Forward and Backward Private DSSE for Range Queries

Cong Zuo[1,2], Shi-Feng Sun[1,2,*],Joseph K. Liu[1,*], Jun Shao[3], Josef Pieprzyk[2,4], Lei Xu[5,6]

Abstract—Due to its capabilities of searches and updates over the encrypted database, the dynamic searchable symmetric encryption (DSSE) has received considerable attention recently. To resist leakage abuse attacks, a secure DSSE scheme usually requires forward and backward privacy. However, the existing forward and backward private DSSE schemes either only support single keyword queries or require more interactions between the client and the server. In this paper, we first give a new leakage function for range queries, which is more complicated than the one for single keyword queries. Furthermore, we propose a concrete forward and backward private DSSE scheme by using a refined binary tree data structure. Finally, the detailed security analysis and extensive experiments demonstrate that our proposal is secure and efficient, respectively.

Index Terms—Dynamic Searchable Symmetric Encryption, Forward Privacy, Backward Privacy, Range Queries.

1 Introduction

Outsourcing data to the cloud is a cost-effective and reliable way to store large amounts of data. However, at the same time, it exposes data to a server that is not always trusted. Hence, the security and privacy of outsourced data should be treated well before using cloud storage. A simple method to mitigate these problems is to encrypt data before outsourcing. Unfortunately, encryption reduces the usability, especially the searchability, of the data due to the nature of encryption. To solve this problem, searchable symmetric encryption (SSE) has been introduced in [1], [2]. In this kind of encryption, as the name indicates, it can encrypt the data while keeping the searchability of the data. Compared with other techniques for enabling searchability over ciphertexts [3], [4], the clear advantages of SSE is efficiency.

Nevertheless, traditional SSE schemes cannot support updates over the encrypted database, which hinders its applications in reality. To support updates of encrypted databases, dynamic SSE (DSSE) has been proposed in [5], [6]. However, updates leak information about data to potential attacks (see [7]). Zhang et al. [8] demonstrated file-injection attacks that break privacy of client queries by injecting a small number of files to an encrypted database. To deal with the attacks, forward and backward privacy have been introduced informally in [9]. Later, they have been formalized in [10] and [11], respectively. In particular, Bost et al. [11] defined three different levels of backward privacy, namely, Type-I, Type-II and Type-III, where Type-I is the most secure and Type-III is the least secure. Note that many other forward and backward private DSSE schemes have been also proposed (see [11], [12] for instance). Nevertheless, a majority of published forward and backward private DSSE schemes support single keyword queries only. This greatly reduces their practicality. In many applications, we need more expressive search queries, such as range queries, for instance.

For range queries, a naïve solution may simply apply queries for all possible values in a range. Note that such solution is not efficient if the range is large as it requires a large communication overhead. To process range queries more efficiently and reduce communication cost, Faber et al. [13] applied a binary tree to the OXT scheme of Cash et al. [14]. Their solution works for static databases only and does support updates. Zuo et al. [15] designed two DSSE schemes using a new binary tree data structure. Their schemes support both range queries and updates. Their first scheme (`SchemeA`) based on the framework of [10] achieves forward privacy. However, it inherits low efficiency of the scheme from [10] due to application of computationally expensive public-key cryptographic operations. For the second scheme (`SchemeB`), the authors combined the bit string representation with the Paillier encryption [16]. The second scheme achieves backward privacy. The maximum number of files the scheme can support is equal to the length of the message space for the Paillier encryption. For a typical implementation, the message length is very small (around 1024 bits) and therefore a scheme can support limited number of files. To reduce storage requirements, the authors homomorphicly add the ciphertexts together and consequently their scheme looses forward privacy. In addition, they did not provide a detailed backward privacy analysis. Later, Wang et al. [17] suggested a generic forward private DSSE with range queries by adapting the `ShemeA` from [15]. To achieve backward privacy, they extended their scheme by applying the generic backward private construction of [11]. Unfortunately, to support the backward privacy,

[1]*Faculty of Information Technology, Monash University, Clayton, 3800, Australia, e-mail: {cong.zuo1,shifeng.sun,joseph.liu}@monash.edu*
[2]*Data61, CSIRO, Melbourne/Sydney, Australia*
[3]*School of Computer and Information Engineering, Zhejiang Gongshang University, Hangzhou, 310018, Zhejiang, China, e-mail: chn.junshao@gmail.com*
[4]*Institute of Computer Science, Polish Academy of Sciences, 01-248 Warsaw, Poland, e-mail: josef.pieprzyk@data61.csiro.au*
[5]*Department of Computer Science, City University of Hong Kong, Hong Kong, e-mail: xuleicrypto@gmail.com*
[6]*School of Science, Nanjing University of Science and Technology, Nanjing, 210094, Jiangsu, China*
[*]*Corresponding author.*
Manuscript received XX XX, XXXX; revised XX XX, XXXX.

their scheme requires another roundtrip between the client and the server. In other words, the client needs to the re-encrypt the matched files and send them back to the server which is not efficient.

Recently, Zuo et al. [18] designed an efficient DSSE scheme with forward and stronger backward privacy by combining bitmap index with a simple symmetric encryption with homomorphic addition. To further support very large databases, they extended their first scheme to multiple block setting. However, their schemes support single keyword queries only.

Our Contributions. In this paper, we develop an efficient forward and backward private DSSE scheme that supports range queries by extending the scheme from the work [18]. The scheme further called `FBDSSE-RQ`. It only requires one roundtrip and the comparison with previous works is given Table 1. In particular, the contributions of this work are as follows:

TABLE 1: Comparison to previous works

Scheme	Forward Privacy	Backward Privacy	Range Queries	Number of Roundtrips
`FIDES` [11]	✓	Type-II	✗	2
`DIANA`$_{del}$ [11]	✓	Type-III	✗	2
`Janus` [11]	✓	Type-III	✗	1
`Janus++` [12]	✓	Type-III	✗	1
`MONETA` [11]	✓	Type-I	✗	3
`FB-DSSE` [18]	✓	Type-I$^-$	✗	1
`SchemeA` [15]	✓	✗	✓	1
`SchemeB` [15]	✗	Unknown	✓	1
Generic [17]	✓	✗	✓	1
Extension [17]	✓	Type-II	✓	2
Our scheme	✓	Type-R	✓	1

- First, we refine the construction of the binary tree introduced in [15]. For our binary tree, we label all nodes by keywords. Names of nodes are derived from their leaf nodes rather from the order of node insertion (see Section 2.2 for more details). We also modify algorithms for the binary tree.
- We define a new backward privacy for our range queries named Type-R. Compared with singe keyword queries, range queries introduce more leakages. In this paper, we map a range query into several keywords that are assigned to nodes of our binary tree. For a range search query $[a, b]$, our range query leaks the number of keywords for the range query, the repetition of these keywords and the final results for the range query[1], and for the update with value v, it leaks the number of keywords have been updated (the number of levels of the binary tree). See Section 4 and 5 for more details.
- We give a forward and Type-R backward private DSSE for range queries. The scheme called `FBDSSE-RQ` uses our refined binary tree and is based on the `FB-DSSE` from the work [18]. In addition, it only requires one roundtrip. See Section 5 for more details.
- Finally, the security analysis and implementation experiments demonstrate that the scheme achieves claimed security goals and is practical.

1. If $a = b$, the leakage of the search query would be same as Type-I$^-$.

1.1 Related Work

Searchable symmetric encryption (SSE) was first introduced by Song et al. [1]. In their scheme, a client encrypts every keyword of a file. For a search query, the client first encrypts a keyword and then finds a match by comparing the (encrypted) keyword to (encrypted) keywords of all the files. As a result, the search time is linear with the number of file/keyword pairs. To reduce the search time, Curtmola et al. [2] deployed inverted index data structure. Consequently, their SSE scheme obtains sublinear search time. In [2], the authors also formally defined the SSE security model. There is a large number of followup papers studying different aspects of SSE. For instance, SSE with expressive queries is examined in [13]–[15], SSE for multi-client setting is explored in [2], [19], dynamic SSE – in [5], [6] and locality SSE – in [20], [21].

Once database is encrypted, SSE schemes do not allow the update of the encrypted database. To support updates of encrypted database, dynamic SSE (DSSE) schemes are introduced in [5], [6]. Early DSSE schemes are, however, vulnerable to file-injection attacks [7], [8]. To deal with the attacks, forward and backward privacy are informally introduced in [9]. Later, Bost [10] formalized the forward privacy. In 2017, Bost et al. [11] defined three levels of backward privacy (Type-I to Type-III, ordered from the most secure to the least secure). Sun et al. [12] designed a DSSE called `Janas++`, which achieves Type-III backward privacy by replacing (public-key) puncturable encryption (PE) with symmetric puncturable encryption (SPE) of `Janas`, and `Janas++` is more efficient than `Janas`.

Most forward and/or backward private DSSE schemes support single keyword queries only. Faber et al. [13] constructed a SSE scheme that accepts range queries. The scheme applies a binary tree data structure to the OXT scheme of Cash et al. [14]. However, the scheme is static (does not allow updates). To design a DSSE for range queries, Zuo et al. [15] deployed a new binary tree data structure. They described two solutions. The first one is based on the scheme by Bost [10] and it achieves forward privacy. The second solution applies the Paillier cryptosystem [16] and it is backward private. Unfortunately, the solution can support a limited number of files. This weakness is due to a limited length of the message space of the Paillier cryptosystem. Wang et al. [17] designed a generic forward private DSSE for range queries. The generic construction applies the framework of `SchemeA` from [15]. Then they extended their first scheme by integrating the generic backward private construction from [11] to achieve Type-II backward privacy. Their scheme, however, requires 2 roundtrips between the client and the server, which is not efficient. Independently, Demertzis et al. [22] developed several SSE schemes for range queries with different security and efficiency tradeoffs by using the binary tree data structure. To support update, they deploy several independent SSE instances and periodically consolidate them together. As far as information leakage is concerned, their schemes leak not only the number of keywords queried but also the level of each keyword in the binary tree.

Recently, Zuo et al. [18] introduced a forward and stronger backward private DSSE which requires one

roundtrip only. Moreover, they introduced a new notion of backward privacy (named Type-I$^-$). Compared with Type-I [11], Type-I$^-$ does not leak the insertion time of matching files. To achieve this, they deployed bitmap index and simply symmetric encryption with homomorphic addition. Experiments show that their DSSE scheme is efficient and practical. Nevertheless, it can support single keyword queries only. To the best of our knowledge, there is no forward and backward private DSSE that can process range queries with one roundtrip only.

There is also another line of investigation that explores usage of trusted hardware (SGX) in order to obtain secure DSSE (see [23], [24], for example). In this paper, we focus on constructing a secure DSSE without the trusted third party. The readers who are interested in this aspect of DSSE design are referred to [23], [24].

1.2 Organization

The remaining sections of this paper are organized as follows. In Section 2, we give the necessary background information and preliminaries. In Section 3, we define our DSSE model and forward and backward privacy notions for our range queries are given in Section 4. In Section 5, we give our forward and backward private DSSE for range queries. The security analysis is given in Section 6. Section 7 discusses implementation of our scheme and its efficiency. Finally, Section 8 concludes the work.

2 PRELIMINARIES

In this paper, λ denotes the security parameter. We use bitmap index to represent file identifiers in the same way as in the work [18]. For a database with y files, we set a bit string bs of length y. If there exists file f_i, we set the i-th bit of bs to 1. Otherwise, it is set to 0. Fig. 1 illustrates setup, addition and deletion of file identifiers. In particular, Fig. 1(a) shows a bitmap index for a database that can store up to $y = 5$ files. The index tells us that the database contains a single file f_2. Fig. 1(b) illustrates addition of file f_1 to the database, i.e. the bit string 00010 (that corresponds to f_1) is added to the index. Fig. 1(c) displays operations on the index, when the file f_2 is deleted from the database. This can be done either by subtracting the string 00100 from the index or by adding $-(00100)_2 = (11100)_2$ to the index (note that operations are performed modulo 2^5).

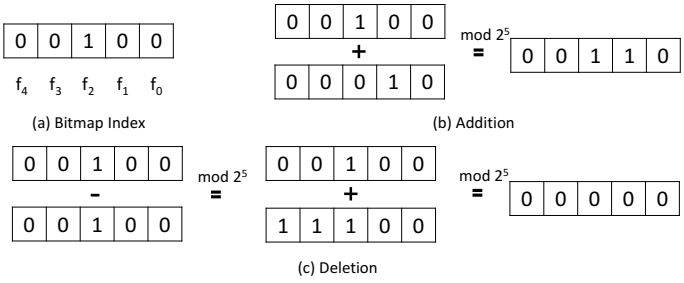


Fig. 1: Illustration of bitmap index operations

2.1 Simple Symmetric Encryption with Homomorphic Addition

Following [25], a simple symmetric encryption with homomorphic addition $\Pi = (\texttt{Setup}, \texttt{Enc}, \texttt{Dec}, \texttt{Add})$ is described by following four algorithms:

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

We claim that the above defined encryption supports homomorphic addition in the sense that knowing two ciphertexts $c_0 = m_0 + sk_0 \bmod n$ and $c_1 = m_1 + sk_1 \bmod b$, anybody can create $\hat{c} = c_0 + c_1 \bmod n$. However, to decrypt $\hat{c}$ and recover $m_0 + m_1 \bmod n$, one needs to know $sk_0 + sk_1 \bmod n$. To prove validity of the claim, it is enough to check that

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

where $\hat{sk} = sk_0 + sk_1 \bmod n$.

Note that Π enjoys perfect security as long as secret keys are used once only. To see that this is true is enough to note that our encryption becomes the well-know one-time pad (OTP) when secret key is chosen randomly and uniformly for each new message.

Perfectly Security [25]. We say Π is perfectly secure if for any adversary $\mathcal{A}$, its advantage defined as below is negligible,

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

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 Binary Tree

In this section, we revisit the binary tree BT from the work [15]. For simplicity, we always use a perfect (a.k.a. full) binary tree and denote the root $root$ as BT. A perfect binary tree is a binary tree with 2^ℓ leaf nodes, where $\ell + 1$ is the number of levels, the root exists at level 0 and leaves belong to the level ℓ. For range queries on attribute A (e.g. age) with range $R = \{0, 1, \cdots, d-1\}$, each leaf of BT is associated with a value v from R. For example, in Fig. 2(a), d is 3. To form a perfect binary tree, we need to add an additional leaf (the dot-line node in Fig. 2(a)). For Fig. 2(c), d is 5. Every node in BT has three pointers, which are initially set to null. The three pointers are $parent$, $left$ and $right$. The $parent$

Algorithm 1 Binary Tree

$\underline{\texttt{TGen}(d)}$

if $d \leq 0$ **then**
  return $\perp$
else if $d = 1$ **then**
  Generate one node n and set this node as BT.
  Associate value 0 to this node and name it as w_0.
  return BT
else
  Generate 2^ℓ leaf nodes ▷ $2^{\ell-1} < d \leq 2^\ell$
  Associate each leaf node with each value $v \in 2^\ell$ and name the corresponding leaf node as w_v.
  for $i = \ell - 1$ to 0 **do**
    Generate 2^i nodes.
    for each node **do**
      Set its left and right child to two consecutive nodes from previous level, where the value of its $leftmost$ is even and the value of its $rightmost$ is odd.
      Name this node as w_{ab}, where a and b are the values associated with its $leftmost$ and $rightmost$.
    end for
  end for
  Set the root node as BT
  return BT
end if

$\underline{\texttt{TGetCover}(q, \texttt{BT})}$ ▷ $q = [a, b]$, where $0 \leq a < b < d$

BRC, Temp, Parent $\leftarrow$ Empty Set
for $i = a$ to b **do**
  Temp $\leftarrow$ Temp $\cup\, w_i$ ▷ Put all the leaf nodes to the temp set Temp.
end for
while Temp $\neq \perp$ **do**
  for two nodes in Temp have the same parent **do**
    Remove these two nodes from Temp, and put the parent node to the set Parent.
  end for
  Move the remaining nodes from Temp to BRC.
  Temp $\leftarrow$ Parent, Parent $\leftarrow \perp$
end while
return BRC

$\underline{\texttt{TPath}(v, \texttt{BT})}$

PT $\leftarrow$ Empty Set
$w \leftarrow w_v$
while $w \neq \perp$ **do**
  PT $\leftarrow$ PT $\cup\, w$
  $w \leftarrow w \cdot parent$
end while
return PT

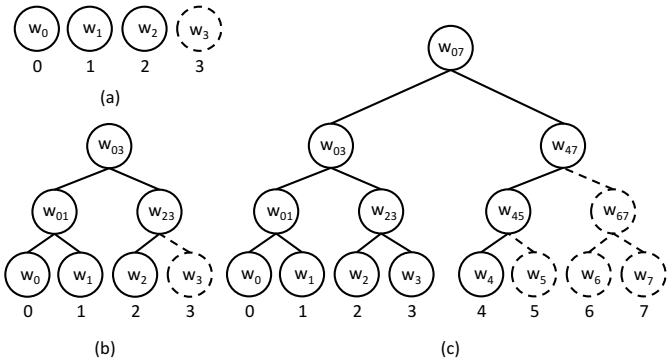


Fig. 2: Binary Tree

links the node with its parent. The pointers $left$ and $right$ connect the node with its left and right child, respectively. We also define the leftmost child $leftmost$ and rightmost child $righmost$. The $leftmost$ leaf is a node, which is the left child of its parent and all parents are left children of their ancestors. The $rightmost$ is defined similarly but for right child. For example, in Fig. 2(b), w_0 is the $leftmost$ leaf of w_{03} and w_3 is the $rightmost$ leaf of w_{03}. Now, we are ready to describe a collection of algorithms for BT (see Algorithm 1 for precise definition).

- BT $\leftarrow$ TGen(d): It takes d and outputs a perfect binary tree BT for 2^ℓ leaf nodes, where $2^{\ell-1} < d \leq 2^\ell$ and ℓ is the smallest such integer. For example, Fig. 2(b) and Fig. 2(c) illustrate a tree constructed for $d = 3$ and $d = 5$ leaves, respectively.
- BRC $\leftarrow$ TGetCover$(q, \texttt{BT})$: The algorithm takes a range $q = [a, b]$ and a binary tree BT as its input and outputs the best range cover BRC that contains all leaves in the range $[a, b]$, where $0 \leq a < b < d$[2]. Note that a BRC has to include the smallest number of parent nodes of leaves in the range. Consider the tree depicted in Fig. 2(c), BRC $= \{w_{23}, w_4\}$ for range query $q = [2, 4]$.
- PT $\leftarrow$ TPath$(v, \texttt{BT})$: The algorithm takes a value v and a binary tree BT as its input and outputs a set PT of nodes that belong to the path traversing from the leaf w_v to the root $root$, where $0 \leq v < d$. For instance, consider the tree in Fig. 2(c). For $v = 1$ (or the leaf w_1), the set PT $= \{w_1, w_{01}, w_{03}, w_{07}\}$.

2.3 Notations

Notations used in the work are given in Table 2.

3 DSSE DEFINITION AND SECURITY MODEL

For range queries, we assume that each file f is characterised by an attribute A (e.g. age), whose value v belongs to the range $R = \{0, 1, \cdots, d-1\}$. We assign the range values to the leaves of our binary tree BT as shown in Fig. 2(b). Consequently, each file contains not only the keyword of its leaf but also the keywords associated with its ancestors.

A database DB stores a list of file-identifier/keyword-set pairs or DB$= (f_i, \mathbf{W}_i)_{i=1}^{y}$, where $f_i \in \{0,1\}^\lambda$ is the file identifier, $\mathbf{W}_i$ is the keyword set and y is the total number of files in DB. For example, consider the tree from Fig. 2(b), the file f_0 is associated with the range value 0 and contains

2. If $a = b$, it becomes a single keyword query for keyword w_a.

TABLE 2: Notations

v	The value in a range query
BT	The full binary tree
$\ell+1$	The number of levels of the binary tree, where the root is in level 0 and the leaves are in level ℓ
d	The boundary of our range query
R	The set of values for our range query $\{0, 1, \cdots, d-1\}$
$[a, b]$	A range query
BRC	The set of least number of nodes to cover range $[a, b]$
PT	The set of nodes in the path from a leaf to the root
DB	A database
λ	The security parameter
ST_c	The current search token for a keyword w
EDB	The encrypted database EDB which is a map
F	A secure PRF
W	The set of all keywords of the database DB
CT	A map stores the current search token ST_c and counter c for every keyword in **W**
f_i	The i-th file
bs	The bit string which is used to represent the existence of files
y	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

keywords from the set $\mathbf{W}_0 = \{w_0, w_{01}, w_{03}\}$. We denote the collection of all distinct keywords in DB by $\mathbf{W} = \cup_{i=1}^{y} \mathbf{W}_i$. The notation **W** means the total number of keywords in the set **W** (or cardinality of the set). The total number of file-identifier/keyword pairs is denoted by $N = \sum_{i=1}^{y} |\mathbf{W}_i|$.

A set of files that satisfy a range query q is denoted by DB(q). Note that 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 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 focus on the search of the metadata only. We ignore the retrieval process of encrypted files from the database.

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:

- (EDB, σ) $\leftarrow$ **Setup**(1^λ, DB): For a security parameter λ and a database DB, the algorithm outputs a pair: an encrypted database EDB and a state σ. EDB is stored by the server and σ is kept by the client.
- ($\mathcal{I}$, $\perp$) $\leftarrow$ **Search**(q, σ; EDB): For a state σ, the client issues a query q and interacts with the server who holds 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.
- (σ', EDB$'$) $\leftarrow$ **Update**(σ, op, in; 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 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 EDB$'$ to the server.

Remark. In literature, there are two result models for SSE schemes. In the first one (considered in the work [14]), the server returns encrypted file identifiers $\mathcal{I}$ so the client needs to decrypt them. In the second one (studied in the work [10]), the server returns the file identifiers to the client as a plaintext. 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 leakages of the original DSSE as input. The leakages are defined by the 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, we can say that leakage of information is restricted to the leakage $\mathcal{L}$. More formally, we consider the following security game. The adversary $\mathcal{A}$ interacts with one of the two worlds DSSEREAL or DSSEIDEAL which are described as follows:

- $\text{DSSEREAL}_{\mathcal{A}}(\lambda)$: On input a database DB, which is chosen by the adversary $\mathcal{A}$, it outputs EDB to the $\mathcal{A}$ by running **Setup**(λ, DB). $\mathcal{A}$ performs search queries q (or update queries (op, in)). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0, 1\}$.
- $\text{DSSEIDEAL}_{\mathcal{A},\mathcal{S}}(\lambda)$: Simulator $\mathcal{S}$ outputs the simulated EDB with the input $\mathcal{L}^{Setup}(\lambda, \text{DB}))$. 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 probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ (with the input $\mathcal{L}$) such that,*

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

Leakage Function. Before defining the leakage function, we define a range query $q = (t, [a, b]) = \{t, w\}_{w \in \text{BRC}}$, where BRC is the best range cover of range $[a, b]$. An update query $u = (t, op, (v, bs)) = \{t, op, (w, bs)\}_{w \in \text{PT}(v)}$, where t is the timestamp, PT contains all the keywords in the path from the leaf node of v to the root, 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 $\text{sp}(q) = \{t : (t, w)\}_{w \in \text{BRC}}$, where t is a timestamp and $q \in Q$. The search pattern leaks the repetition of search queries on q. Denote a result pattern $\text{rp}(q) = \overline{bs}$, where $\overline{bs}$ represents all file identifiers that match the range query q. Note that, after a search query, we implicitly assume that the server knows the final result $\overline{bs}$, since the client may retrieve the file identifiers represented by $\overline{bs}$ which is not described in this paper. Moreover, the server can infer if a range query contain other range queries or not by looking at $\overline{bs}$.

4 Forward and Backward Privacy for Our Range Queries

To support range queries, we incorporate the binary tree data structure (see Section 2.2 for more details). For an update with value v, we need to update every node (keyword) in the path from the corresponding leaf node to the root node, where the value v is within the boundaries of the current binary tree. For the update with a value v, we need to issue several updates (all the keywords from the leaf to the root), hence the number of updates (the number of levels of the binary tree) is leaked.

4.1 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 [10]:

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 is the identifier of the modified file, μ_i is the number of keywords corresponding to the updated file f_i.

Remark. For our range query, the leakage function will be $\mathcal{L}^{Update}(op, v, bs) = \mathcal{L}'(op, bs, \ell + 1)$, where $\ell + 1$ is the number of levels of the full binary tree $\mathtt{BT}$.

4.2 Backward Privacy

Given a time interval, in which two search queries for the same range occur. Backward privacy ensures that there is no leak of information about the files that have been previously added and later deleted. Note that information about files leaks if the second search query is issued after the files are added but before they are deleted. In [18], Zuo et al. formulated a stronger level of backward privacy named Type-I$^-$ for single keyword queries. To deal with range queries, we map a range query to several keywords. For our range queries, to update a value, we need to update every keyword, which contains this value, hence the update leaks the number of keywords corresponding to the value, which is the number of levels of the binary tree $\ell + 1$. We call the new backward privacy Type-R.

- Type-R: Given a time interval between two calls issued for a range query q. Then it leaks the files that currently match q and the total number of updates for each w, where $w \in \mathtt{BRC}$. Update of a leaf (value v) leaks the number of keywords corresponding to the value.

To define Type-R formally, we need to introduce $\mathtt{Time}$. For a range query q, $\mathtt{Time}(q)$ lists the timestamp t of all updates corresponding to each w, where $w \in \mathtt{BRC}$. Formally, for a sequence of update queries Q':

$$\mathtt{Time}(q) = \{t : (t, op, (w, bs))\}_{w \in \mathtt{BRC}}.$$

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

$$\mathcal{L}^{Update}(op, v, bs) = \mathcal{L}'(op, \ell + 1),$$

$$\mathcal{L}^{Search}(q) = \mathcal{L}''(\mathtt{sp}(q), \mathtt{rp}(q), \mathtt{Time}(q)),$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless, $\ell + 1$ is the number of levels of the full binary tree $\mathtt{BT}$.

5 Forward and Backward Private DSSE for Range Queries

Now, we are ready to give our forward and backward private DSSE for range queries. We call it $\mathtt{FBDSSE\text{-}RQ}$ and it is defined by Algorithm 2. Our DSSE is based on the framework of [18], 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 algorithm and two protocols:

- $(\mathtt{EDB}, \sigma = (n, d, K, \mathbf{CT})) \leftarrow \mathbf{Setup}(1^\lambda)$: The algorithm is run by a client. It takes the security parameter λ as input. Then it chooses a secret key K and an integer n, where $n = 2^y$ and y is the maximum number of files that this scheme can support. Moreover, it sets the range query boundary d, two empty maps $\mathtt{EDB}$ and $\mathbf{CT}$, where $R = \{0, \cdots, d-1\}$ is set of values for our range queries and the two maps 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 $\mathtt{EDB}$ and the state $\sigma = (n, d, K, \mathbf{CT})$, and the client keeps $(d, K, \mathbf{CT})$ secret.
- $(\sigma', \mathtt{EDB}') \leftarrow \mathbf{Update}(v, bs, \sigma; \mathtt{EDB})$: The protocol runs between a client and a server. The client inputs a value v ($v \in R$), a state σ and a bit string bs[3]. The client updates each keyword $w \in \mathtt{PT}$. For each keyword w, he/she 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 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 $\mathtt{EDB}'$.
- $bs \leftarrow \mathbf{Search}(q, \sigma; \mathtt{EDB})$: The protocol runs between a client and a server. The client inputs a range query q and a state σ, and the server inputs $\mathtt{EDB}$. Firstly, the client gets $\mathtt{BRC}$. For each keyword $w \in \mathtt{BRC}$, he/she 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

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

Algorithm 2 FBDSSE−RQ

Setup(1^λ)

Client:

$K \xleftarrow{\$} \{0,1\}^\lambda$, $n \leftarrow \texttt{Setup}(1^\lambda)$
CT, EDB $\leftarrow$ empty map
Set the range boundary d.
return $(\texttt{EDB}, \sigma = (n, d, K, \textbf{CT}))$

Update$(v, bs, \sigma; \texttt{EDB})$ $\triangleright\ 0 \le v < d$

Client:

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

Server:

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

Search$(q, \sigma, \texttt{EDB})$ $\triangleright\ q = [a, b]$, where $0 \le a < b < d$.

Client:

$\texttt{BT} \leftarrow \texttt{TGen}(d)$
$\texttt{BRC} \leftarrow \texttt{TGetCover}(q, \texttt{BT})$
for $w \in \texttt{BRC}$ **do**
  $K_w || K'_w \leftarrow F_K(w)$, $(ST_c, c) \leftarrow \textbf{CT}[w]$
  if $(ST_c, c) = \perp$ **then**
    return $\perp$
  end if
end for
Send $\{(K_w, ST_c, c)\}_{w \in \texttt{BRC}}$ to the server.

Server:

$Sum \leftarrow 0$
for each (K_w, ST_c, c) **do**
  $Sum_e \leftarrow 0$
  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
  $\texttt{EDB}[UT_c] \leftarrow (Sum_e, \perp)$
  $Sum \leftarrow \texttt{Add}(Sum, Sum_e, n)$
end for
Send Sum to the client.

Client:

$Sum_{sk} \leftarrow 0$
for $w \in \texttt{BRC}$ **do**
  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
end for
$bs \leftarrow \texttt{Dec}(Sum_{sk}, Sum, n)$
return bs

all the encrypted bit strings e corresponding to w. To reduce the communication overhead, the server adds them together by using the homomorphic addition (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 EDB. Moreover, the client does not need to re-encrypt the final result bs which makes our scheme more efficient than the one in [17].

6 Security Analysis

In this section, we give the security proof of our proposed scheme.

Theorem 1. *(Adaptive forward and Type-R backward privacy of FBDSSE−RQ). Let F be a secure PRF, $\Pi = (\texttt{Setup}, \texttt{Enc}, \texttt{Dec}, \texttt{Add})$ be a perfectly secure simple symmetric encryption with homomorphic addition, and H_1, H_2 and H_3 be random oracles. We define $\mathcal{L}_{FBDSSE-RQ} = (\mathcal{L}^{Search}_{FBDSSE-RQ}, \mathcal{L}^{Update}_{FBDSSE-RQ})$, where $\mathcal{L}^{Search}_{FBDSSE-RQ}(q) = (\texttt{sp}(q), \texttt{rp}(q), \texttt{Time}(q))$ and $\mathcal{L}^{Update}_{FBDSSE-RQ}(op, v, bs) = \mathcal{L}(\ell + 1)$. Then FBDSSE−RQ is $\mathcal{L}_{FBDSSE-RQ}$-adaptively forward and Type-R backward private.*

Proof. Similar to the proof from [18], we formulate a sequence of games from DSSEREAL to DSSEIDEAL. We show that every two consecutive games are indistinguishable. Finally, we simulate DSSEIDEAL with the leakage functions defined in **Theorem 1**.

Game G_0: G_0 is exactly same as the real world game $\texttt{DSSEREAL}^{\texttt{FBDSSE−RQ}}_{\mathcal{A}}(\lambda)$. So we can write that

$$\Pr[\texttt{DSSEREAL}^{\texttt{FBDSSE−RQ}}_{\mathcal{A}}(\lambda) = 1] = \Pr[G_0 = 1].$$

Game G_1: Instead of generation of a key for a keyword w using F, we chooses the key at random and with uniform probability. The key and the corresponding keyword are stored in the table Key. If a keyword has been queried, then the corresponding key is fetched from the table Key. Assuming that an adversary $\mathcal{A}$ is able to distinguish between G_0

Algorithm 3 G_2

Setup(1^λ)
Client:
$K \xleftarrow{\$} \{0,1\}^\lambda$, $n \leftarrow \texttt{Setup}(1^\lambda)$
CT, EDB $\leftarrow$ empty map
Set the range boundary d.
return $(\texttt{EDB}, \sigma = (n, d, K, \textbf{CT}))$

Update$(v, bs, \sigma; \texttt{EDB})$ $\triangleright\ 0 \le v < d$
Client:
$\texttt{BT} \leftarrow \texttt{TGen}(d)$
$\texttt{PT} \leftarrow \texttt{TPath}(v, \texttt{BT})$
for $w \in \texttt{PT}$ **do**
  $K_w || K'_w \leftarrow \texttt{Key}(w)$
  $(ST_0, \cdots, ST_c, c) \leftarrow \textbf{CT}[w]$
  if $(ST_c, c) = \perp$ **then**
    $c \leftarrow -1, ST_c \leftarrow \{0,1\}^\lambda$
  end if
  $ST_{c+1} \leftarrow \{0,1\}^\lambda$
  $\textbf{CT}[w] \leftarrow (ST_0, \cdots, ST_{c+1}, c+1)$
  $UT_{c+1} \leftarrow \{0,1\}^\lambda$
  $\texttt{UT}[w, c+1] \leftarrow UT_{c+1}$
  $C_{ST_c} \leftarrow H_2(K_w, ST_{c+1}) \oplus ST_c$
  $sk_{c+1} \leftarrow H_3(K'_w, c+1)$
  $e_{c+1} \leftarrow \texttt{Enc}(sk_{c+1}, bs, n)$
  Send $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$ to the server.
end for
Server:
Upon receiving $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$
Set $\texttt{EDB}[UT_{c+1}] \leftarrow (e_{c+1}, C_{ST_c})$

Search$(q, \sigma, \texttt{EDB})$ $\triangleright\ q = [a, b]$, where $0 \le a < b \le d-1$.
Client:
$\texttt{BT} \leftarrow \texttt{TGen}(d)$
$\texttt{BRC} \leftarrow \texttt{TGetCover}(q, \texttt{BT})$
for $w \in \texttt{BRC}$ **do**
  $K_w || K'_w \leftarrow \texttt{Key}(w)$
  $(ST_0, \cdots, ST_c, c) \leftarrow \textbf{CT}[w]$
  if $(ST_c, c) = \perp$ **then**
    return $\perp$
  end if
  for $i = 0$ to c **do**
    $H_1(K_w, ST_i) \leftarrow \texttt{UT}[w, i]$
  end for
end for
Send $\{(K_w, ST_c, c)\}_{w \in \texttt{BRC}}$ to the server.
Server:
$Sum \leftarrow 0$
for each (K_w, ST_c, c) **do**
  $Sum_e \leftarrow 0$
  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
  $\texttt{EDB}[UT_c] \leftarrow (Sum_e, \perp)$
  $Sum \leftarrow \texttt{Add}(Sum, Sum_e, n)$
end for
Send Sum to the client.
Client:
$Sum_{sk} \leftarrow 0$
for $w \in \texttt{BRC}$ **do**
  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
end for
$bs \leftarrow \texttt{Dec}(Sum_{sk}, Sum, n)$
return bs

and G_1, then we can build an adversary $\mathcal{B}_1$ to distinguish between F and a truly random function. More formally,

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

Game G_2: The game is described in Algorithm 3. For the **Update** protocol, an update token UT is picked randomly and is stored in table UT. When the **Search** protocol is called, the random tokens are generated by the random oracle H_1 such that $H_1(K_w, ST_c) = \texttt{UT}[w, c]$. The value (K_w, ST_c) is stored in table $\texttt{H}_1$ for future queries. If an entry (K_w, ST_{c+1}) already in table $\texttt{H}_1$, then we cannot obtain the requested equality $H_1(K_w, ST_{c+1}) = \texttt{UT}[w, c+1]$ and the game aborts. Now, we show that the abortion possibility is negligible. As a search token is chosen randomly, the probability of a correct guess for search token ST_{c+1} by the adversary is $1/2^\lambda$. If $\mathcal{A}$ makes polynomial number $p(\lambda)$ of queries, then

$$\Pr[G_1 = 1] - \Pr[G_2 = 1] \le p(\lambda)/2^\lambda$$

Game G_3: We model the H_2 as a random oracle which is similar to H_1 in G_2. So we can write

$$\Pr[G_2 = 1] - \Pr[G_3 = 1] \le p(\lambda)/2^\lambda$$

Game G_4: Again, we model the H_3 as a random oracle. If the adversary does not know the key K'_w, then the probability of guessing the right key is $1/2^\lambda$ (we set the length of K'_w to λ). Assuming that $\mathcal{A}$ makes polynomial number $p(\lambda)$ of queries, the probability is $p(\lambda)/2^\lambda$. So we have

$$\Pr[G_3 = 1] - \Pr[G_4 = 1] \le p(\lambda)/2^\lambda$$

Game G_5: We replace the bit string bs by the string of all zeros (its length is y). If the 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 perfect security of the simple symmetric encryption with homomorphic addition Π. So we have

Algorithm 4 Simulator $\mathcal{S}$

$\mathcal{S}$.**Setup**(1^λ)

$n \leftarrow \mathtt{Setup}(1^\lambda)$
Set the range boundary d.
CT, $\mathtt{EDB} \leftarrow$ empty map
return $(\mathtt{EDB}, \mathbf{CT}, n, d)$

$\mathcal{S}$.**Update**$(\ell + 1)$
Client:

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

$\mathcal{S}$.**Search**$(\mathtt{sp}(q), \mathtt{rp}(q), \mathtt{Time}(q))$
Client:

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

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

Simulator Now we can replace the searched range query q with $\mathtt{sp}(q)$ in G_5 to simulate the ideal world in Algorithm 4, it uses the first timestamp $\hat{q} \leftarrow \mathtt{min}\ \mathtt{sp}(q)$ for the range query q. We ignore a part of Algorithm 3 which does not influence the view of the adversary.

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**, the simulator starts from the current search token ST_c and choose a random string for previous search token. Then it embeds it to the ciphertext C through H_2. Moreover, $\mathcal{S}$ embeds the $\overline{bs}$ to the ST_c of the last keyword in $\mathtt{BRC}$ and all 0s to the remaining search tokens through H_3. Finally, we map the pairs (w, i) to the global update count t. Then we can map the values in table $\mathtt{UT}$, $\mathtt{C}$ and $\mathtt{sk}$ that we chose randomly in **Update** to the corresponding values for the pair (w, i) in the **Search**. Hence,

$$\Pr[G_5 = 1] = \Pr[\mathtt{DSSEIDEAL}^{\mathtt{FBDSSE-RQ}}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]$$

Finally,

$$\Pr[\mathtt{DSSEREAL}^{\mathtt{FBDSSE-RQ}}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathtt{DSSEIDEAL}^{\mathtt{FBDSSE-RQ}}_{\mathcal{A},\mathcal{S}}(\lambda) = 1] \leq \mathbf{Adv}^{\mathrm{prf}}_{F,\mathcal{B}_1}(\lambda) + \mathbf{Adv}^{\mathrm{PS}}_{\Pi,\mathcal{B}_2}(\lambda) + 3p(\lambda)/2^\lambda$$

which completes the proof. □

7 Experimental Analysis

In this section, 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. 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. 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.

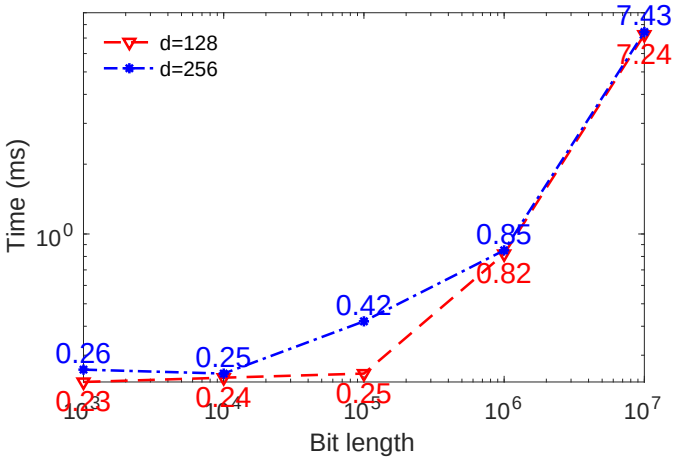


Fig. 3: The update time of $\mathtt{FBDSSE-RQ}$ with different bit length for different d

First, we show the update time of our scheme with

different bit length for different d in Fig. 3. The bit length refers to y, which is equal to the maximum number of files supported by the system. d refers the boundary of our range query. We update one time for each value. Then we get the total update time for all values and divide the number of values to get the average update time for each value. With the increase of bit length, from Fig. 3, we can see that the update time increases, except the update time with the bit length from 10^3 to 10^4 for $d = 256$. This is because the module addition has not contributed too much when the bit string is less than 10^4. We also observe that the average update time of each value for $d = 256$ is larger than the one for $d = 128$. This is due to the fact that when $d = 256$, the binary tree has more levels which means it needs more updates than the one with $d = 128$.

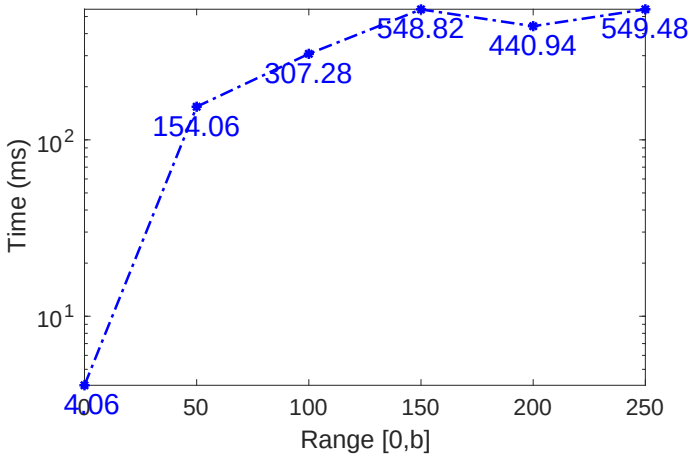


Fig. 4: The search time of FBDSSE-RQ with different range ($d = 256$, bit length is 10^7)

In Fig. 4, we evaluate the search time of our scheme with different range ($[0, 0], [0, 50], [0, 100], [0, 150], [0, 200], [0, 250]$), where $d = 256$ and bit length is 10^7. From Fig. 4, it can be seen that a lager range results in a larger search time. However, this is not always true. The search time is depends on the number of keywords in BRC of each range. The search time for range $[0, 150]$ is larger than the search time for range $[0, 200]$, because the number of keywords in BRC for range $[0, 150]$ is larger than the one for $[0, 200]$. In addition, with the increase of the bit length, the search time increases.

Theoretically, the bit string can be of an arbitrary length but a larger n (e.g. $\ell = 2^{23}$) will significantly contribute a lot of modular computation time. To support an extreme large number of files, we can divide a large bit string into several short bit strings as the multi-block setting in [18]. We refer readers to [18] for more details.

8 Conclusion

In this paper, we propose a forward and backward private DSSE for range queries (named FBDSSE-RQ) which requires only 1 roundtrip. In other words, for every search, it does not require the re-encryption of the matching files which makes our scheme more efficient. Moreover, we refine the construction of the binary tree in [15]. Names of nodes are derived from their leaf nodes rather from the order of node insertion [15]. In addition, we define a new backward privacy for range queries named Type-R. For our range query, to update a file with value v, it leaks the number of keywords that have been updated due to the binary tree data structure. From the security and experimental analyses, we can see that our proposed scheme achieves claimed security goals and is efficient, respectively. In the future, we will investigate more expressive 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], National Natural Science Foundation of China [grant number U1709217] 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

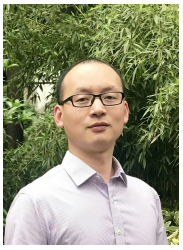
[1] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *S&P 2000*. IEEE, 2000, pp. 44–55.
[2] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *CCS 2006*. ACM, 2006, pp. 79–88.
[3] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *CCS 2014*. ACM, 2014, pp. 215–226.
[4] C. Gentry, "Fully homomorphic encryption using ideal lattices." in *Stoc*, vol. 9, no. 2009, 2009, pp. 169–178.
[5] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *CCS 2012*. ACM, 2012, pp. 965–976.
[6] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation." in *NDSS 2014*, vol. 14. Citeseer, 2014, pp. 23–26.
[7] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *CCS 2015*. ACM, 2015, pp. 668–679.
[8] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption." in *USENIX Security Symposium*, 2016, pp. 707–720.
[9] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage." in *NDSS 2014*, vol. 71, 2014, pp. 72–75.
[10] R. Bost, "$\Sigma o \varphi o \varsigma$: Forward secure searchable encryption," in *CCS 2016*. ACM, 2016, pp. 1143–1154.
[11] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *CCS 2017*. ACM, 2017, pp. 1465–1482.
[12] S.-F. Sun, X. Yuan, J. K. Liu, R. Steinfeld, A. Sakzad, V. Vo, and S. Nepal, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *CCS 2018*. ACM, 2018, pp. 763–780.
[13] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," in *ESORICS 2015*. Springer, 2015, pp. 123–145.
[14] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO 2013*. Springer, 2013, pp. 353–373.
[15] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption schemes supporting range queries with forward/backward privacy," *CoRR*, vol. abs/1905.08561, 2019. [Online]. Available: http://arxiv.org/abs/1905.08561

[16] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EuroCrypt 1999*. Springer, 1999, pp. 223–238.
[17] J. Wang and S. S. M. Chow, "Forward and backward-secure range-searchable symmetric encryption," *IACR Cryptology ePrint Archive*, vol. 2019, p. 497, 2019. [Online]. Available: https://eprint.iacr.org/2019/497
[18] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *European Symposium on Research in Computer Security*. Springer, 2019, p. To appeal.
[19] S.-F. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen, "An efficient non-interactive multi-client searchable encryption with support for boolean queries," in *ESORICS 2016*. Springer, 2016, pp. 154–172.
[20] D. Cash and S. Tessaro, "The locality of searchable symmetric encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2014, pp. 351–368.
[21] I. Miers and P. Mohassel, "Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality." in *NDSS*, 2017.
[22] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, M. Garofalakis, and C. Papamanthou, "Practical private range search in depth," *ACM Transactions on Database Systems (TODS)*, vol. 43, no. 1, p. 2, 2018.
[23] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "Hardidx: Practical and secure index with sgx," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2017, pp. 386–408.
[24] G. Amjad, S. Kamara, and T. Moataz, "Forward and backward private searchable encryption with sgx," in *Proceedings of the 12th European Workshop on Systems Security*. ACM, 2019, p. 4.
[25] C. Castelluccia, E. Mykletun, and G. Tsudik, "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.

Cong Zuo received the B.S. degree from the School of Computer Engineering, Nanjing Institute of Technology, and the M.S. degree from the School of Computer Science and Information Engineering, Zhejiang Gongshang University, China. He is currently pursuing the Ph.D. degree with Monash University under the supervision of Dr. Joseph K. Liu and Shi-Feng Sun. He is also affiliated with Data61 and his Data61 supervisor is Josef Pieprzyk. His main research interest is applied cryptography.

Shi-Feng Sun was awarded his Ph.D degree in Computer Science and Technology from Shanghai Jiao Tong University in 2016. During his Ph.D. study, he worked as a visiting scholar in the Department of Computing and Information Systems at the University of Melbourne for one year. Currently, he is a research fellow in the Faculty of Information Technology at Monash University. His research interest centers on cryptography and data privacy, particularly on provably secure cryptosystems against side-channel attacks, data privacy-preserving technology in cloud storage, and privacy-enhancing technology in blockchain.

Joseph K. Liu is an Associate Professor in the Faculty of Information Technology, Monash University. He got his PhD from the Chinese University of Hong Kong at 2004. Prior to joining Monash at 2015, he has worked as a research scientist at Institute for Infocomm Research (I2R) in Singapore for more than 7 years. His research areas include cyber security, blockchain, IoT security, applied cryptography and privacy enhanced technology. He has received more than 5700 citations and his H-index is 43, with more than 170 publications in top venues such as CRYPTO, ACM CCS. He is currently the lead of the Monash Cyber Security Group. He has established the Monash Blockchain Technology Centre at 2019 and serves as the founding director. His remarkable research in linkable ring signature forms the theory basis of Monero (XMR), one of the largest cryptocurrencies in the world. He has been given the Dean's Award for Excellence in Research Impact in 2018, and the prestigious ICT Researcher of the Year 2018 Award by the Australian Computer Society (ACS), the largest professional body in Australia representing the ICT sector, for his contribution to the blockchain and cyber security community.



Jun Shao received the Ph.D. degree from the Department of Computer Science and Engineering at Shanghai Jiao Tong University, Shanghai, China in 2008. He was a postdoc in the School of Information Sciences and Technology at Pennsylvania State University, USA from 2008 to 2010. He is currently a professor of the School of Computer Science and Information Engineering at Zhejiang Gongshang University, Hangzhou, China. His research interests include network security and applied cryptography.

Josef Pieprzyk is a Senior Principal Research Scientist at CSIRO Data61. His main research interest focus is Cryptology and Information Security and includes design and analysis of cryptographic algorithms (such as encryption, hashing and digital signatures), secure multiparty computations, cryptographic protocols, copyright protection, e-commerce, web security and cybercrime prevention. He is a member of the editorial boards for International Journal of Information Security (Springer), Journal of Mathematical Cryptology (De Gruyter), Open Access Journal of Cryptography (MDPI), International Journal of Applied Cryptography (Inderscience Publishers), Fundamenta Informaticae (IOS Press), International Journal of Security and Networks (Inderscience Publishers) and International Journal of Information and Computer Security (Inderscience Publishers). He published 5 books, edited 10 books (conference proceedings), 6 book chapters, and more than 300 papers in refereed journals and refereed international conferences.

Lei Xu is working on his Ph.D. degree in School of Science, Nanjing University of Science and Technology, Nanjing, China. received his bachelor degree from Anhui Normal University, China, in 2012. During the period from April 2017 to April 2018, he is also a visiting Ph.D. student at Faculty of Information Technology, Monash University. His main research interests focus on public key cryptography and information security, including searchable encryption mechanism.