathcal{L}$-adaptively-secure if for every PPT adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ (with the input $\mathcal{L}$) such that,*

$$|\Pr[\texttt{DSSEREAL}_{\mathcal{A}}(\lambda) = 1] - \Pr[\texttt{DSSEIDEAL}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq negl(\lambda).$$

Leakage Function. Before define the leakage function, we define a search query $q = (t, w)$ and an update query $u = (t, op, (w, bs))$, where t is the timestamp, w is the keyword to be searched (or updated), op is the update operation and bs denotes a list of file identifiers to be updated. For a list of search queries Q, we define a search pattern $\texttt{sp}(w) = \{t : (t, w) \in Q\}$, where t is a timestamp. The search pattern leaks the repetition of search queries on w. Result pattern $\texttt{rp}(w) = \overline{bs}$, $\overline{bs}$ represents all file identifiers that currently matching w. Note that, after a search query, we implicitly assume that the server knows the final result $\overline{bs}$[5].

[5] After getting $\overline{bs}$, the client may retrieve the file identifiers represented by $\overline{bs}$ which is not described in this paper.

3.3 Forward Privacy

Informally, for any adversary who may continuously observe the interactions between the server and the client, forward privacy guarantees that an update does not leak information about the newly added files that match the previously issued queries. The definition given below is taken from [2]:

Definition 2. *A $\mathcal{L}$-adaptively-secure DSSE scheme is forward-private if the update leakage function $\mathcal{L}^{Update}$ can be written as*

$$\mathcal{L}^{Update}(op, in) = \mathcal{L}'(op, \{(f_i, \mu_i)\}),$$

where $\{(f_i, \mu_i)\}$ is the set of modified file-identifier/keywords pairs, μ_i is the number of keywords corresponding to the updated file f_i.

Remark. In this paper, the leakage function will be $\mathcal{L}^{Update}(op, w, bs) = \mathcal{L}'(op, bs)$.

3.4 Backward Privacy

Similarly, within any period that two search queries on the same keyword happened, backward privacy ensures that it does not leak information about the files that have been previously added and later deleted. Note that, information about files is leaked if the second search query is issued after the files are added but before they are deleted. In 2017, Bost et al. [3] formulated three different levels of backward privacy from Type-I to Type-III in decreasing level of privacy. In our construction, we use a new data structure (see Fig. 1), which achieves a stronger level of backward privacy. We call it Type-I$^-$, which is somewhat stronger than Type-I. We refer readers to [3] for more details. Type-I$^-$ and Type-I definitions are given below.

- Type-I$^-$: Given a time interval between two search queries for a keyword w, then it leaks the files that currently match w and the total number of updates for w and the update time for each update.
- Type-I: Given a time interval between two search queries for a keyword w, then it leaks not only files that currently match w and the total number of updates for w but additionally when the matched files were inserted.

To define Type-I$^-$ formally, we need a new leakage functions $\texttt{Time}$. For a search query on keyword w, $\texttt{Time}(w)$ lists the timestamp t of all updates corresponding to w. Formally, for a sequence of update queries Q':

$$\texttt{Time}(w) = \{t : (t, op, (w, bs)) \in Q'\}.$$

Definition 3. *A $\mathcal{L}$-adaptively-secure DSSE scheme is Type-I$^-$ backward-private iff the search and update leakage function $\mathcal{L}^{Search}, \mathcal{L}^{Update}$ can be written as:*

$$\mathcal{L}^{Update}(op, w, bs) = \mathcal{L}'(op), \mathcal{L}^{Search}(w) = \mathcal{L}''(\mathtt{sp}(w), \mathtt{rp}(w), \mathtt{Time}(w)),$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

4 Our Construction

In this section, we give our Type-I$^-$ backward private DSSE scheme. To achieve forward privacy, we follow the framework of the forward-private DSSE from [2]. To improve the efficiency of the underlying forward-private DSSE, we use a hash function that replaces a public key primitive (i.e. one-way function in [2]) to achieve forward privacy. See Section 4.2 for more details.

4.1 Overview

To achieve backward privacy, the DSSE schemes from [3, 19] used puncturable encryption, which can be used to "puncture" the deleted file identifiers. Then the deleted file identifiers cannot be decrypted (searched). The schemes achieve Type-III backward privacy only. In our construction, instead of encrypting file identifiers independently, we use a new data structure, the bitmap index (as illustrated in Fig. 1.(a)), where all file identifiers are represented by a single bit string. To add or delete a file identifier, the bit string is modified as shown in Fig. 1.(b) and Fig. 1.(c), respectively. Besides, our scheme does not leak the update type since both addition and deletion are done by one modulo addition[6]. To securely update the encrypted database, our scheme requires an additive homomorphic encryption as the underlying encryption primitive.

The most popular additive homomorphic encryption is the Paillier cryptosystem [14]. Unfortunately, the Paillier cryptosystem attracts a very large computation overhead and can support a limited number of files (up to the number of bits in a message/ciphertext space, e.g. 1024 bits). After observing our bitmap index, we notice that we only need the addition of ciphertexts (the bit strings). Also, we do not need to use one key for all encryption and decryption. Therefore we can use a simple symmetric encryption (the key can be only used once) with homomorphic addition (Section 2.1) to add the ciphertexts (and the keys) simultaneously. To save the client storage, we can use a hash function with a secret key K to generate all the one time keys. E.g. $H(K, c)$, where c is a counter. It is worth nothing that this technique has been used in [7] in the context of wireless sensor networks.

4.2 DSSE with Forward and Stronger Backward Privacy

Now we are ready to give our forward and stronger backward private DSSE construction `FB-DSSE` – see Algorithm 1. Our scheme is based on the framework of the forward private DSSE from [2], a simple symmetric encryption with homomorphic addition $\Pi = (\mathtt{Setup}, \mathtt{Enc}, \mathtt{Dec}, \mathtt{Add})$, and a keyed PRF F_K with key K. The scheme is defined by the following algorithms:

- $(\mathtt{EDB}, \sigma = (n, K, \mathbf{CT})) \leftarrow \mathbf{Setup}(1^\lambda)$: The algorithm is run by a client. It takes the security parameter λ as input. Then it chooses a secret key K

[6] Deletion is by adding a negative number.

and an integer n, where $n = 2^\ell$ and ℓ is the maximum number of files that this scheme can support. Moreover, it initializes two empty maps $\texttt{EDB}$ and $\mathbf{CT}$, which are used to store the encrypted database as well as the current search token ST_c and the current counter c (the number of updates) for each keyword $w \in \mathbf{W}$, respectively. Finally, it outputs encrypted database $\texttt{EDB}$ and the state $\sigma = (n, K, \mathbf{CT})$, and the client keeps $(K, \mathbf{CT})$ secret.

Algorithm 1 FB-DSSE

Setup(1^λ)

$K \xleftarrow{\$} \{0,1\}^\lambda$, $n \leftarrow \texttt{Setup}(1^\lambda)$
$\mathbf{CT}$, $\texttt{EDB} \leftarrow$ empty map
return $(\texttt{EDB}, \sigma = (n, K, \mathbf{CT}))$

Update$(w, bs, \sigma; \texttt{EDB})$

Client:

$K_w || K'_w \leftarrow F_K(w)$, $(ST_c, c) \leftarrow \mathbf{CT}[w]$
if $(ST_c, c) = \perp$ **then**
    $c \leftarrow -1$, $ST_c \leftarrow \{0,1\}^\lambda$
end if
$ST_{c+1} \leftarrow \{0,1\}^\lambda$
$\mathbf{CT}[w] \leftarrow (ST_{c+1}, c+1)$
$UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
$C_{ST_c} \leftarrow H_2(K_w, ST_{c+1}) \oplus ST_c$
$sk_{c+1} \leftarrow H_3(K'_w, c+1)$
$e_{c+1} \leftarrow \texttt{Enc}(sk_{c+1}, bs, n)$
Send $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$ to server.

Server:

$\texttt{EDB}[UT_{c+1}] \leftarrow (e_{c+1}, C_{ST_c})$

Search$(w, \sigma; \texttt{EDB})$

Client:

$K_w || K'_w \leftarrow F_K(w)$, $(ST_c, c) \leftarrow \mathbf{CT}[w]$
if $(ST_c, c) = \perp$ **then**
    return $\varnothing$
end if
Send (K_w, ST_c, c) to server.

Server:

$Sum_e \leftarrow 0$
for $i = c$ to 0 **do**
    $UT_i \leftarrow H_1(K_w, ST_i)$
    $(e_i, C_{ST_{i-1}}) \leftarrow \texttt{EDB}[UT_i]$
    $Sum_e \leftarrow \texttt{Add}(Sum_e, e_i, n)$
    Remove $\texttt{EDB}[UT_i]$
    if $C_{ST_{i-1}} = \perp$ **then**
        $Break$
    end if
    $ST_{i-1} \leftarrow H_2(K_w, ST_i) \oplus C_{ST_{i-1}}$
end for
$\texttt{EDB}[UT_c] \leftarrow (Sum_e, \perp)$
Send Sum_e to client.

Client:

$Sum_{sk} \leftarrow 0$
for $i = c$ to 0 **do**
    $sk_i \leftarrow H_3(K'_w, i)$
    $Sum_{sk} \leftarrow Sum_{sk} + sk_i \bmod n$
end for
$bs \leftarrow \texttt{Dec}(Sum_{sk}, Sum_e, n)$
return bs

– $(\sigma', \texttt{EDB}') \leftarrow$ **Update**$(w, bs, \sigma; \texttt{EDB})$: The algorithm runs between a client and a server. The client inputs a keyword w, a state σ and a bit string bs[7]. Next the client encrypts the bit string bs by using the simple symmetric encryption with homomorphic addition to get the encrypted bit string e. To save the client storage, the one time key sk_c is generated by a hash function $H_3(K'_w, c)$, where c is the counter. Then he/she chooses a random search token and use a hash function to get the update token. He/She also uses another hash function to mask the previous search token. After that, the

[7] Note that, we can update many file identifiers through one update query by using bit string representation bs.

client sends the update token, e and the masked previous search token C to the server and update $\mathbf{CT}$ to get a new state σ'. Finally, the server outputs an updated encrypted database $\mathtt{EDB}'$.

- $bs \leftarrow \mathbf{Search}(w, \sigma; \mathtt{EDB})$: The protocol runs between a client and a server. The client inputs a keyword w and a state σ, and the server inputs $\mathtt{EDB}$. Firstly, the client gets the search token corresponding to the keyword w from $\mathbf{CT}$ and generates the K_w. Then he/she sends them to the server. The server retrieves all the encrypted bit strings e corresponding to w. To reduce the communication overhead, the server adds them together by using the homomorphic addition ($\mathtt{Add}$) of the simple symmetric encryption to get the final result Sum_e and sends it to the client. Finally, the client decrypts it and outputs the final bit string bs which can be used to retrieve the matching files. Note that, in order to save the server storage, for every search, the server can remove all entries corresponding to w and store the final result Sum_e corresponding to the current search token ST_c to the $\mathtt{EDB}$.

4.3 Multi-block Extension for Large Number of Files

The number of files supported by FB-DSSE is determined by the length of the public parameter $n = 2^\ell$, which is the modulus. Theoretically, it can be of an arbitrary length but a larger n (e.g. $\ell = 2^{23}$) will significantly slow down modular operations. Efficiency analysis and experiments will be given in Section 6. However, there are many applications that require a system able to manage up to a billion of files. Therefore, we still need to find an efficient solution for such applications.

In this section, we extend our basic scheme to multi-block setting in order to handle a larger number of files efficiently. The idea is to split the long bit sequence ℓ into multiple smaller blocks and have multiple (e.g. B) blocks $\mathbf{bs}$ instead of one block bs. For every block of $\mathbf{bs}$, the operations are exactly the same as FB-DSSE. We denote the extension of FB-DSSE as MB-FB-DSSE, which is shown in Algorithm 2. This scheme consists of following algorithms:

- $(\mathtt{EDB}, \sigma = (n, K, \mathbf{CT})) \leftarrow \mathbf{Setup}(1^\lambda)$: The algorithm is exactly same as the one in FB-DSSE.
- $(\sigma', \mathtt{EDB}') \leftarrow \mathbf{Update}(w, \mathbf{bs}, \sigma; \mathtt{EDB})$ and $\mathbf{bs} \leftarrow \mathbf{Search}(w, \sigma; \mathtt{EDB})$: The two protocols are similar to the ones in FB-DSSE. The difference is that we use multiple blocks $\mathbf{bs}$ rather than one block bs to support large number of files, where $\mathbf{bs}$ is an array with length B and it stores all the bit strings bs. For each block, the operations are exactly same as the ones in FB-DSSE. Correspondingly, we have $\mathbf{e}$, $\mathbf{sk}$, $\mathbf{Sum_e}$ and $\mathbf{Sum_{sk}}$ which are arrays with the length of B. $\mathbf{Sum_e}$ and $\mathbf{Sum_{sk}}$ are used to store the sum of all encrypted bit string arrays $\mathbf{e}$ and secret keys $\mathbf{sk}$, respectively. The length of the bit string bs will be reduced and the computation time of these blocks will be shorter.

5 Security Analysis

In this section, we give the security analysis of our schemes.

Algorithm 2 Multi-block extension MB-FB-DSSE (Difference in red)

Setup(1^λ)

1: Same as the one in FB-DSSE.

Update$(w, \mathbf{bs}, \sigma; \mathtt{EDB})$

Client:

1: Same as the one in FB-DSSE.
2: **for** $j = 0$ to B **do**
3: $\quad \mathbf{sk}_{c+1}[j] \leftarrow H_3(K'_w, c+1||j)$
4: $\quad \mathbf{e}_{c+1}[j] \leftarrow \mathtt{Enc}(\mathbf{sk}_{c+1}[j], \mathbf{bs}[j], n)$
5: **end for**
6: Send $(ST_{c+1}, (\mathbf{e}_{c+1}, C_{ST_c}))$ to the server.

Server:

7: $\mathtt{EDB}[ST_{c+1}] \leftarrow (\mathbf{e}_{c+1}, C_{ST_c})$

Search$(w, \sigma; \mathtt{EDB})$

Client:

1: Same as the one in FB-DSSE.

Server:

2: $\mathbf{Sum_e} \leftarrow 0$
3: **for** $i = c$ to 0, $j = 0$ to B **do**
4: $\quad UT_i \leftarrow H_1(K'_w, ST_i)$
5: $\quad (\mathbf{e}_i, C_{ST_{i-1}}) \leftarrow \mathtt{EDB}[UT_i]$
6: $\quad \mathbf{Sum_e}[j] \leftarrow \mathtt{Add}(\mathbf{Sum_e}[j], \mathbf{e}_i[j], n)$
7: $\quad$ Same as the one in FB-DSSE.
8: **end for**
9: $\mathtt{EDB}[ST_c] \leftarrow (\mathbf{Sum_e}, \perp)$
10: Send $\mathbf{Sum_e}$ to the client.

Client:

11: $\mathbf{Sum_{sk}} \leftarrow 0$
12: **for** $i = c$ to 0, $j = 0$ to B **do**
13: $\quad \mathbf{sk}_i[j] \leftarrow H_3(K'_w, i||j)$
14: $\quad \mathbf{Sum_{sk}}[j] \leftarrow$
$\quad\quad \mathbf{Sum_{sk}}[j] + \mathbf{sk}_i[j] \bmod n$
15: **end for**
16: **for** $j = 0$ to B **do**
17: $\quad \mathbf{bs}[j] \leftarrow \mathtt{Dec}(\mathbf{Sum_{sk}}[j], \mathbf{Sum_e}[j], n)$
18: **end for**
19: **return** $\mathbf{bs}$

Theorem 1. *(Adaptive security of FB-DSSE). Let F be a secure PRF, $\Pi = (\mathtt{Setup}, \mathtt{Enc}, \mathtt{Dec}, \mathtt{Add})$ be a perfectly secure simple symmetric encryption with homomorphic addition, and H_1, H_2 and H_3 be random oracles. We define $\mathcal{L}_{FB\text{-}DSSE} = (\mathcal{L}^{Search}_{FB\text{-}DSSE}, \mathcal{L}^{Update}_{FB\text{-}DSSE})$, where $\mathcal{L}^{Search}_{FB\text{-}DSSE}(w) = (\mathtt{sp}(w), \mathtt{rp}(w), \mathtt{Time}(w))$ and $\mathcal{L}^{Update}_{FB\text{-}DSSE}(op, w, bs) = \perp$. Then FB-DSSE is $\mathcal{L}_{FB\text{-}DSSE}$-adaptively secure.*

Similar to [2], we will set a set of games from DSSEREAL to DSSEIDEAL, and we will show that every two consecutive games is indistinguishable. Finally, we will simulate DSSEIDEAL with the leakage functions defined in **Theorem 1**. Due to the page limitation, we move the full proof to the Appendix.

Corollary 1. *(Adaptive forward privacy of FB-DSSE). FB-DSSE is forward-private.*

From **Theorem 1**, we can infer that FB-DSSE achieves forward privacy, since the leakage function $\mathcal{L}^{Update}$ of FB-DSSE does not leak the keyword during update as defined in **Definition 2**.

Corollary 2. *(Adaptive Type-I$^-$ backward privacy of FB-DSSE). FB-DSSE is Type-I$^-$ backward-private.*

From **Theorem 1**, we can infer that FB-DSSE achieves Type-I$^-$ backward privacy, since the leakage functions of FB-DSSE leak less information than the leakage functions in **Definition 3**.

Remark. For the multi-block extension MB-FB-DSSE, the underlying construction is almost same as FB-DSSE except that it encrypts multi-block bit string **bs**

rather than one bit string bs. Hence, it inherits the forward privacy and Type-I$^-$ backward privacy of FB-DSSE.

6 Experimental Analysis

Our schemes deploy simple symmetric primitives to achieve strong backward privacy, which are more efficient than the schemes from [3, 12] because the authors of [3, 12] deploy ORAM [11, 20] to achieve strong backward privacy. The scheme Janus++ from [19] is the most efficient backward-private scheme which is based on the scheme Janus from [3]. However, Janus++ only achieves Type-III backward privacy. Table 3 compares the results.

Table 3: Comparison of computing overhead

Scheme	Search	Update
Janus [3]	$O(n_w + d_w) \cdot t_{\mathtt{PE.Dec}}$	$O(1) \cdot (t_{\mathtt{PE.Enc}}$ or $t_{\mathtt{PE.Punc}})$
Janus++ [19]	$O(n_w + d_w) \cdot t_{\mathtt{SPE.Dec}}$	$O(1) \cdot (t_{\mathtt{SPE.Enc}}$ or $t_{\mathtt{SPE.Punc}})$
MONETA [3]	$\hat{O}(a_w logN + log^3 N) \cdot t_{\mathtt{SKE}}$	$\hat{O}(log^2 N) \cdot t_{\mathtt{SKE}}$
ORION [12]	$O(n_w log^2 N) \cdot t_{\mathtt{SKE}}$	$O(log^2 N) \cdot t_{\mathtt{SKE}}$
FB-DSSE [Sec. 4.2]	$O(a_w) \cdot t_{ma}$	$O(1) \cdot t_{ma}$
MB-FB-DSSE [Sec. 4.3]	$O(a_w) \cdot t_{ma} \cdot B$	$O(1) \cdot t_{ma} \cdot B$

N is the number of keyword/file-identifier pairs, n_w is the number of files currently matching keyword w, d_w is the number of deleted entries for keyword w, and a_w is the total number of updates corresponding to keyword w. $t_{\mathtt{PE.Enc}}$, $t_{\mathtt{PE.Dec}}$ and $t_{\mathtt{PE.Punc}}$ are the encryption, decryption and puncture time of a public puncturable encryption. $t_{\mathtt{SPE.Enc}}$, $t_{\mathtt{SPE.Dec}}$ and $t_{\mathtt{SPE.Punc}}$ are the encryption, decryption and puncture time of a symmetric puncturable encryption. $t_{\mathtt{SKE}}$ is the encryption and decryption time of a symmetric key encryption. t_{ma} is the computational time of a modular addition, and B is the number of blocks. $\hat{O}$ notation hides polylogarithmic factors.

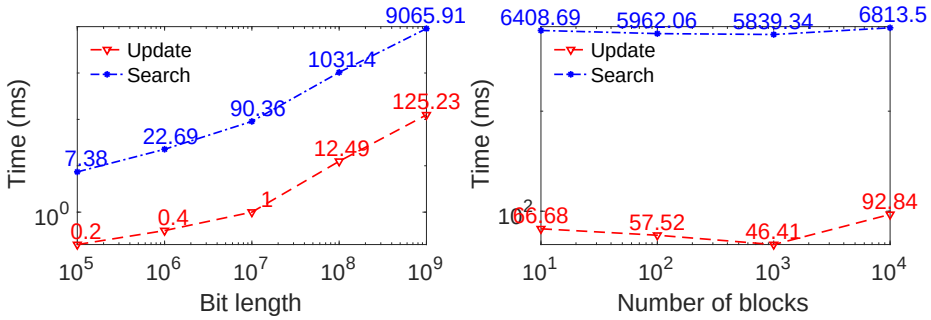
Now we are ready to give the experimental evaluation. We evaluate the performance of our schemes in a test bed of one workstation. This machine plays the roles of client and server. The hardware and software of this machine are as follows: Mac Book Pro, Intel Core i7 CPU @ 2.8GHz RAM 16GB, Java Programming Language, and macOS 10.13.2. Note that, we use the bitmap index to denote file identifiers and we tested the search and update time for one keyword. We use the "BigInteger" with different bit length to denote the bitmap index with different size which acts as the database with different number of files. The relation between the i-th bit and the actual file is out of our scope.

For every keyword, we run the update operation **Update** for this keyword 20 times. In other words, every keyword has 20 entries. The update time includes the client token generation time and server update time, and the search time includes the token generation time, the server search time and the client decryption time. Note that the result only depends on the maximum number of files supported by the system (the bit length), but not the actual number of files in the server.

First, we give the search and update time of FB-DSSE with different bit length in Fig. 2(a). The bit length refers to ℓ, which is equal to the maximum number of files supported by the system. From Fig. 2(a), we can see that the update and search time grow with the increasing of bit length. We also can observe that the update time with the bit length from 10^5 to 10^6 does not increase a lot. This is because the addition and modular have not contributed too much when the bit length is less than 10^6.

In Fig. 2(b), we evaluate the search and update time of MB-FB-DSSE with different number of blocks, where the total bit length is 10^9. When we divide one bit string (10^9) into different blocks, we can see that the running time is lesser than one block in Fig. 2(a). For the number of blocks from 10 to 10^3, it can be seen that the update and search time decrease. However, when the number of blocks is 10^4, the update and search time increase, due to the fact that when the bit string decreases to a certain length, the addition and modular time do not decrease too much.

To support an extreme large number of files (such as 1 billion), MB-FB-DSSE may be preferred than FB-DSSE. For example, the search and update time of MB-FB-DSSE are 5.84s and 46.41ms, respectively, where the number of blocks is 10^3 and the bit length of each block is 10^6. However, the search and update time of FB-DSSE supporting 1 billion files (bit length = 10^9) are 9.07s and 125.23ms, respectively.



(a) The running time of FB-DSSE with different bit length

(b) The running time of MB-FB-DSSE with different number of blocks for 10^9 bits

Fig. 2: The running time of our schemes

7 Conclusion

In this paper, we gave a DSSE scheme with stronger (named Type-I$^-$) backward privacy which also achieves forward privacy efficiently. Moreover, to make it scalable for supporting billions of files with high efficiency, we extended our

first scheme to the multi-block setting. From the experimental analysis, we can see that the efficiency of the first scheme with extreme large bit length can be improved by splitting a long bit string into multiple short bit strings. Currently, our schemes only support single keyword queries. In the future, we will make our schemes support multiple keywords queries.

Acknowledgment

The authors thank the anonymous reviewers for the valuable comments. This work was supported by the Natural Science Foundation of Zhejiang Province [grant number LZ18F020003] and the Australian Research Council (ARC) Grant DP180102199. Josef Pieprzyk has been supported by the Australian Research Council grant DP180102199 and Polish National Science Center grant 2018/31/B/ST6/03003.

References

1. Amjad, G., Kamara, S., Moataz, T.: Forward and backward private searchable encryption with sgx. In: Proceedings of the 12th European Workshop on Systems Security. p. 4. ACM (2019)
2. Bost, R.: $\Sigma o \varphi o \varsigma$: Forward secure searchable encryption. In: CCS 2016. pp. 1143–1154. ACM (2016)
3. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: CCS 2017. pp. 1465–1482. ACM (2017)
4. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: CCS 2015. pp. 668–679. ACM (2015)
5. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation. In: NDSS 2014. vol. 14, pp. 23–26. Citeseer (2014)
6. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: CRYPTO 2013, pp. 353–373. Springer (2013)
7. Castelluccia, C., Mykletun, E., Tsudik, G.: Efficient aggregation of encrypted data in wireless sensor networks. 3rd intl. In: Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Sensor Networks, Italy (2005)
8. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: CCS 2006. pp. 79–88. ACM (2006)
9. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Rich queries on encrypted data: Beyond exact matches. In: ESORICS 2015. pp. 123–145. Springer (2015)
10. Fuhry, B., Bahmani, R., Brasser, F., Hahn, F., Kerschbaum, F., Sadeghi, A.R.: Hardidx: Practical and secure index with sgx. In: IFIP Annual Conference on Data and Applications Security and Privacy. pp. 386–408. Springer (2017)
11. Garg, S., Mohassel, P., Papamanthou, C.: Tworam: Efficient oblivious ram in two rounds with applications to searchable encryption. In: Annual Cryptology Conference. pp. 563–592. Springer (2016)

12. Ghareh Chamani, J., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: CCS 2018. pp. 1038–1055. ACM (2018)
13. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: CCS 2012. pp. 965–976. ACM (2012)
14. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: EuroCrypt 1999. pp. 223–238. Springer (1999)
15. Sharma, V.: Bitmap index vs. b-tree index: Which and when? Oracle Technical Network (2005), http://www.oracle.com/technetwork/articles/sharma-indexes-093638.html
16. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: S&P 2000. pp. 44–55. IEEE (2000)
17. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS 2014. vol. 71, pp. 72–75 (2014)
18. Sun, S.F., Liu, J.K., Sakzad, A., Steinfeld, R., Yuen, T.H.: An efficient non-interactive multi-client searchable encryption with support for boolean queries. In: ESORICS 2016. pp. 154–172. Springer (2016)
19. Sun, S.F., Yuan, X., Liu, J.K., Steinfeld, R., Sakzad, A., Vo, V., Nepal, S.: Practical backward-secure searchable encryption from symmetric puncturable encryption. In: CCS 2018. pp. 763–780. ACM (2018)
20. Wang, X.S., Nayak, K., Liu, C., Chan, T., Shi, E., Stefanov, E., Huang, Y.: Oblivious data structures. In: CCS 2014. pp. 215–226. ACM (2014)
21. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: USENIX Security Symposium. pp. 707–720 (2016)
22. Zuo, C., Macindoe, J., Yang, S., Steinfeld, R., Liu, J.K.: Trusted boolean search on cloud using searchable symmetric encryption. In: Trustcom 2016. pp. 113–120. IEEE (2016)
23. Zuo, C., Sun, S.F., Liu, J.K., Shao, J., Pieprzyk, J.: Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In: ESORICS 2018. pp. 228–246. Springer (2018)
24. Zuo, C., Sun, S.F., Liu, J.K., Shao, J., Pieprzyk, J.: Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In: ESORICS 2019 (To appeal)

Appendix

Theorem 1. *(Adaptive security of FB-DSSE). Let F be a secure PRF, Π = ($\mathtt{Setup}$, $\mathtt{Enc}, \mathtt{Dec}, \mathtt{Add}$) be a perfectly secure simple symmetric encryption with homomorphic addition, and H_1, H_2 and H_3 be random oracles and output λ bits. We define $\mathcal{L}_{\mathtt{FB\text{-}DSSE}} = (\mathcal{L}^{Search}_{\mathtt{FB\text{-}DSSE}}, \mathcal{L}^{Update}_{\mathtt{FB\text{-}DSSE}})$, where $\mathcal{L}^{Search}_{\mathtt{FB\text{-}DSSE}}(w) = (\mathtt{sp}(w), \mathtt{rp}(w), \mathtt{Time}(w))$ and $\mathcal{L}^{Update}_{\mathtt{FB\text{-}DSSE}}(op, w, bs) = \perp$. Then FB-DSSE is $\mathcal{L}_{\mathtt{FB\text{-}DSSE}}$-adaptively secure.*

Proof. In this proof, the server is the adversary $\mathcal{A}$ who tries to break the security of our FB-DSSE. The challenger $\mathcal{C}$ is responsible for generating the search tokens and ciphertexts, and the simulator $\mathcal{S}$ simulates the transcripts between $\mathcal{A}$ and $\mathcal{C}$ at the end.

Game G_0: G_0 is exactly same as the real world game $\mathtt{DSSEREAL}^{\mathtt{FB\text{-}DSSE}}_{\mathcal{A}}(\lambda)$, such that

$$\Pr[\mathtt{DSSEREAL}_{\mathcal{A}}^{\mathtt{FB\text{-}DSSE}}(\lambda) = 1] = \Pr[G_0 = 1].$$

Game G_1: In G_1, when querying F to generate a key for a keyword w, the challenger $\mathcal{C}$ chooses a new random key if the keyword w is never queried before, and stores it in a table `Key`. Otherwise return the key corresponding to w in the table `Key`. If an adversary $\mathcal{A}$ is able to distinguish between G_0 and G_1, we can then build an adversary $\mathcal{B}_1$ to distinguish between F and a truly random function. More formally,

$$\Pr[G_0 = 1] - \Pr[G_1 = 1] \leq \mathbf{Adv}_{F,\mathcal{B}_1}^{\mathtt{prf}}(\lambda).$$

Game G_2: In G_2, as depicted in Algorithm 3, in the **Update** protocol, we pick random strings for the update token UT and store it in table `UT`. Then, in the **Search** protocol, we program these random strings to the output of the random oracle H_1 where $H_1(K_w, ST_c) = \mathtt{UT}[w, c]$. When $\mathcal{A}$ queries H_1 with the input (K_w, ST_c), $\mathcal{C}$ will output $\mathtt{UT}[w, c]$ to $\mathcal{A}$ and store this entry in table $\mathtt{H}_1$ for future queries. If the entry (K_w, ST_{c+1}) already in table $\mathtt{H}_1$, $\mathtt{UT}[w, c+1]$ cannot be programed to the output of $H_1(K_w, ST_{c+1})$ and this game aborts. Now, we will show that the possibility of the game aborts is negligible. The search token is chosen randomly by the challenger $\mathcal{C}$, then the possibility that the adversary guesses the right search token ST_{c+1} is $1/2^\lambda$. Assume $\mathcal{A}$ makes polynomial p queries, then the possibility is $p/2^\lambda$. So we have

$$\Pr[G_1 = 1] - \Pr[G_2 = 1] \leq p/2^\lambda$$

Game G_3: In G_3, we model the H_2 as a random oracle which is similar to H_1 in G_2. Then we have

$$\Pr[G_2 = 1] - \Pr[G_3 = 1] \leq p/2^\lambda$$

Game G_4: In G_4, similar to G_2, we model the H_3 as a random oracle. $\mathcal{A}$ does not know the key K'_w, then the possibility that he guesses the right key is $1/2^\lambda$ (we set the length of K'_w to λ). Assume $\mathcal{A}$ makes polynomial p queries, the possibility is $p/2^\lambda$. So we have

$$\Pr[G_3 = 1] - \Pr[G_4 = 1] \leq p/2^\lambda$$

Game G_5: In G_5, we replace the bit string bs with an all 0 bit string, and the length of the all 0 bit string is ℓ. If an adversary $\mathcal{A}$ is able to distinguish between G_5 and G_4, then we can build a reduction $\mathcal{B}_2$ to break the perfectly security of the simple symmetric encryption with homomorphic addition Π. So we have

$$\Pr[G_4 = 1] - \Pr[G_5 = 1] \leq \mathbf{Adv}_{\Pi,\mathcal{B}_2}^{\mathtt{PS}}(\lambda).$$

Simulator Now we can replace the searched keyword w with $\mathtt{sp}(w)$ in G_5 to simulate the simulator $\mathcal{S}$ in Algorithm 4, $\mathcal{S}$ uses the first timestamp $\hat{w} \leftarrow \mathtt{min}$

Algorithm 3 G_2

Setup(1^λ)

$K \xleftarrow{\$} \{0,1\}^\lambda$, $n \leftarrow \texttt{Setup}(1^\lambda)$
CT, EDB $\leftarrow$ empty map
return $(\texttt{EDB}, \sigma = (n, K, \mathbf{CT}))$

Update$(w, bs, \sigma; \texttt{EDB})$

Client:

$K_w || K'_w \leftarrow \texttt{Key}(w)$
$(ST_0, \cdots, ST_c, c) \leftarrow \mathbf{CT}[w]$
if $(ST_c, c) = \perp$ **then**
    $c \leftarrow -1$, $ST_c \leftarrow \{0,1\}^\lambda$
end if
$ST_{c+1} \leftarrow \{0,1\}^\lambda$
$\mathbf{CT}[w] \leftarrow (ST_0, \cdots, ST_{c+1}, c+1)$
$UT_{c+1} \leftarrow \{0,1\}^\lambda$
$\texttt{UT}[w, c+1] \leftarrow UT_{c+1}$
$C_{ST_c} \leftarrow H_2(K_w, ST_{c+1}) \oplus ST_c$
$sk_{c+1} \leftarrow H_3(K'_w, c+1)$
$e_{c+1} \leftarrow \texttt{Enc}(sk_{c+1}, bs, n)$
Send $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$ to server.

Server:

$\texttt{EDB}[UT_{c+1}] \leftarrow (e_{c+1}, C_{ST_c})$

Search$(w, \sigma; \texttt{EDB})$

Client:

$K_w || K'_w \leftarrow \texttt{Key}(w)$
$(ST_0, \cdots, ST_c, c) \leftarrow \mathbf{CT}[w]$
if $(ST_c, c) = \perp$ **then**
    return $\varnothing$
end if
for $i = 0$ to c **do**
    $\texttt{H}_1(K_w, ST_i) \leftarrow \texttt{UT}[w, i]$
end for
Send (K_w, ST_c, c) to server.

Server:

$Sum_e \leftarrow 0$
for $i = c$ to 0 **do**
    $UT_i \leftarrow H_1(K_w, ST_i)$
    $(e_i, C_{ST_{i-1}}) \leftarrow \texttt{EDB}[UT_i]$
    $Sum_e \leftarrow \texttt{Add}(Sum_e, e_i, n)$
    Remove $\texttt{EDB}[UT_i]$
    if $C_{ST_{i-1}} = \perp$ **then**
        Break
    end if
    $ST_{i-1} \leftarrow H_2(K_w, ST_i) \oplus C_{ST_{i-1}}$
end for
$\texttt{EDB}[UT_c] \leftarrow (Sum_e, \perp)$
Send Sum_e to client.

Client:

$Sum_{sk} \leftarrow 0$
for $i = c$ to 0 **do**
    $sk_i \leftarrow H_3(K'_w, i)$
    $Sum_{sk} \leftarrow Sum_{sk} + sk_i \bmod n$
end for
$bs \leftarrow \texttt{Dec}(Sum_{sk}, Sum_e, n)$
return bs

$\texttt{sp}(w)$ for the keyword w. We remove the useless part of Algorithm 3 which will not influence the view of $\mathcal{A}$.

Now we are ready to show that G_5 and **Simulator** are indistinguishable. For **Update**, it is obvious since we choose new random strings for each update in G_5. For **Search**, $\mathcal{S}$ starts from the current search token ST_c and choose a random string for previous search token. Then $\mathcal{S}$ embeds it to the ciphertext C through H_2. Moreover, $\mathcal{S}$ embeds the $\overline{bs}$ to the ST_c and all 0s to the remaining search tokens through H_3. Finally, we map the pairs (w, i) to the globe update count t. Then we can map the values in table $\texttt{UT}$, $\texttt{C}$ and $\texttt{sk}$ that we chose randomly in **Update** to the corresponding values for the pair (w, i) in the **Search**. Hence,

$$\Pr[G_5 = 1] = \Pr[\texttt{DSSEIDEAL}^{\texttt{FB-DSSE}}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]$$

Finally,

$$\Pr[\mathtt{DSSEREAL}_{\mathcal{A}}^{\mathtt{FB\text{-}DSSE}}(\lambda)=1]-\Pr[\mathtt{DSSEIDEAL}_{\mathcal{A},\mathcal{S}}^{\mathtt{FB\text{-}DSSE}}(\lambda)=1]$$

$$\leq \mathbf{Adv}_{F,\mathcal{B}_1}^{\mathtt{prf}}(\lambda)+\mathbf{Adv}_{\Pi,\mathcal{B}_2}^{\mathtt{PS}}(\lambda)+3p/2^{\lambda}$$

which completes the proof.

Algorithm 4 Simulator $\mathcal{S}$

$\mathcal{S}.\mathbf{Setup}(1^{\lambda})$

$n \leftarrow \mathtt{Setup}(1^{\lambda})$
$\mathbf{CT}$, $\mathtt{EDB} \leftarrow$ empty map
return $(\mathtt{EDB}, \mathbf{CT}, n)$

$\mathcal{S}.\mathbf{Update}()$

Client:

$\mathtt{UT}[t] \leftarrow \{0,1\}^{\lambda}$
$\mathtt{C}[t] \leftarrow \{0,1\}^{\lambda}$
$\mathtt{sk}[t] \leftarrow \{0,1\}^{\lambda}$
$\mathtt{e}[t] \leftarrow \mathtt{Enc}(\mathtt{sk}[t], 0s, n)$
Send $\mathtt{UT}[t], (\mathtt{e}[t], \mathtt{C}[t]))$ to the server.
$t \leftarrow t+1$

$\mathcal{S}.\mathbf{Search}(\mathtt{sp}(w), \mathtt{rp}(w), \mathtt{Time}(w))$

Client:

$\hat{w} \leftarrow \mathtt{min}\ \mathtt{sp}(w)$
$K_{\hat{w}}||K'_{\hat{w}} \leftarrow \mathtt{Key}(\hat{w})$
$(ST_c, c) \leftarrow \mathbf{CT}[\hat{w}]$
parse $\mathtt{rp}(w)$ as $\overline{bs}$.
Parse $\mathtt{Time}(w)$ as $(t_0, \cdots, t_c)$.
if $(ST_c, c) = \perp$ **then**
  return $\varnothing$
end if
for $i = c$ to 0 **do**
  $ST_{i-1} \leftarrow \{0,1\}^{\lambda}$
  Program $H_1(K_{\hat{w}}, ST_i) \leftarrow \mathtt{UT}[t_i]$
  Program $H_2(K_{\hat{w}}, ST_i) \leftarrow \mathtt{C}[t_i] \oplus ST_{i-1}$
  if $i = c$ **then**
    Program $H_3(K'_{\hat{w}}, i) \leftarrow \mathtt{sk}[t_i] - \overline{bs}$
  else
    Program $H_3(K'_{\hat{w}}, i) \leftarrow \mathtt{sk}[t_i]$
  end if
end for
Send $(K_{\hat{w}}, ST_c, c)$ to the server.