Dynamic Searchable Symmetric Encryption Schemes Supporting Range Queries with Forward (and Backward) Security

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) is a useful cryptographic tool in encrypted cloud storage. However, it has been reported that DSSE usually suffers from file-injection attacks and content leak of deleted documents. To mitigate these attacks, forward security and backward security have been proposed. Nevertheless, the existing forward/backward-secure DSSE schemes can only support single keyword queries. To address this problem, in this paper, we propose two DSSE schemes supporting range queries. One is forward-secure and supports a large number of documents. The other can achieve both forward security and backward security, while it can only support a limited number of documents. Finally, we also give the security proofs of the proposed DSSE schemes in the random oracle model.

Keywords: Dynamic searchable symmetric encryption
Forward security · Backward security · Range queries

1 Introduction

Searchable symmetric encryption (SSE) is a useful cryptographic primitive that can encrypt the data to protect its confidentiality while keeping its searchability. Dynamic SSE (DSSE) further provides data dynamics that allows the client to update data over the time without losing data confidentiality and searchability. Due to this property, DSSE is highly demanded in encrypted cloud. However, many existing DSSE schemes [8,14] suffer from file-injection attacks [7,22], where

J. Lopez et al. (Eds.): ESORICS 2018, LNCS 11099, pp. 228–246, 2018.
https://doi.org/10.1007/978-3-319-98989-1_12

the adversary can compromise the privacy of a client query by injecting a small portion of new documents to the encrypted database. To resist this attack, Zhang et al. [22] highlighted the need of forward security that was informally introduced by Stefanov et al. [19]. The formal definition of forward security for DSSE was given by Bost [5] who also proposed a concrete forward-secure DSSE scheme. Furthermore, Bost et al. [6] demonstrated the damage of content leak of deleted documents and proposed the corresponding security notion–backward security. Several backward-secure DSSE schemes were also presented in [6].

Nevertheless, the existing forward/backward-secure DSSE schemes only support single keyword queries, which are not expressive enough in data search service [12,13]. To solve this problem, in this paper, we aim to design forward/backward-secure DSSE schemes supporting range queries. Our design starts from the regular binary tree in [13] to support range queries. However, the binary tree in [13] cannot be applied directly to the dynamic setting. It is mainly because that the keywords in [13] are labelled according to the corresponding tree levels that will change significantly in the dynamic setting. A naïve solution is to replace all old keywords by the associated new keywords. This is, however, not efficient. To address this problem, we have to explore new approaches for our goal.

Our Contributions. To achieve above goal, we propose two new DSSE constructions supporting range queries in this paper. The first one is forward-secure but with a larger client overhead in contrast to [13]. The second one is a more efficient DSSE which achieves both forward and backward security at the same time. In more details, our main contributions are as follows:

- To make the binary tree suitable for range queries in the dynamic setting, we introduce a new binary tree data structure, and then present the first forward-secure DSSE supporting range queries by applying it to Bost's scheme [5]. However, the forward security is achieved at the expense of suffering from a large storage overhead on the client side.
- To reduce the large storage overhead, we further propose another DSSE scheme supporting range queries by leveraging the Paillier cryptosystem [17]. With its homomorphic property, this construction can achieve not only forward security, but also backward security. Notably, due to the limitation of the Paillier cryptosystem, it cannot support large-scale database consisting a large number of documents. Nevertheless, it suits well for certain scenarios where the number of documents is moderate. The new approach may give new lights on designing more efficient and secure DSSE schemes.
- Also, the comparison with related works in Table 1 and detailed security analyses are provided, which demonstrate that our constructions are not only forward (and backward)-secure but also with a comparable efficiency.

Table 1. Comparison with existing DSSE schemes

Scheme	Client computation		Client storage	Range queries	Forward security	Backward security	Document number
	Search	Update					
[13]	w_R	-	$O(1)$	✓	✗	✗	Large
[5]	-	$O(1)$	$O(W)$	✗	✓	✗	Large
Ours A	w_R	$\lceil log(W) \rceil + 1$	$O(2W)$	✓	✓	✗	Large
Ours B	w_R	$\lceil log(W) \rceil + 1$	$O(1)$	✓	✓	✓	Small

W is the number of keywords in a database, w_R is the number of keywords for a range query (we map a range query to a few different keywords).

1.1 Related Works

Song et al. [18] were the first using symmetric encryption to facilitate keyword search over the encrypted data. Later, Curtmola et al. [11] gave a formal definition for SSE and the corresponding security model in the static setting. To make SSE more scalable and expressive, Cash et al. [9] proposed a new scalable SSE supporting Boolean queries. Following this construction, many extensions have been proposed. Faber et al. [13] extended it to process a much richer collection of queries. For instance, they used a binary tree with keywords labelled according to the tree levels to support range queries. Zuo et al. [23] made another extension to support general Boolean queries. Cash et al.'s construction has also been extended into multi-user setting [15,20,21]. However, the above schemes cannot support data update. To solve this problem, some DSSE schemes have been proposed [8,14].

However, designing a secure DSSE scheme is not an easy job. Cash et al. [7] pointed out that only a small leakage leveraged by the adversary would be enough to compromise the privacy of clients' queries. A concrete attack named file-injection attack was proposed by Zhang et al. [22]. In this attack, the adversary can infer the concept of a client queries by injecting a small portion of new documents into encrypted database. This attack also highlights the need for forward security which protects security of new added parts. Accordingly, we have backward security that protects security of new added parts and later deleted. These two security notions were first introduced by Stefanov et al. [19]. The formal definitions of forward/backward security for DSSE were given by Bost [5] and Bost et al. [6], respectively. In [5], Bost also proposed a concrete forward-secure DSSE scheme, it does not support physical deletion. Later on, Kim et al. [16] proposed a forward-secure DSSE scheme supporting physical deletion. Meanwhile, Bost et al. [6] proposed a forward/backward-secure DSSE to reduce leakage during deletion. Unfortunately, all the existing forward/backward-secure DSSE schemes only support single keyword queries. Hence, forward/backward-secure DSSE supporting more expressive queries, such as range queries, are quite desired.

Apart from the binary tree technique, order preserving encryption (OPE) can also be used to support range queries. The concept of OPE was proposed by

Agrawal et al. [1], and it allows the order of the plaintexts to be preserved in the ciphertexts. It is easy to see that this kind of encryption would lead to the leakage in [2,3]. To reduce this leakage, Boneh et al. [4] proposed another concept named order revealing encryption (ORE), where the order of the ciphertexts are revealed by using an algorithm rather than comparing the ciphertexts (in OPE) directly. More efficient ORE schemes were proposed later [10]. However, ORE-based SSE still leaks much information about the underlying plaintexts. To avoid this, in this paper, we focus on how to use the binary tree structure to achieve range queries.

1.2 Organization

The remaining sections of this paper are organized as follows. In Sect. 2, we give the background information and building blocks that are used in this paper. In Sect. 3, we give the definition of DSSE and its security definition. After that in Sect. 4, we present a new binary tree and our DSSE schemes. Their security analyses are given in Sect. 5. Finally, Sect. 6 concludes this work.

2 Preliminaries

In this section, we describe cryptographic primitives (building blocks) that are used in this work.

2.1 Trapdoor Permutations

A trapdoor permutation (TDP) Π is a one-way permutation over a domain D such that (1) it is "easy" to compute Π for any value of the domain with the public key, and (2) it is"easy" to calculate the inverse Π^{-1} for any value of a co-domain $\mathcal{M}$ only if a matching secret key is known. More formally, Π consists of the following algorithms:

- $\mathtt{TKeyGen}(1^\lambda) \rightarrow (\mathtt{TPK}, \mathtt{TSK})$: For a security parameter 1^λ, the algorithm returns a pair of cryptographic keys: a public key TPK and a secret key TSK.
- $\Pi(\mathtt{TPK}, x) \rightarrow y$: For a pair: public key TPK and $x \in D$, the algorithm outputs $y \in \mathcal{M}$.
- $\Pi^{-1}(\mathtt{TSK}, y) \rightarrow x$: For a pair: a secret key TSK and $y \in \mathcal{M}$, the algorithm returns $x \in D$.

One-wayness. We say Π is one-way if for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, an advantage

$$\mathtt{Adv}^{\mathtt{OW}}_{\Pi,\mathcal{A}}(1^\lambda) = \Pr[x \leftarrow \mathcal{A}(\mathtt{TPK}, y)]$$

is negligible, where $(\mathtt{TSK}, \mathtt{TPK}) \leftarrow \mathtt{TKeyGen}(1^\lambda)$, $y \leftarrow \Pi(\mathtt{TPK}, x)$, $x \in D$.

2.2 Paillier Cryptosystem

A Paillier cryptosystem $\Sigma = (\mathtt{KeyGen}, \mathtt{Enc}, \mathtt{Dec})$ is defined by following three algorithms:

- $\mathtt{KeyGen}(1^\lambda) \rightarrow (\mathtt{PK}, \mathtt{SK})$: It chooses at random two primes p and q of similar lengths and computes $n = pq$ and $\phi(n) = (p-1)(q-1)$. Next it sets $g = n+1$, $\beta = \phi(n)$ and $\mu = \phi(n)^{-1} \bmod n$. It returns $\mathtt{PK} = (n, g)$ and $\mathtt{SK} = (\beta, \mu)$.
- $\mathtt{Enc}(\mathtt{PK}, m) \rightarrow c$: Let m be the message, where $0 \leq m < n$, the algorithm selects an integer r at random from $\mathbb{Z}_n$ and computes a ciphertext $c = g^m \cdot r^n \bmod n^2$.
- $\mathtt{Dec}(\mathtt{SK}, c) \rightarrow m$: The algorithm calculates $m = L(c^\beta \bmod n^2) \cdot \mu \bmod n$, where $L(x) = \frac{x-1}{n}$.

Semantically Security. We say Σ is semantically secure if for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, an advantage

$$\mathtt{Adv}^{\mathtt{IND\text{-}CPA}}_{\Sigma,\mathcal{A}}(1^\lambda) = |\Pr[\mathcal{A}(\mathtt{Enc}(\mathtt{PK}, m_0)) = 1] - \Pr[\mathcal{A}(\mathtt{Enc}(\mathtt{PK}, m_1)) = 1]|$$

is negligible, where $(\mathtt{SK}, \mathtt{PK}) \leftarrow \mathtt{KeyGen}(1^\lambda)$, $\mathcal{A}$ chooses m_0, m_1 and $|m_0| = |m_1|$.

Homomorphic Addition. Paillier cryptosystem is homomorphic, i.e.

$$\mathtt{Dec}(\mathtt{Enc}(m_1) \cdot \mathtt{Enc}(m_2)) \bmod n^2 = m_1 + m_2 \bmod n.$$

We need this property to achieve forward security of our DSSE.

2.3 Notations

The list of notations used is given in Table 2.

3 Dynamic Searchable Symmetric Encryption (DSSE)

We follow the database model given in the paper [5]. A database is a collection of (index, keyword set) pairs denoted as $\mathtt{DB} = (ind_i, \mathbf{W}_i)_{i=1}^d$, where $ind_i \in \{0,1\}^\ell$ and $\mathbf{W}_i \subseteq \{0,1\}^*$. The set of all keywords of the database $\mathtt{DB}$ is $\mathbf{W} = \cup_{i=1}^d \mathbf{W}_i$, where d is the number of documents in $\mathtt{DB}$. We identify $W = |\mathbf{W}|$ as the total number of keywords and $N = \Sigma_{i=1}^d |\mathbf{W}_i|$ as the number of document/keyword pairs. We denote $\mathtt{DB}(w)$ as the set of documents that contain a keyword w. To achieve a sublinear search time, we encrypt the file indices of $\mathtt{DB}(w)$ corresponding to the same keyword w (a.k.a. inverted index[1]).

A DSSE scheme Γ consists of an algorithm **Setup** and two protocols **Search** and **Update** as described below.

[1] It is an index data structure where a word is mapped to a set of documents which contain this word.

Table 2. Notations (used in our constructions)

W	The number of keywords in a database DB
BDB	The binary database which is constructed from a database DB by using our binary tree BT
m	The number of values in the range $[0, m-1]$ for our range queries
v	A value in the range $[0, m-1]$ where $0 \leq v < m$
n_i	The i-th node in our binary tree which is considered as the keyword
$\texttt{root}_o$	The root node of the binary tree before update
$\texttt{root}_n$	The root node of the binary tree after update
ST_c	The current search token for a node n
$\mathcal{M}$	A random value for ST_0 which is the first search token for a node n
UT_c	The current update token for a node n
T	A map which is used to store the encrypted database EDB
N	A map which is used to store the current search token for n_i
NSet	The node set which contains the nodes
TPK	The public key of trapdoor permutation
TSK	The secret key of trapdoor permutation
PK	The public key of Paillier cryptosystem
SK	The secret key of Paillier cryptosystem
f_i	The i-th file
PBT	Perfect binary tree
CBT	Complete binary tree
VBT	Virtual perfect binary tree
ABT	Assigned complete binary tree

- $(\texttt{EDB}, \sigma) \leftarrow$ **Setup**$(\texttt{DB}, 1^\lambda)$: For a security parameter 1^λ and a database DB. The algorithm outputs an encrypted database EDB for the server and a secret state σ for the client.
- $(\mathcal{I}, \perp) \leftarrow$ **Search**$(q, \sigma, \texttt{EDB})$: The protocol is executed between a client (with her query q and state σ) and a server (with its EDB). At the end of the protocol, the client outputs a set of file indices $\mathcal{I}$ and the server outputs nothing.
- $(\sigma', \texttt{EDB}') \leftarrow$ **Update**$(\sigma, op, in, \texttt{EDB})$: The protocol runs between a client and a server. The client input is a state σ, an operation $op = (add, del)$ she wants to perform and a collection of $in = (ind, \mathbf{w})$ pairs that are going to be modified, where add, del mean the addition and deletion of a document/keyword pair, respectively, and ind is the file index and $\mathbf{w}$ is a set of keywords. The server input is EDB. **Update** returns an updated state σ' to the client and an updated encrypted database $\texttt{EDB}'$ to the server.

3.1 Security Definition

The security definition of DSSE is formulated using the following two games: $\mathtt{DSSEREAL}^{\Gamma}_{\mathcal{A}}(1^{\lambda})$ and $\mathtt{DSSEIDEAL}^{\Gamma}_{\mathcal{A},\mathcal{S}}(1^{\lambda})$. The $\mathtt{DSSEREAL}^{\Gamma}_{\mathcal{A}}(1^{\lambda})$ is executed using DSSE. The $\mathtt{DSSEIDEAL}^{\Gamma}_{\mathcal{A},\mathcal{S}}(1^{\lambda})$ is simulated using the leakage of DSSE. The leakage is parameterized by a function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$, which describes what information is leaked to the adversary $\mathcal{A}$. If the adversary $\mathcal{A}$ cannot distinguish these two games, then we can say there is no other information leaked except the information that can be inferred from the leakage function $\mathcal{L}$. More formally,

- $\mathtt{DSSEREAL}^{\Gamma}_{\mathcal{A}}(1^{\lambda})$: On input a database $\mathtt{DB}$, which is chosen by the adversary $\mathcal{A}$, it outputs $\mathtt{EDB}$ by using $\mathbf{Setup}(1^{\lambda}, \mathtt{DB})$ to the adversary $\mathcal{A}$. $\mathcal{A}$ can repeatedly perform a search query q (or an update query (op, in)). The game outputs the results generated by running $\mathbf{Search}(q)$ (or $\mathbf{Update}(op, in)$) to the adversary $\mathcal{A}$. Eventually, $\mathcal{A}$ outputs a bit.
- $\mathtt{DSSEIDEAL}^{\Gamma}_{\mathcal{A},\mathcal{S}}(1^{\lambda})$: On input a database $\mathtt{DB}$ which is chosen by the adversary $\mathcal{A}$, it outputs $\mathtt{EDB}$ to the adversary $\mathcal{A}$ by using a simulator $\mathcal{S}(\mathcal{L}^{Stp}(1^{\lambda}, \mathtt{DB}))$. Then, it simulates the results for the search query q by using the leakage function $\mathcal{S}(\mathcal{L}^{Srch}(q))$ and uses $\mathcal{S}(\mathcal{L}^{Updt}(op, in))$ to simulate the results for update query (op, in). Eventually, $\mathcal{A}$ outputs a bit.

Definition 1. *A DSSE scheme Γ is $\mathcal{L}$-adaptively-secure if for every PPT adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ such that*

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

4 Constructions

In this section, we give two DSSE constructions. In order to process range queries, we deploy a new binary tree which is modified from the binary tree in [13]. Now, we first give our binary tree used in our constructions.

4.1 Binary Tree for Range Queries

In a binary tree $\mathtt{BT}$, every node has at most two children named $left$ and $right$. If a node has a child, then there is an edge that connects these two nodes. The node is the parent *parent* of its child. The root *root* of a binary tree does not have parent and the leaf of a binary tree does not have any child. In this paper, the binary tree is stored in thew form of linked structures. The first node of $\mathtt{BT}$ is the root of a binary tree. For example, the root node of the binary tree $\mathtt{BT}$ is $\mathtt{BT}$, the left child of $\mathtt{BT}$ is $\mathtt{BT}.left$, and the parent of $\mathtt{BT}$'s left child is $\mathtt{BT}.left.parent$, where $\mathtt{BT} = \mathtt{BT}.left.parent$.

In a complete binary tree $\mathtt{CBT}$, every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible (the leaf level may not full). A perfect binary tree $\mathtt{PBT}$ is a binary tree in which all internal nodes (not the leaves) have two children and all leaves have the same depth or same level. Note that, $\mathtt{PBT}$ is a special $\mathtt{CBT}$.

4.2 Binary Database

In this paper, we use binary database BDB which is generated from DB. In DB, keywords (the first row in Fig. 1(c)) are used to retrieve the file indices (every column in Fig. 1(c)). For simplicity, we map keywords in DB to the values in the range $[0, m-1]$ for range queries[2], where m is the maximum number of values. If we want to search the range [0, 3], a naïve solution is to send every value in the range (0, 1, 2 and 3) to the server, which is not efficient. To reduce the number of keywords sent to the server, we use the binary tree as shown in Fig. 1(a). For the range query [0, 3], we simply send the keyword n_3 (the minimum nodes to cover value 0, 1, 2 and 3) to the server. In BDB, every node in the binary tree is the keyword of the binary database, and every node has all the file indices for its decedents, as illustrated in Fig. 1(d).

As shown in Fig. 1(a), keyword in BDB corresponding to node i (the black integer) is n_i (e.g. the keyword for node 0 is n_0.). The blue integers are the keywords in DB and are mapped to the values in the range [0, 3]. These values are associated with the leaves of our binary tree. The words in red are the file indices in DB. For every node (keyword), it contains all the file indices in its descendant leaves. Node n_1 contains f_0, f_1, f_2, f_3 and there is no file in node n_4 (See Fig. 1(d)). For a range query $[0, 2]$, we need to send the keywords n_1, n_4 (n_1 and n_4 are the minimum number of keywords to cover the range $[0, 2]$.) to the server, and the result file indices are f_0, f_1, f_2 and f_3.

Bit String Representation. We parse the file indices for every keyword in BDB (e.g. every column in Fig. 1(d)) into a bit string, which we will use later. Suppose there are $y-1$ documents in our BDB, then we need y bits to represent the existence of these documents. The highest bit is the sign bit (0 means positive and 1 means negative). If f_i contains keyword n_j, then the i-th bit of the bit string for n_j (every keyword has a bit string) is set to 1. Otherwise, it is set to 0. For update, if we want to add a new file index f_i (which also contains keyword n_j) to keyword n_j, we need a positive bit string, where the i-th bit is set to 1 and all other bits are set to 0. Next, we add this bit string to the existing bit string associated with n_j [3]. Then, f_i is added to the bit string for n_j. If we want to delete file index f_i from the bit string for n_j, we need a negative bit string (the most significant bit is set to 1), the i-th bit is set to 1 and the remaining bits are set to 0. Then, we need to get the complement of the bit string [4]. Next, we add the complement bit string as in the add operation. Finally, the f_i is deleted from the bit string for n_j.

[2] In different applications, we can choose different kinds of values. For instance, audit documents of websites with particular IP addresses. we can search the whole network domain, particular host or application range.

[3] Note that, in the range queries, the bit strings are bit exclusive since a file is corresponded to one value only.

[4] In a computer, the subtraction is achieved by adding the complement of the negative bit string.

For example, in Fig. 1(b), the bit string for n_0 is 000001, and the bit string for n_4 is 000000. Assume that we want to delete file index f_0 from n_0 and add it to n_4. First we need to generate bit string 000001 and add it to the bit string (000000) for n_4. Next we generate the complement bit string 111111 (the complement of 100001) and add it to 000001 for n_0. Then, the result bit strings for n_0 and n_4 are 000000 and 000001, respectively. As a result, the file index f_0 has been moved from n_0 to n_4.

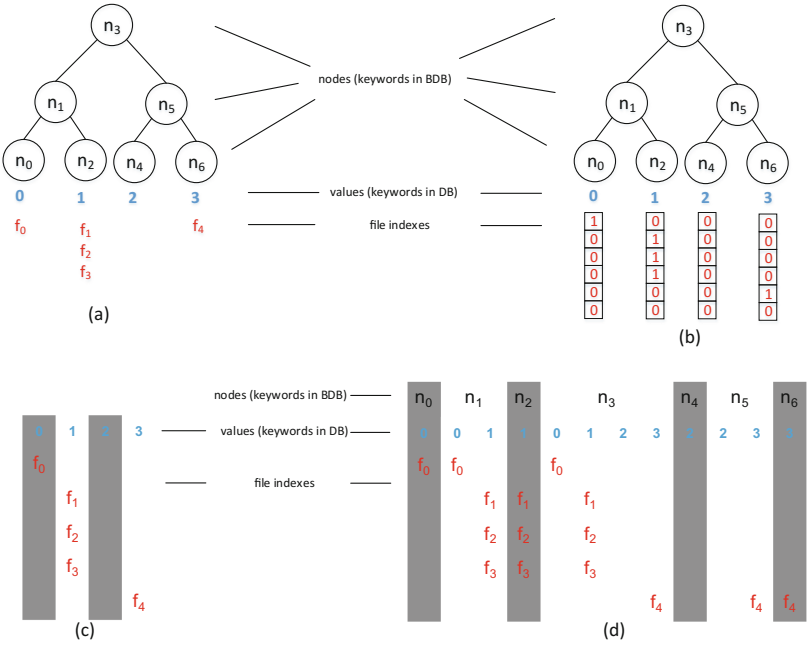

Fig. 1. Architecture of our binary tree for range query (Color figure online)

Binary Tree Assignment and Update. As we use the binary tree to support data structure needed in our DSSE, we define the following operations that are necessary to manipulate the DSSE data structure.

TCon(m): For an integer m, the operation builds a complete binary tree `CBT`. `CBT` has $\lceil log(m) \rceil + 1$ levels, where the root is on the level 0, and the leaves are on the level $\lceil log(m) \rceil$. All leaves are associated with the m consecutive integers from left to right.

TAssign(`CBT`): The operation takes a `CBT` as an input and outputs an assigned binary tree `ABT`, where nodes are labelled by appropriate integers. The operation applies **TAssignSub** recursively. Keywords then are assigned to the node integers.

TAssignSub(c, CBT): For an input pair: a counter c and CBT, the operation outputs an assigned binary tree. It is implemented as a recursive function. It starts from 0 and assigns to nodes incrementally. See Fig. 2 for an example.

Algorithm 1. Our Binary Tree

TCon(m)
Input integer m
Output complete binary tree CBT
1: Construct a CBT with $\lceil log(m) \rceil + 1$ levels.
2: Set the number of leaves to m.
3: Associate the leaves with m consecutive integers [0,m-1] from left to right.
4: **return** CBT

TAssign(CBT)
Input complete binary tree CBT
Output assigned binary tree ABT
1: Counter $c = 0$
2: **TAssignSub**(c, CBT)
3: **return** ABT

TAssignSub(c, CBT)
Input CBT, counter c
Output Assigned binary tree ABT
1: **if** CBT.$left \neq \perp$ **then**
2: **TAssignSub**(c, CBT.$left$)
3: **end if**
4: Assign CBT with counter c.
5: $c = c + 1$
6: **if** CBT.$right \neq \perp$ **then**
7: **TAssignSub**(c, CBT.$right$)
8: **end if**
9: Assign CBT with counter c.
10: $c = c + 1$
11: **return** ABT

TGetNodes(n, ABT)
Input node n, ABT
Output NSet
1: **NSet** $\leftarrow$ Empty Set
2: **while** $n \neq \perp$ **do**
3: **NSet** $\leftarrow$ **NSet** $\cup$ n
4: $n = n.parent$
5: **end while**
6: **return NSet**

TUpdate(add, v, CBT)
Input op= add, value v, CBT
Output updated CBT
1: **if** CBT $= \perp$ **then**
2: Create a node.
3: Associate value $v = 0$ to this node.
4: Set CBT to this node.
5: **else if** CBT is PBT or CBT has one node **then**
6: Create a new root node $\texttt{root}_n$.
7: Create a VBT = CBT
8: CBT.$parent$=VBT.$parent$=$\texttt{root}_n$
9: CBT = $\texttt{root}_n$
10: Associate v to the least virtual leaf and set this leaf and its parents as real.
11: **else**
12: Execute line 10.
13: **end if**
14: **return** CBT

TGetNodes(n, ABT): For an input pair: a node n and a tree ABT, the operation generates a collection of nodes in a path from the node n to the root node. This operation is needed for our update algorithm if a client wants to add a file to a leaf (a value in the range). The file is added to the leaf and its parent nodes.
TUpdate(add, v, CBT): The operation takes a value v and a complete binary tree CBT and updates CBT so the tree contains the value v. For simplicity, we consider the current complete binary tree contains values in the range $[0, v-1]$[5].

[5] Note that, we can use **TUpdate** many times if we need to update more values.

Depending on the value of v, the operation is executed according to the following cases:

- $v = 0$: It means that the current complete binary tree is null, we simply create a node and associate value $v = 0$ with the node. The operation returns the node as CBT.
- $v > 0$: If the current complete binary tree is a perfect binary tree PBT or it consists of a single node only, we need to create a virtual binary tree VBT, which is a copy of the current binary tree. Next, we merge the virtual perfect binary tree with the original one getting a large perfect binary tree. Finally, we need to associate the value v with the least virtual leaf (the leftmost virtual leaf without a value) of the virtual binary tree and set this leaf and its parents as real. For example, in Fig. 2(a), $v = 4$, the nodes with solid line are real and the nodes with dot line are virtual which can be added later. Otherwise, we directly associate the value v to the least virtual leaf and set this leaf and its parents as real [6]. In Fig. 2(b), $v = 5$.

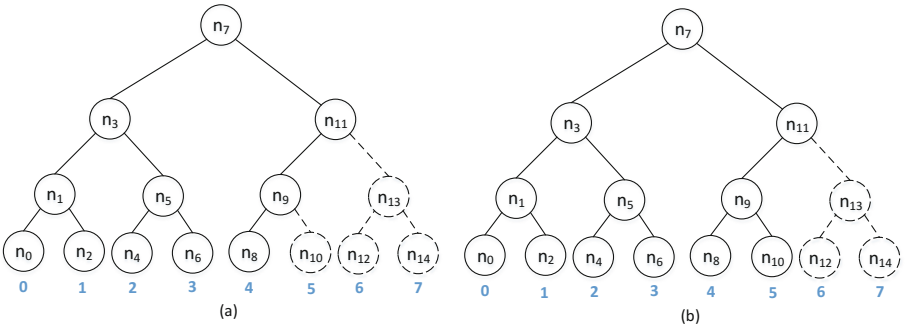

Fig. 2. Example of update operation

Note that, in our range queries, we need to parse a normal database DB to its binary form BDB. First, we need to map keywords of DB to integers in the range $[0, |W|-1]$, where $|W|$ is the total number of keywords in DB. Next, we construct a binary tree as described above. The keywords are assigned to the nodes of the binary tree and are associated with the documents of their descendants. For example, In Fig. 1a, the keywords are $\{n_0, n_1, \cdots, n_6\}$ and $\mathtt{BDB}(n_0) = \{f_0\}$, $\mathtt{BDB}(n_1) = \{f_0, f_1, f_2, f_3\}$.

4.3 DSSE Range Queries - Construction A

In this section, we apply our new binary tree to the Bost [5] scheme to support range queries. For performing a ranger query, the client in our scheme first determine a collection of keywords to cover the requested range. Then, she generates

[6] Only if its parents were virtual, then we need to convert them to real.

the search token corresponding to each node (in the cover) and sends them to the sever, which can be done in a similar way as [5]. Now we are ready to present the first DSSE scheme that supports range queries and is forward-secure. The scheme is described in Algorithm 2, where F is a cryptographically strong pseudorandom function (PRF), H_1 and H_2 are keyed hash functions and Π is a trapdoor permutation.

Setup(1^λ): For a security parameter 1^λ, the algorithm outputs $(\mathtt{TPK}, \mathtt{TSK}, K, \mathbf{T}, \mathbf{N}, m)$, where TPK and TSK are the public key and secret keys of the trapdoor permutation, respectively, K is the secret key of function F, $\mathbf{T}$, $\mathbf{N}$ are maps and m is the maximum number of the values in our range queries. The map $\mathbf{N}$ is used to store the pair keyword/(ST_c, c) (current search token and the counter c, please see Algorithm 2 for more details.) and is kept by the client. The map $\mathbf{T}$ is the encrypted database EDB that used to store the encrypted indices which is kept by the server.

Search$([a, b], \sigma, m, \mathtt{EDB})$: The protocol is executed between a client and a server. The client asks for documents, whose keywords are in the range $[a, b]$, where $0 \leq a \leq b < m$. The current state of EDB is σ and the integer m describes the maximum number of values. Note that knowing m, the client can easily construct the complete binary tree. The server returns a collection of file indices of requested documents.

Update$(add, v, ind, \sigma, m, \mathtt{EDB})$: The protocol is performed jointly by a client and server. The client wishes to add an integer v together with a file index ind to EDB. The state of EDB is σ, the number of values m. There are following three cases:

- $v < m$: The client simply adds ind to the leaf, which contains value v and its parents (See line 9–24 in Algorithm 2). This is a basic update, which is similar to the one from [5].
- $v = m$: The client first updates the complete binary tree to which she adds the value v. If a new root is added to the new complete binary tree, then the server needs to add all file indices of the old complete binary tree to the new one. Finally, the server needs to add ind to the leaf, which contains value v and its parents.
- $v > m$: The client uses **Update** as many times as needed. For simplicity, we only present the simple case $v = m$, i.e., the newly added value v equals the maximum number of values of the current range $[0, m-1]$, in the description of Algorithm 2.

The DSSE supports range queries at the cost of large client storage, since the number of search tokens is linear in the number of all nodes of the current tree instead of only leaves. In [5], the number of entries at the client is $|W|$, while it would be roughly $2|W|$ in this construction. Moreover, communication costs is heavy since the server needs to return all file indices to the client when the binary tree is updated with a new root. To overcome the weakness, we give a new construction with lower client storage and communication costs in the following section.

Algorithm 2. Construction A

Setup(1^λ)
Input security parameter 1^λ
Output $(\mathtt{TPK}, \mathtt{TSK}, K, \mathbf{T}, \mathbf{N}, m)$
$K \leftarrow \{0,1\}^\lambda$
$(\mathtt{TSK}, \mathtt{TPK}) \leftarrow \mathtt{TKeyGen}(1^\lambda)$
$\mathbf{T}$, $\mathbf{N} \leftarrow$ empty map
$m = 0$
return $(\mathtt{TPK}, \mathtt{TSK}, K, \mathbf{T}, \mathbf{N}, m)$

Search$([a,b], \sigma, m, \mathtt{EDB})$
Client:
Input $[a,b], \sigma, m$
Output (K_n, ST_c, c)
$\mathtt{CBT} \leftarrow \mathbf{TCon}(m)$
$\mathtt{ABT} \leftarrow \mathbf{TAssign}(\mathtt{CBT})$
RSet $\leftarrow$ Find the minimum nodes to cover $[a,b]$ in $\mathtt{ABT}$
for $n \in$ **RSet do**
  $K_n \leftarrow F_K(n)$
  $(ST_c, c) \leftarrow \mathbf{N}[n]$
  if $(ST_c, c) \neq \perp$ **then**
    Send (K_n, ST_c, c) to the server.
  end if
end for
Server:
Input (K_n, ST_c, c), $\mathtt{EDB}$
Output (ind)
Upon receiving (K_n, ST_c, c)
for $i = c$ to 0 **do**
  $UT_i \leftarrow H_1(K_n, ST_i)$
  $e \leftarrow \mathbf{T}[UT_i]$
  $ind \leftarrow e \oplus H_2(K_n, ST_i)$
  Output the ind
  $ST_{i-1} \leftarrow \Pi(\mathtt{TPK}, ST_i)$
end for

Update$(add, v, ind, \sigma, m, \mathtt{EDB})$
Client:
Input add, v, ind, σ, m
Output (UT_{c+1}, e)
$\mathtt{CBT} \leftarrow \mathbf{TCon}(m)$
if $v = m$ **then**
  $\mathtt{CBT} \leftarrow \mathbf{TUpdate}(add, v, \mathtt{CBT})$
  $m \leftarrow m + 1$
  if CBT added a new root **then**
    $(ST_c, c) \leftarrow \mathbf{N}[\mathtt{root}_o]$
    $\mathbf{N}[\mathtt{root}_n] \leftarrow (ST_c, c)$
  end if
  Get the leaf n_v of value v.
  $\mathtt{ABT} \leftarrow \mathbf{TAssign}(\mathtt{CBT})$
  NSet $\leftarrow \mathbf{TGetNodes}(n_v, \mathtt{ABT})$
  for every node $n \in$ **NSet do**
    $K_n \leftarrow F_K(n)$
    $(ST_c, c) \leftarrow \mathbf{N}[n]$
    if $(ST_c, c) = \perp$ **then**
      $ST_0 \leftarrow \mathcal{M}, c \leftarrow -1$
    else
      $ST_{c+1} \leftarrow \Pi^{-1}(\mathtt{TSK}, ST_c)$
    end if
    $\mathbf{N}[n] \leftarrow (ST_{c+1}, c+1)$
    $UT_{c+1} \leftarrow H_1(K_n, ST_{c+1})$
    $e \leftarrow ind \oplus H_2(K_n, ST_{c+1})$
    Send (UT_{c+1}, e) to the Server.
  end for
else if $v < m$ **then**
  Execute line 9-24.
end if
Server:
Input (UT_{c+1}, e), $\mathtt{EDB}$
Output $\mathtt{EDB}$
Upon receiving (UT_{c+1}, e)
Set $\mathbf{T}[UT_{c+1}] \leftarrow e$

4.4 DSSE Range Queries - Construction B

In this section, we give the second construction by leveraging the Paillier cryptosystem [17], which significantly reduce the client storage and communication costs compared with the first one. With the homomorphic addition property of the Paillier cryptosystem, we can add and delete the file indices by parsing them into binary strings, as illustrated in Sect. 4.2. Next we briefly describe our second

DSSE, which can not only support range queries but also achieve both forward and backward security. The scheme is described in Algorithm 3.

Setup(1^λ): For a security parameter 1^λ, the algorithm returns $(\mathtt{PK}, \mathtt{SK}, K, \mathbf{T}, m)$, where PK and SK are the public and secret keys of the Paillier cryptosystem, respectively, K is the secret key of a PRF F, m is the maximum number of values which can be used to reconstruct the binary tree and the encrypted database EDB is stored in a map $\mathbf{T}$ which is kept by the server.

Search$([a, b], \sigma, m, \mathtt{EDB})$: The protocol is executed between a client and a server. The client queries for documents, whose keywords are in the range $[a, b]$, where $0 \leq a \leq b < m$. σ is the state of EDB, and integer m specifies the maximum values for our range queries. The server returns encrypted file indices e to the client, who can decrypt e by using the secret key SK of Pailler Cryptosystem and obtain the file indices of requested documents.

Update$(op, v, ind, \sigma, m, \mathtt{EDB})$: The protocol runs between a client and a server. A requested update is named by the parameter op. The integer v and the file index ind specifies the tree nodes that need to be updated. The current state σ, the integer m and the server with input EDB. If $op = add$, the client generates a bit string as prescribed in Sect. 4.2. In case when $op = delete$, the client creates the complement bit string as given in Sect. 4.2. The bit string bs is encrypted using the Paillier cryptosystem. The encrypted string is denoted by e. There are following three cases:

- $v < m$: The client sends the encrypted bit string e with the leaf n_v containing value v and its parents to server. Next the server adds e with the existing encrypted bit strings corresponding to the nodes specified by the client. See line 11–23 in Algorithm 3 which is similar to the update in Algorithm 2.
- $v = m$: The client first updates the complete binary tree to which she adds the value v. If a new root is added to the new complete binary tree, then the client retrieves the encrypted bit string of the root (before update). Next the client adds it to the new root by sending it with the new root to the server. Finally, the client adds e to the leaf that contains value v and its parents as in $v < m$ case.
- $v > m$: The client uses **Update** as many times as needed. For simplicity, we only consider $v = m$, where m is the number of values in the maximum range.

In this construction, it achieves both forward and backward security. Moreover, the communication overhead between the client and the server is significantly reduced due to the fact that for each query, the server returns a single ciphertext to the client at the cost of supporting small number of documents. Since, in Paillier cryptosystem, the length of the message is usually small and fixed (e.g. 1024 bits).

This construction can be applied to applications, where the number of documents is small and simultaneously the number of keywords can be large. The reason for this is the fact that for a given keyword, the number of documents which contain it is small. Consider a temperature forecast system that uses a database, which stores records from different sensors (IoT) located in different

Algorithm 3. Construction B

Setup(1^λ)
Input security parameter 1^λ
Output $(\mathtt{PK}, \mathtt{SK}, K, \mathbf{T}, m)$
1: $K \leftarrow \{0,1\}^\lambda$
2: $(\mathtt{SK}, \mathtt{PK}) \leftarrow \mathtt{KeyGen}(1^\lambda)$
3: $\mathbf{T} \leftarrow$ empty map
4: $m = 0$
5: **return** $(\mathtt{PK}, \mathtt{SK}, K, \mathbf{T}, m)$

Search$([a,b], \sigma, m, \mathtt{EDB})$
Client:
Input $[a,b], \sigma, m$
Output (UT_n)
1: $\mathtt{CBT} \leftarrow \mathbf{TCon}(m)$
2: $\mathtt{ABT} \leftarrow \mathbf{TAssign}(\mathtt{CBT})$
3: **RSet** $\leftarrow$ Find the minimum nodes to cover $[a,b]$ in ABT
4: **for** $n \in$ **RSet do**
5: $UT_n \leftarrow F_K(n)$
6: Send UT_n to the server.
7: **end for**
Server:
Input (UT_n), EDB
Output (e)
8: Upon receiving UT_n
9: $e \leftarrow \mathbf{T}[UT_n]$
10: Send e to the Client.

Update$(op, v, ind, \sigma, m, \mathtt{EDB})$
Client:
Input op, v, ind, σ, m
Output (UT_n, e)
1: $\mathtt{CBT} \leftarrow \mathbf{TCon}(m)$
2: **if** $v = m$ **then**
3: $\mathtt{CBT} \leftarrow \mathbf{TUpdate}(add, v, \mathtt{CBT})$
4: $m \leftarrow m + 1$
5: **if** CBT added a new root **then**
6: $UT_{\mathtt{root}_o} \leftarrow F_K(\mathtt{root}_o)$
7: $UT_{\mathtt{root}_n} \leftarrow F_K(\mathtt{root}_n)$
8: $e \leftarrow \mathbf{T}[UT_{\mathtt{root}_o}]$
9: $\mathbf{T}[UT_{\mathtt{root}_n}] \leftarrow e$
10: **end if**
11: Get the leaf n_v of value v.
12: $\mathtt{ABT} \leftarrow \mathbf{TAssign}(\mathtt{CBT})$
13: **NSet** $\leftarrow \mathbf{TGetNodes}(n_v, \mathtt{ABT})$
14: **if** $op = add$ **then**
15: Generate the bit string bs as state in Bit String Representation of Sect. 4.2.
16: **else if** $op = del$ **then**
17: Generate the complement bit string bs as state in Bit String Representation of Sect. 4.2.
18: **end if**
19: **for** every node $n \in$ **NSet do**
20: $UT_n \leftarrow F_K(n)$
21: $e \leftarrow \mathtt{Enc}(\mathtt{PK}, bs)$
22: Send (UT_n, e) to the server.
23: **end for**
24: **else if** $v < m$ **then**
25: Execute line 11-23.
26: **end if**
Server:
Input (UT_n, e), EDB
Output EDB
1: Upon receiving (UT_n, e)
2: $e' \leftarrow \mathbf{T}[UT_n]$
3: **if** $e' \neq \perp$ **then**
4: $e \leftarrow e \cdot e'$
5: **end if**
6: $\mathbf{T}[UT_n] \leftarrow e$

cities across Australia. In the application, the cities (sensors) can be considered as documents and temperature measurements can be considered as the keywords. For example, Sydney and Melbourne have the temperature of 18°C. Adelaide and Wollongong have got 17°C and 15°C, respectively. If we query for cities, whose temperature measurements are in the range from 17 to 18°C, then the outcome includes Adelaide, Sydney and Melbourne. Here, the number of cities (documents) is not large. The number of different temperature measurements (keywords) can be large depending on requested precision.

5 Security Analysis

In our constructions, we parse a range query into several keywords. Then, following [5], the leakage to the server is summarized as follows:

- search pattern $\mathtt{sp}(w)$, the repetition of the query w.
- history $\mathtt{Hist}(w)$, the history of keyword w. It includes all the updates made to $\mathtt{DB}(w)$.
- contain pattern $\mathtt{cp}(w)$, the inclusion relation between the keyword w with previous queried keywords.
- time $\mathtt{Time}(w)$, the number of updates made to $\mathtt{DB}(w)$ and when the update happened.

Note that, contain pattern $\mathtt{cp}(w)$ is an inherited leakage for range queries when the file indices are revealed to the server. If a query w' is a subrange of query w, then the file index set for w' will also be a subset of the file index set for w.

5.1 Forward Security and Backward Security

Forward security means that an update does not leak any information about keywords of updated documents matching a query we previously issued. A formal definition is given below:

Definition 2. *([5]) A $\mathcal{L}$-adaptively-secure DSSE scheme Γ is forward-secure if the update leakage function $\mathcal{L}^{Updt}$ can be written as*

$$\mathcal{L}^{Updt}(op, in) = \mathcal{L}'(op, (ind_i, \mu_i))$$

where (ind_i, μ_i) is the set of modified documents paired with number μ_i of modified keywords for the updated document ind_i.

Backward security means that a search query on w does not leak the file indices that previously added and later deleted. More formally, we use the level I definition of [6] with modifications which leaks less information. It leaks the encrypted documents currently matching w, when they were updated, and the total number of updates on w.

Definition 3. *A $\mathcal{L}$-adaptively-secure DSSE scheme Γ is insertion pattern revealing backward-secure if the the update leakage function $\mathcal{L}^{Srch}$, $\mathcal{L}^{Updt}$ can be written as $\mathcal{L}^{Updt}(op, w, ind) = \mathcal{L}'(op)$, $\mathcal{L}^{Srch}(w) = \mathcal{L}''(\mathtt{Time}(w))$.*

5.2 Construction A

Since the first DSSE construction is based on [5], it inherits security of the original design. Adaptive security of the construction A can be proven in the Random Oracle Model and is a modification of the security proof of [5]. Due to page limitation, we give a sketch proof here, and refer the reader to the full version [24] for the full proof.

Theorem 1. *(Adaptive forward security of A). Let* $\mathcal{L}_{\Gamma_A} = (\mathcal{L}^{Srch}_{\Gamma_A}, \mathcal{L}^{Updt}_{\Gamma_A})$, *where* $\mathcal{L}^{Srch}_{\Gamma_A}(n) = (\mathit{sp}(n), \mathit{Hist}(n), \mathit{cp}(n))$, $\mathcal{L}^{Updt}_{\Gamma_A}(add, n, ind) = \perp$. *The construction A is* $\mathcal{L}_{\Gamma_A}$*-adaptively forward-secure.*

Proof. (Sketch) Compared with [5], this construction additionally leaks the contain pattern cp as described in Sect. 3.1. Other leakages are exactly the same as [5]. Since the server executes one keyword search and update one keyword/file-index pair at a time. Note that the server does not know the secret key of the trapdoor permutation, so it cannot learn anything about the pair even if the keyword has been searched by the client previously.

5.3 Construction B

The adaptive security of second DSSE construction relies on the semantic security of Paillier cryptosystem. All file indices are encrypted using the public key of Paillier cryptosystem. Without the secret key, the server cannot learn anything from the ciphertext. Due to page limitation, we give a sketch proof here and refer the reader to the full version [24] for the full proof.

Theorem 2. *(Adaptive forward security of B). Let* $\mathcal{L}_{\Gamma_B} = (\mathcal{L}^{Srch}_{\Gamma_B}, \mathcal{L}^{Updt}_{\Gamma_B})$, *where* $\mathcal{L}^{Srch}_{\Gamma_B}(n) = (\mathit{sp}(n))$, $\mathcal{L}^{Updt}_{\Gamma_B}(op, n, ind) = (\mathit{Time}(n))$. *Construction B is* $\mathcal{L}_{\Gamma_B}$*-adaptively forward-secure.*

Proof. (Sketch) In construction B, for the update, we only leak the number of updates corresponding to the queried keywords **n**. Since all cryptographic operations are performed at the client side where no keys are revealed to the server, the server can learn nothing from the update, given that the Paillier cryptosystem scheme is IND-CPA secure. We can simulate the DSSEREAL as in Algorithm 3 and simulate the DSSEIDEAL by encrypting all 0's strings for the EDB. The adversary $\mathcal{A}$ can not distinguish the real ciphertext from the ciphertext of 0's. Then, $\mathcal{A}$ cannot distinguish DSSEREAL from DSSEIDEAL. Hence, our Construction B achieves forward security. □

Theorem 3. *(Adaptive backward security of B). Let* $\mathcal{L}_{\Gamma_B} = (\mathcal{L}^{Srch}_{\Gamma_B}, \mathcal{L}^{Updt}_{\Gamma_B})$, *where* $\mathcal{L}^{Srch}_{\Gamma_B}(n) = (\mathit{sp}(n), \mathit{Hist}(n))$, $\mathcal{L}^{Updt}_{\Gamma_B}(op, n, ind) = (\mathit{Time}(n))$. *Construction B is* $\mathcal{L}_{\Gamma_B}$*-adaptively backward-secure.*

Proof. (Sketch) The construction B does not leak the type of update (either add or del) on encrypted file indices since it has been encrypted. Moreover, it does not leak the file indices that previously added and later deleted. The construction B is backward-secure. Since the leakage is same as Theorem 2, then the simulation is same as Theorem 2. □

6 Conclusion

In this paper, we give two secure DSSE schemes that support range queries. The first DSSE construction applies our binary tree to the scheme from [5] and is

forward-secure. However, it incurs a large storage overhead in the client and a large communication costs between the client and the server. To address these problems, we propose the second DSSE construction with range queries that uses Paillier cryptosystem. It achieves both the forward and backward security. Although the second DSSE construction cannot support large number of documents, it can still be very useful in certain applications. In the future, we would like to construct more scalable DSSE schemes with 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], the National Natural Science Foundation of China [grant number 61472364] and the Australian Research Council (ARC) Grant DP180102199. Josef Pieprzyk has been supported by National Science Centre, Poland, project registration number UMO-2014/15/B/ST6/05130.

References

1. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: Proceedings of the 2004 ACM SIGMOD Ninterational Conference on Management of Data, pp. 563–574. ACM (2004)
2. Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A.: Order-preserving symmetric encryption. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 224–241. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01001-9_13
3. Boldyreva, A., Chenette, N., O'Neill, A.: Order-preserving encryption revisited: improved security analysis and alternative solutions. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 578–595. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_33
4. Boneh, D., Lewi, K., Raykova, M., Sahai, A., Zhandry, M., Zimmerman, J.: Semantically secure order-revealing encryption: multi-input functional encryption without obfuscation. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 563–594. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_19
5. Bost, R.: $\Sigma o\varphi o\varsigma$: forward secure searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1143–1154. ACM (2016)
6. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1465–1482. ACM (2017)
7. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 668–679. ACM (2015)
8. Cash, D., et al.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: NDSS, vol. 14, pp. 23–26. Citeseer (2014)
9. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_20

10. Chenette, N., Lewi, K., Weis, S.A., Wu, D.J.: Practical order-revealing encryption with limited leakage. In: Peyrin, T. (ed.) FSE 2016. LNCS, vol. 9783, pp. 474–493. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-52993-5_24
11. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, pp. 79–88. ACM (2006)
12. Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., Garofalakis, M.: Practical private range search revisited. In: Proceedings of the 2016 International Conference on Management of Data, pp. 185–198. ACM (2016)
13. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Rich queries on encrypted data: beyond exact matches. In: Pernul, G., Ryan, P.Y.A., Weippl, E. (eds.) ESORICS 2015. LNCS, vol. 9327, pp. 123–145. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24177-7_7
14. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 965–976. ACM (2012)
15. Kasra Kermanshahi, S., Liu, J.K., Steinfeld, R.: Multi-user cloud-based secure keyword search. In: Pieprzyk, J., Suriadi, S. (eds.) ACISP 2017. LNCS, vol. 10342, pp. 227–247. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60055-0_12
16. Kim, K.S., Kim, M., Lee, D., Park, J.H., Kim, W.H.: Forward secure dynamic searchable symmetric encryption with efficient updates. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1449–1463. ACM (2017)
17. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48910-X_16
18. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings 2000 IEEE Symposium on Security and Privacy. S&P 2000, pp. 44–55. IEEE (2000)
19. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS, vol. 71, pp. 72–75 (2014)
20. 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: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9878, pp. 154–172. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45744-4_8
21. Wang, Y., Wang, J., Sun, S.-F., Liu, J.K., Susilo, W., Chen, X.: Towards multi-user searchable encryption supporting boolean query and fast decryption. In: Okamoto, T., Yu, Y., Au, M.H., Li, Y. (eds.) ProvSec 2017. LNCS, vol. 10592, pp. 24–38. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68637-0_2
22. 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)
23. Zuo, C., Macindoe, J., Yang, S., Steinfeld, R., Liu, J.K.: Trusted boolean search on cloud using searchable symmetric encryption. In: Trustcom/BigDataSE/ISPA, 2016 IEEE, pp. 113–120. IEEE (2016)
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. IACR Cryptology ePrint Archive (2018). http://eprint.iacr.org/