FFSSE: Flexible Forward Secure Searchable Encryption with Efficient Performance

Zheli Liu, Siyi Lv, Yu Wei[1], Jin Li[2], Joseph K. Liu[3], and Yang Xiang[4]

[1]College of Computer and Control Engineering, Nankai University, China
[2]School of Computer Science, Guangzhou University, China
[3]Department of Computer Science, the Monesh University, Australia
[4]Department of Computer Science, Swinburne University, Australia

Abstract

Searchable Symmetric Encryption (SSE) has been widely applied in the design of encrypted database for exact queries or even range queries in practice. In spite of its efficiency and functionalities, it always suffers from information leakages. Some recent attacks point out that forward privacy is the desirable security goal. However, there are only a very small number of schemes achieving this security. In this paper, we propose a new flexible forward secure SSE scheme, denoted as "FFSSE", which has the best performance in the literature, namely with fast search operation, fast token generation and $O(1)$ update complexity. It also supports both *add* and *delete* operations in the unique instance. Technically, we exploit a novel "key-based blocks chain" technique based on symmetric cryptographic primitive, which can be deployed in arbitrary index tree structures or key-value structures directly to provide forward privacy. In order to reduce the storage on the client side, we further propose an efficient permutation technique (with similar function as trapdoor permutation) to support the re-construction of the search tokens. Experiments show that our scheme is 4×, 300× and 300× faster than the $\Sigma o\phi o\varsigma$ (the state-of-the-art forward private SSE scheme proposed in CCS 2016) in search, update and token generation, respectively. Security analysis shows that our scheme is secure.

1 Introduction

Along with the popularity of outsourcing data to the cloud, data privacy becomes a critical consideration. To protect the privacy, users usually encrypt data before uploading them to database or storage server. However, encryption breaks the data availabilities, such that keyword search, range query or other functions cannot be directly applied over the ciphertext. To address the above issues, some cryptographic techniques have been proposed. Among them, searchable encryption (SE) is the mechanism to allow querying of the encrypted data without leaking data privacy to the cloud server.

Searchable Symmetric Encryption (SSE) is designed on symmetric cryptographic primitives and thus has very good performance (when compared to public key searchable encryption [1–3]). It allows a client to store encrypted documents on a server, then to retrieve all documents containing a certain keyword (or collection of keywords) at a later point. To do so, most efficient SSE schemes work in the following way: First, the client computes a search token t corresponding to the queried keyword k and sends t to the server. By using t, the server retrieves the documents identifiers containing the keyword w, then sends back them. (These document identifiers need not be the actual file name. They can be instead of the pointers to the encrypted files or even encrypted index of documents/records stored in the server.) The client then downloads the appropriate documents according to its requirements.

Now, SSE has been widely used in encrypted databases [4–6, 8]. Take the CryptDB [4] as an example, besides supporting SQL LIKE operator by using an SSE scheme [9] directly, it utilizes the SSE to implement the SQL equality queries (=, !=, IN, NOT IN, etc) when the values in the column are not unique. Recently, some researches [6, 7, 28–30] have been proposed to apply SSE to support rich queries, like conjunctive query [10], range query [11], and so on. Moreover, ARX [5] has applied SSE to provide equality query over the encrypted NoSQL databases.

1.1 Leakages and attacks of SSE

Although SSE can protect the privacy of the data content to a certain level, the deterministic encryption used in SSE makes the cloud server easy to observe the repeated queries and other leakages, which are modeled as *size patterns*, *query patterns* and *access patterns*:

- *Size pattern* [12]. It means that a server can know the number of keyword-document pairs of the stored data. Some of the SSE schemes may also leak the total number of messages and/or the total number of keywords.
- *Search pattern* [13]. It means that a server can know the repeated deterministic tokens. If the server can also know a repeated query, it is also called as "*query pattern*".
- *Access pattern* [13]. It means that a server can know the search results, including matching document identifiers of a keyword search and the document identifiers of the added/deleted documents.

Generally, these leakages can be mitigated by utilizing an oblivious RAM (ORAM) [18, 19]. Unfortunately, ORAM usually brings the massive storage space, huge bandwidth cost and lots of interactions between the user and the server for each keyword search. Thus, for the consideration of efficiency, almost all the existing works assume these leakage patterns are allowable under real-ideal [13], universal composable (UC) [25, 26], or other specific security models.

Potential attacks based on these leakages have not been thoroughly analyzed and exploited until inference attack conducted by Islam et al. [14] in 2012. However, in order to let the inference attack be successful, a server needs to know the content of all the original messages. In 2015, Cash et al. [15] studied more effective leakage-abuse attacks, in which count attack is used for query recovery by abusing co-occurrence patterns of keywords and size pattern. In 2016, Zhang et al. [16] studied file-injection attacks by abusing the file-access patterns. By injecting files to the client, the malicious server can know the query privacy even the plaintext of encrypted document. Kellaris et al. [17] developed the first attack against systems leaking only *communication volume*, in which *communication volume* means the server learns the number of returned items of a query. This attack can be even applied to the SSE schemes based on fully homomorphic encryption (FHE) or ORAM.

1.2 Need of forward privacy

The above attacks showed that even small leakage can be leveraged by an attacker to reveal the client's queries. Especially in the dynamic database, the server could inject new document to the client to obtain more advantages. Zhang et al. [16] showed that the file-injection attack can be devastating. They actually consider both adaptive and non-adaptive attacks. The adaptive attack refers to whether the server injects documents before or after the client's query is made. It is more effective, however, it cannot be applied to the forward secure schemes, which achieve the forward privacy [12]. This fact highlights the importance of forward privacy in any real-world deployment.

Forward privacy means a malicious server cannot learn if a newly added document matches previous search queries. From a practical point of view, it is important for the user to securely and dynamically build the encrypted database. As far as the authors know, there are only a limited number of schemes [11, 12, 20–23] in the literature claimed to have forward privacy. Most of them are suffered from inefficiencies, while the last scheme (i.e., $\Sigma o\phi o\varsigma$ [21]) is very efficient in practice for both searches and updates.

Table 1: Comparison with typical forward private SSE schemes. N is the number of entries (i.e. the number of keyword-document pairs) in the database, while $\mathcal{W}$ is the number of distinct keywords, $\mathcal{D}$ is the number of documents, and $\mathcal{M}$ is the length of search token. The n_w is the size of the search result set for keyword w, and a_w is the number of times the queried keyword w was historically added and deleted to the database, a_w^* is the a_w with a maximum value CT (when a_w^* is bigger than CT, it will be 1), ℓ is the first empty level which contains 2^ℓ blocks. Denote $\mathcal{AE}$ as asymmetric encryption, $\mathcal{SE}$ as symmetric encryption, μ as the length of data identifier, λ as the security parameters and $\mathcal{TT}$ as tree traverse operation.

Scheme	Computation				Communication		Storage
	Search	Update	Token Generation		Search	Update	Client
			search	update			
Previous works							
SPS14 [12]	$O(\min\{ a_w + \log N, n_w \log^3 N \})$	$O(\log^2 N)$	$O(1) \cdot \mathcal{SE}$	$O(2^\ell) \cdot \mathcal{SE}$	$O(n_w + \log N)$	$O(\log N)$	$O(N^\alpha)$
TWORAM [22]	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(\log^2 N)$	$O(n_w) \cdot \mathcal{SE}$	$O(n_w) \cdot \mathcal{SE}$	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(\log^3 N)$	$O(\log N)$
$\Sigma o\phi o\varsigma$-B [21]	$O(a_w) \cdot (\mathcal{AE} + \mathcal{TT})$	$O(1)$	$O(a_w) \cdot \mathcal{AE}$	$O(a_w) \cdot \mathcal{AE}$	$O(n_w)$	$O(1)$	$O(W(\log \mathcal{D} + \log \mathcal{M}))$
$\Sigma o\phi o\varsigma$ [21]	$O(a_w) \cdot (\mathcal{AE} + \mathcal{TT})$	$O(1)$	$O(a_w) \cdot \mathcal{AE}$	$O(a_w) \cdot \mathcal{AE}$	$O(n_w)$	$O(1)$	$O(W \log \mathcal{D})$
This works							
FFSSE	$O(a_w) \cdot (\mathcal{SE} + \mathcal{TT})$	$O(1)$	$O(1) \cdot \mathcal{SE}$	$O(1) \cdot \mathcal{SE}$	$O(n_w)$	$O(1)$	$O(W(\mu + \lambda))$
FFSSE$-\varepsilon$	$O(a_w^*) \cdot (\mathcal{SE} + \mathcal{TT})$	$O(1)$	$O(a_w^*) \cdot \mathcal{SE}$	$O(a_w^*) \cdot \mathcal{SE}$	$O(n_w)$	$O(1)$	$O(W \log \mathcal{D})$

1.3 Motivation

The $\Sigma o\phi o\varsigma$ protocol proposed by Bost [21] in 2016 is the first efficient forward private SSE scheme to our knowledge. Its construction is based on the inverted index. For each element in the indexed list $\mathbb{L}_w = (ind_0, \cdots, ind_n)$ of keyword w, $\Sigma o\phi o\varsigma$ encrypts it and stores it at a (logical) location. Its idea is simple: for preventing the malicious server from knowing where the newly added document belongs to, it generates the storage location based on the inverse of a one-way trapdoor permutation (TDP), which can only be performed in the client side with his/her private key. During the search operation, the protocol allows the server to re-compute these storage positions based on the evaluation of permutation using the public information. By this way, it achieves the optimal $O(1)$ update complexity and $O(a_w)$ search computation complexity, where a_w denotes the number of times the queried keyword w was historically added to the database.

The outstanding contribution of $\Sigma o\phi o\varsigma$ is exploiting a novel design of efficient forward private SSE without using ORAM. It inspires us to study similar techniques with higher performance and broader application scopes, aiming at improving the following limitations of $\Sigma o\phi o\varsigma$:

- *Asymmetric primitive.* The TDP technique in $\Sigma o\phi o\varsigma$ relies on the asymmetric cryptographic primitive, namely, the search operation requires $O(a_w)$ times asymmetric encryptions and decryptions (implemented by RSA). As we all know, asymmetric primitive is more expensive than symmetric primitive. Therefore we focus on *how to exploit the new technique based on symmetric cryptographic primitive which has the same effect as TDP, as to further improve its performance*;

- *Dependence on token generation rule.* The storage position in $\Sigma o\phi o\varsigma$ must be generated based on the repeated operation by the private key of the asymmetric primitive. That is, it specifies the token generation rule. If we regard the (local) position (even the token) as the *index*, the *index* generation must rely on the token generation rule. So that, the TDP in $\Sigma o\phi o\varsigma$ conflicts with the storage structure with its own index generation rule. This limitation would limit its application scopes. Therefore we also focus on *how to exploit the new technique independent of token generation rule, to provide forward private security for the existing structures with its own index generation rule.*

1.4 Our Contributions

We construct a flexible forward secure searchable encryption scheme based on the symmetric primitives, denoted as "FFSSE", which has the most efficient performance until now (fast search operation and token generation, $O(1)$ update complexity), and supports both add and delete operations in the unique instance. As shown in Table 1, it can be seen that: 1) FFSSE and $\Sigma o\phi o\varsigma$s ($\Sigma o\phi o\varsigma$-B and $\Sigma o\phi o\varsigma$) have the best asymptotic complexity in search and update. But in fact, FFSSE improves 4× search and 300× update than $\Sigma o\phi o\varsigma$-B

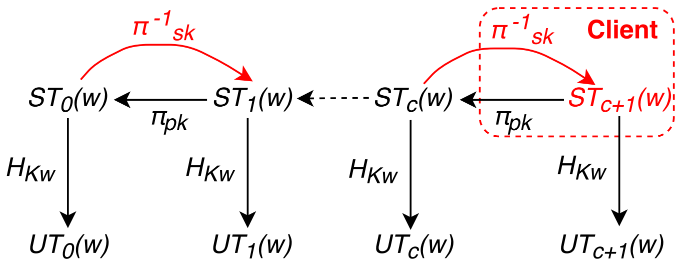

Figure 1: Trapdoor permutation (TDP) technique. Operations in red can only be done by the client, using the secret key sk.

(100× search and at least 300× update than $\Sigma o\phi o\varsigma$) through experiments, owing to the adopted symmetric cryptographic primitive; 2) FFSSE has the most efficient token generation among all the forward private SSE schemes.

In addition to the basic scheme, we propose an improved scheme, denoted as FFSSE-ε, to further reduce the client storage to the same as $\Sigma o\phi o\varsigma$. We remove the tree traverse operations and reduce the number of nodes in the index tree, by dynamic merging the document identifiers of the same keyword. Compared to $\Sigma o\phi o\varsigma$, the search, update and token generation of FFSSE-ε improve 60×, 200× and 200× respectively through experiments.

Technically, we exploit: 1) a novel "key-based blocks chain" (KBBC) method based on symmetric cryptographic primitive. Apart from the good performance, it does not rely on the index generation rule, so that it can be deployed in arbitrary index tree structures or key-value structures to provide forward private security; 2) an efficient permutation technique (with similar function as TPD), denoted as "ε-sTDP". It adopts KBBC technique to link the blocks and achieve the forward privacy. Meanwhile, it applies the one-way permutation $\mathcal{P}$ (not necessary to have a trapdoor and can be instantiated by symmetric cryptographic primitive) instead of TDP to re-generate the tokens and reduce the client storage. It is similar to the usage of TDP in $\Sigma o\phi o\varsigma$, but has a higher performance.

2 Related work

2.1 Searchable symmetric encryption

The first practical searchable encryption scheme was introduced by Song et al. [9] in 2000, using only symmetric primitives. Later, known as Symmetric Searchable Encryption (SSE) schemes, several schemes based on this similar concept were proposed. There are two kinds of SSE scheme, static one and dynamic one. Because the static SSE doesn't support dynamically adding or deleting keyword-document pairs after initializing the database or storage system, recent researches mainly focus on the constructions of dynamic SSE.

Among the design approaches of SSE, approach based on inverted index is used widely. In this approach, an index is in the form of (*key*, *value*), where the *key* is a keyword and the *value* consists of a list of document identifiers associated with the keyword. Compared to other approaches, it achieves the sublinear search time. This is because searching for a keyword immediately returns the list of document identifiers matching the keyword. In 2006, Curtmola et al. [13] firstly introduced this approach, and subsequent schemes are proposed by Kamara et al. [24], Kurosawa and Ohtaki [25], Cash et al. [8], Naveed et al. [26], Stefanov et al. [12] and Bost [21], and so on.

However, creating a dynamic scheme based on this approach faces a great challenge. When adding/removing a document, each of the keyword tokens must be linearly scanned through to add/remove a message entry. Fortunately, the novel design of Bost [21] provides not only the forward private security, but also the efficient solution for this challenge.

2.2 Trapdoor permutation technique

We review the trapdoor one-way permutations described in [27]. A trapdoor permutation family $\prod$ over a group D comprises the following three algorithms:

- $Generate(1^\lambda)$ is a randomized generation algorithm. Its input is 1^λ and its output is the description s of a permutation along with the corresponding trapdoor t.
- $Evaluate(s, x)$ is a deterministic algorithm. Its inputs are the permutation description s and a value $x \in D$. Its outputs is $a \in D$, the image of x under the permutation.
- $Invert(s, t, a)$ is a deterministic algorithm. Its inputs are the permutation description s, the trapdoor t and a value $a \in D$. It outputs the preimage of a under the permutation.

The output of the $Generate$ algorithm is a probability distribution Π on permutations, therefore $(\pi, \pi^{-1}) \xleftarrow{R} \Pi$, where π is a permutation and π^{-1} is the inverse permutation. For a permutation with the description of s, we have $Invert(s, t, Evaluate(s, x)) = x$.

Definition 1. *The advantage of algorithm* A *in inverting a trapdoor permutation family is:*
$Adv_{\mathrm{A}}^{Invert} = P[x = \mathrm{A}(s, Evaluate(s, x)) : (s, t) \xleftarrow{R} Generate, x \xleftarrow{R} D]$.

TDP in $\Sigma o\phi o\varsigma$. In $\Sigma o\phi o\varsigma$, TDP is built by the asymmetric primitive. As shown in Figure 1, only the client who owes the trapdoor (private key sk) can generate the permutation, but the server who owes the public info (public key pk) can evaluate this permutation. We can also see that the permutation about w is built by encrypting a random value $ST_0(w)$ for many times using the trapdoor sk (i.e., π_{sk}^{-1}), meanwhile, it can be retrieved by decrypting the last one using the public info pk (i.e., π_{pk}).

The $\Sigma o\phi o\varsigma$ uses the TDP to construct a forward secure SSE. Most specially, for each keyword w, there is a permutation over the set which contains the search tokens of keyword w denoted by $D(w)$. The client maintains a counter c and its search token $ST_c(w)$ for each keyword w. When adds a keyword-document pair for keyword w, the client will first produce a search token by $ST_{c+1}(w) \leftarrow \pi_{sk}^{-1}(ST_c(w))$, and then produce a storage position (called as update token) $UT_{c+1}(w)$ using a keyed hash. Finally update its client state and store the document identifier into $UT_{c+1}(w)$ in the server. Since the latest search token is stored in the client, the malicious server cannot know where the $UT_{c+1}(w)$ is produced from.

3 Preliminaries

3.1 Notations

Denote $\mathrm{negl}(\lambda)$ as a negligible function, where λ is the security parameter. Denote $x \xleftarrow{\$} X$ as x is uniformly sampled from the finite set X. In general, the symmetric key is the λ bits string uniformly sampled from $\{0,1\}^\lambda$. Denote $E_k(m)$ as the symmetric encryption of the message m by the key k, $D_k(c)$ as the symmetric decryption of the ciphertext c by the key k. Denote $\mathcal{F}$ as the symmetric encryption scheme selected in the real construction. Moreover, denote $\mathbf{H}(k, m)$ as a keyed hash function with the key k and message m as inputs.

Denote $\mathbb{W}$ as the keyword set and $\mathcal{W}$ as the number of distinct keywords ($\mathcal{W} = |\mathbb{W}|$). Denote $\mathbb{W}_x \subseteq \{0,1\}^*$ as the x-th keyword, and thus $\mathbb{W} = \bigcup_{i=1}^{\mathcal{W}} \mathbb{W}_i$. Denote $\mathbb{D}$ as the document set and $\mathcal{D}$ as the number of documents ($\mathcal{D} = |\mathbb{D}|$); Denote $ind_x \in \{0,1\}^\mu$ as the x-th document identifier with the bit length of μ, and thus $\mathbb{D} = (ind_i)_{i=1}^{\mathcal{D}}$.

In general, we denote w as the keyword, a_w as the number of times the queried keyword w was historically *added* and *deleted* to the *database* DB, and n_w as the size of search results for keyword w. Denote $\mathrm{DB}(w)$ as the set of documents containing keyword w in the *database* DB, and N as the total number of keyword-document pairs which the database supports. Denote $\mathbf{Hist}(w)$ as the history of keyword w. It lists all the modifications made to $\mathrm{DB}(w)$ over the time.

3.2 Dynamic searchable symmetric encryption

We review the general definition of *dynamic searchable encryption scheme* in [21], $\mathcal{DSSE} = (\textbf{Setup}, \textbf{Search}, \textbf{Update})$, containing an algorithm and two client-server protocols:

- **Setup**(DB): it is an algorithm for setting up the whole encrypted database supporting keyword search. It takes a *database* DB as input, outputs a pair (EDB, K, σ), where EDB is the encrypted database, K is the secret key contained by the client and σ is the client's state.

- **Search**(K, q, σ; EDB) = (Search$_C$(K, σ, q), Search$_S$(EDB)): it is a client-server protocol for performing a search query. The Search$_C$(K, σ, q) in the client takes the key K and its state σ as inputs, outputs a query q; the Search$_S$(EDB) in the server takes the EDB as input, outputs the results as document identifiers matching the query q. For single-keyword search schemes, a search query is restricted to a unique keyword w.

- **Update**(K, σ, *op*, *in*; EDB) = (Update$_C$(K, σ, *op*, *in*), Update$_S$(EDB)): it is a client-server protocol supporting update operations of a document. The update operations are taken from the set {*add*, *del*}, meaning, respectively, the addition and the deletion of a document/keyword pair. To do these update operations, the client takes inputs as the key K, an operator *op*, client's state σ and an input *in* parsed as the index *ind* and a set W of keywords; while the server takes the EDB as input.

3.3 Forward privacy security

Leakage definition. Define $\mathcal{L} = (\mathcal{L}^{Setup}, \mathcal{L}^{Search}, \mathcal{L}^{Update})$ as the leakage function, describing what protocols in $\mathcal{FFSSE}$ leak to the adversary. More formally, the leakage function $\mathcal{L}$ will keep as state the *query list* Q: the list of all queries issued so far, and for a search query on keyword w, entries are (i, w) , or for an **op** update query with input **in**, entries are $(i, \textbf{op}, \textbf{in})$. The integer i is a timestamp, initially set to 0, and it increments with the query times. Define $\texttt{sp}(x)$ and $\texttt{qp}(x)$ respectively denote the search and query patterns, formally

$\texttt{sp}(x) = \{j : (j, x) \in Q\}$ (only matches search queries)
$\texttt{qp}(x) = \{j : (j, x) \in Q$ or
$(j, \textbf{op}, \textbf{in})$ and x appears in **in** $\}$.

Forward privacy. We review the definition of forward privacy described in [21].

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

$$\mathcal{L}^{Update}(\boldsymbol{op}, \boldsymbol{in}) = \mathcal{L}'(\boldsymbol{op}, \{(ind_i, u_i)\})$$

where $\{(ind_i, u_i)\}$ is the set of modified documents paired with the number u_i of modified keywords for the updated document ind_i.

3.4 Key-based blocks chain

Before giving the details of our SSE protocol, we first present our proposed "key-based blocks chain" (KBBC) technique to link arbitrary number of data blocks in a set $\mathcal{B}$ as a chain $\mathcal{C}$ while hiding their relations. Define $\mathbb{B}$ as the data blocks set, $\mathbb{B} = \{b_0, \cdots, b_n\}$ and $|\mathbb{B}| = n$. Denote $\mathcal{C}$ as a chain built over the subset $\mathcal{B}$ in $\mathbb{B}$, i.e., $\mathcal{B} \subseteq \mathbb{B}$. Denote n_c as the number of chains over the set $\mathbb{B}$. Assume a block can only belong to one chain, then $1 \leq n_c \leq n$.

Block definitions. We define three types of blocks: *head block*, *tail block* and *internal block*, which are the first block, last block and other blocks in a chain $\mathcal{C}$, denoted by $\mathcal{C}.head$, $\mathcal{C}.tail$ and $\mathcal{C}.internal$, respectively. For each data block b in set $\mathbb{B}$, define it as b=(*id*, *value*, *key*, *ptr*), where *id* and *value* are data identifier and data value of this block, the *ptr* and *key* are data identifier and encryption key of its next block. Denote *b.id* as the *id* of block b and others are similar.

KBBC algorithms. There are three algorithms in KBBC:

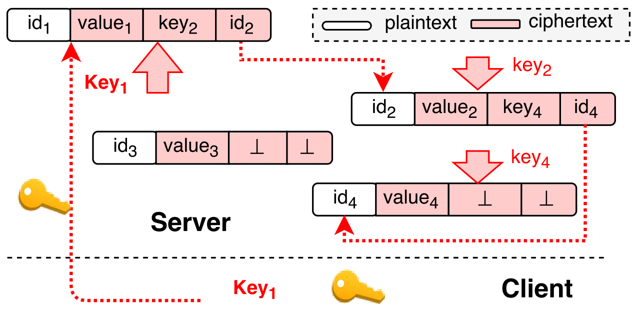

Figure 2: An example of key-based blocks chain. There are four blocks with data identifiers of id_1, id_2, id_3 and id_4, whose encryption keys are key_1, key_2, key_3 and key_4 respectively. The *head block* in the chain is the id_1 whose encryption key is maintained in the client; the id_2 is a *internal block* and its key is stored in the previous block id_1; the *tail block* is the id_4, whose key is stored in the previous block id_2, however, its ptr value is $\perp$ because it is the end of this chain.

- $Init()$: it initializes a blocks chain. It takes none input but outputs a description $\mathcal{C}$ for this chain.
- $AddHead(\mathcal{C}, id, value, 1^\lambda)$: it adds a *head block* to the chain. It takes chain description $\mathcal{C}$, the block identifier id, block value $value$ and security parameter 1^λ as inputs, and outputs a new head block b. It has four steps: 1) generate a block as b=(id, $value$, $\mathcal{C}.head.key$, $\mathcal{C}.head.id$); 2) sample a random key k from $\{0,1\}^\lambda$; 3) utilizes k to encrypt contents of block b except id; 4) store b in the server.
- $Retrieve(\mathcal{C}, id, k)$: it retrieves a next block in the chain. It takes the chain description $\mathcal{C}$, block identifier id and block key k as inputs, and outputs the identifier and encryption key of the next block. It has tree steps: 1) find the block b by its identifier id; 2) decrypt block b by its key k; 3) output the obtained identifier $b.ptr$ and key $b.key$.

Build a blocks chain. To build a blocks chain, we first execute $Init()$ to initialize the chain $\mathcal{C}$; then, for each block $b_i \in \mathcal{B}$ ($\mathcal{B} \subseteq \mathbb{B}, 0 \leq i \leq |\mathcal{B}| - 1$), we execute $AddHead(\mathcal{C}$, $b_i.id$, $b_i.value$, $1^\lambda)$ repeatedly, until all blocks are added to the chain $\mathcal{C}$. In fact, its idea is very simple: to link the blocks, the ptr and key of a block are set to the data identifier and encryption key of its next block (but for the *tail block*, they are set to $\perp$); to hide the relations, the identifier and encryption key of each block are encrypted and stored in its previous block (but for the *head block*, they are stored in the client). Figure 2 shows an example of a chain built from blocks with identifiers of id_1, id_2 and id_4.

Forward privacy. The KBBC algorithms are simple but powerful. Obviously, it can provide forward privacy for the newly added block, because its next block identifier is encrypted by a random key stored in the client.

Theorem 1. *(AddHead leaks no information). Define* $\mathcal{L}_S = (\mathcal{L}_S^{Retrieve}, \mathcal{L}_S^{AddHead})$, *where*
$\mathcal{L}_S^{Retrieve} = (n_c, num_c, \textbf{\textit{Hist}}(\mathcal{C}))$, $\mathcal{L}_S^{AddHead} = \perp$.
KBBC is $\mathcal{L}_S - forward - privacy$.

Proof. If $\mathcal{C}$ has already been retrieved, the adversary knows the number of blocks which have already been in $\mathcal{C}$ (denote as $num_\mathcal{C}$) and the history of $\mathcal{C}$ including id, $value$, key of the retrieved blocks (denote as $\textbf{Hist}(\mathcal{C})$). If all the chains of $\mathbb{B}$ have been retrieved, it leaks the number of chains over the set $\mathbb{B}$ (denote as n_c). Therefore,

$$\mathcal{L}_S^{Retrieve} = (n_c, num_c, \textbf{Hist}(\mathcal{C})).$$

According to the algorithm KBBC.$AddHead$, when we add a new *head block* to $\mathcal{C}$ which has already been retrieved, we will generate a new block as b=(id^*, $value^*$, $\mathcal{C}.head.key$, $\mathcal{C}.head.id$), which is encrypted by a new key k stored in client side. Without the key k, the adversary can not decrypt b, so that the added *head block* and blocks of $\mathcal{C}$ are indistinguishable. Moreover, the adversary can only know that we add a block into $\mathbb{B}$, but cannot know which chain it belongs to and weather generate a new chain. Therefore,

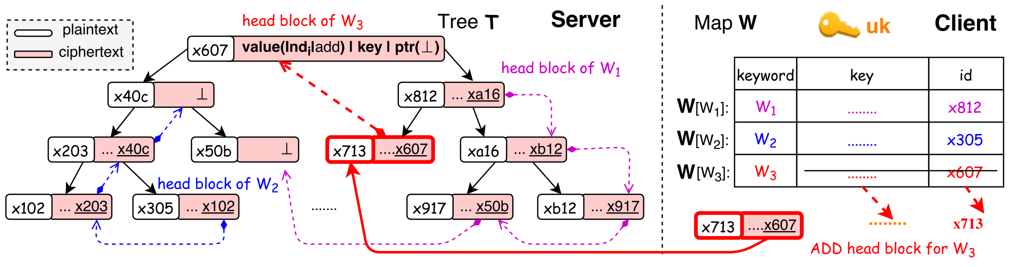

Figure 3: Storage structure and operations of FFSSE. There are three keywords (W_1, W_2 and W_3) and their inverted index lists ($\mathcal{L}_{W_1}$, $\mathcal{L}_{W_2}$ and $\mathcal{L}_{W_3}$) in a *database* DB. The client stores the encryption key, identifier of *head block* of each list and a counter in the map $\mathbf{W}$. The server stores each document identifier in these lists into the *value* part of a data block in a node of the tree $\mathcal{T}$. The root node shows the whole parts of a data block.

$$\mathcal{L}_S^{AddHead} = \perp,$$

the adversary cannot get any advantage to learn more additional information through $AddHead$ operation. □

Independent of index generation rule. In KBBC, the index (data identifier) can be generated by random selection or some algorithms according to application requirements. Compared to TDP in $\Sigma o\phi o\varsigma$, it is an important characteristic, which allows building a chain with forward privacy over the existing structures with its own index generation rule. It is more flexible and may therefore be valuable for some other designs with the consideration of forward security.

In fact, we can regard each block as a "**key-value**" structure, where the *id* with clear text is the **key**, which may be the index or position in the concrete storage structure, other encrypted parts are the **value**. So that, KBBC can be applied in most popular index tree (like AVL, B-tree) or key-value storage structures, to provide the forward private security. In other words, it is suitable for both relational databases and NoSQL databases.

4 The FFSSE construction

In this section, we describe our proposed forward secure searchable encryption scheme (denoted as "FFSSE"), which is based on our "key-based blocks chain".

4.1 Storage structure

Like the $\Sigma o\phi o\varsigma$, we consider the SSE construction with inverted index scheme (such as [13] and derived works). For each keyword $w \in \mathbb{W}$, denote $\mathbb{L}_w$ as an indexed list to store the document identifiers ($ind_0, \cdots, ind_{n_w}$) containing the keyword w, where n_w is the size of $\mathbb{L}_w$, that is, $n_w = |\mathbb{L}_w|$.

In the cloud server, we adopt the binary search tree $\mathcal{T}$ as an example (other index tree or key-value structures are also OK) to store the data blocks defined in Section 3.4. Each node stores a data block. We denote $\mathcal{T}[id]$ as the node indexed by id, and $\mathcal{T}[id].data$ as the stored data. Each element (document identifier) in the list $\mathbb{L}_w$ is stored in the *value* part of a data block. All the elements in every indexed list are stored into the tree $\mathcal{T}$, based on the "key-based blocks chain" described in Section 3.4. As a result, there are total $\mathcal{W}$ inverted lists containing $\mathcal{N}$ nodes stored in the $\mathcal{T}$, whose depth is $\log \mathcal{N}$.

In the client, we adopt a map $\mathbf{W}$ to store the state of each keyword. The keyword state of w can be defined as the tuples of st_w=(id, key), where id and key are data identifier and encryption key of the *head block* in the inverted list $\mathbb{L}_w$. The default value of id is $\perp$. For a keyword $w \in \mathbb{W}$, $\mathbf{W}[w]$ is mapped to its state st_w.

Figure 3 shows this storage structure. To support both add and delete operations, we actually store the document identifier ind and its operator op into the *value* of a block. More specifically, "$ind \| add$" denotes the add operation but "*ind*‖*del*" denotes the delete operation for document *ind*, where "‖" denotes the concatenation of two strings. See the root node in tree $\mathcal{T}$ of Figure 3 for an example. In practice, the *op* can be defined as a byte, for example, 0 for *add* but 1 for *del*. Moreover, besides the map $\mathbf{W}$, the client should securely store the secrete key *uk*.

4.2 Our construction

Algorithm 1 FFSSE: Forward private SSE scheme supporting both add and delete operation

```
Setup()
 1: $uk \xleftarrow{\$} \{0,1\}^\lambda$
 2: $\mathbf{W} \leftarrow empty\ map,\ \mathcal{T} \leftarrow empty\ tree$
Search($w,\ \sigma$; EDB)
    Client:
 1: $(id, key) \leftarrow (\mathbf{W}[w].id, \mathbf{W}[w].key)$
 2: Send token $(id, key)$ to the server.
    Server:
 3: $\mathbf{S} \leftarrow empty\ set,\ i \leftarrow 0$
 4: repeat
 5:     $b \leftarrow D_{key}(\mathcal{T}[id].data)$
 6:     $id \leftarrow b.ptr$
 7:     $key \leftarrow b.key$
 8:     $ind \| op \leftarrow b.value$
 9:     $\mathbf{S}[i{+}{+}] \leftarrow ind \| op$
10: until $(b.ptr == \perp)$
11: $\mathbf{S} \leftarrow Merge(\mathbf{S})$
    // merge the same $ind$ with $add$ and $del$ operations
12: Send each $ind$ in $\mathbf{S}$ to the client.
Update($op,\ w,\ ind,\ \sigma$; EDB)
    Client:
 1: $(id, key) \leftarrow (\mathbf{W}[w].id, \mathbf{W}[w].key)$
 2: $r \xleftarrow{\$} \{0,1\}^\lambda$
 3: $id^* \leftarrow \mathbf{H}(uk, r)$
 4: $key^* \leftarrow \mathrm{KeyGen}(1^\lambda)$
 5: $b \leftarrow (id^*, E_{key^*}(\text{"}ind \| op\text{"} \| key \| id))$
 6: $\mathbf{W}[w].id \leftarrow id^*$
 7: $\mathbf{W}[w].key \leftarrow key^*$
 8: Send block $b$ to the server.
    Server:
 9: Insert block $b$ into the tree $\mathcal{T}$.
```

Algorithm 1 gives the formal description of our FFSSE scheme. It allows only insertion of data block into tree $\mathcal{T}$, and supports both *add* and *delete* operations of keyword-document pairs. In the pseudo code, $\mathbf{H}$ is a keyed hash functions whose output is μ bits long. Encryption $E_k(m)$ and decryption $D_k(c)$ are implemented by an IND-CPA (indistinguishability against the chosen plaintext attack) secure symmetric cryptographic primitive.

Update operation. When the client wants to update (*add* or *delete*, denoted as *op*) a document (with identifier *ind*) matching w, it runs KBBC.$AddHead(\mathbb{L}_w, \cdot)$. More specifically, it firstly products a new data block b, whose identifier is generated based on a random value sampled from $\{0,1\}^\lambda$. The *b.value* is set to the document identifier and its operation, i.e., *ind*‖*op*; the *b.key* and *b.ptr* are set to $\mathbf{W}[w].key$ and $\mathbf{W}[w].id$

respectively, that is, encryption key and data identifier of current *head block*. Then, the client samples a symmetric key key^* with KeyGen(1^λ) function, and utilizes it to encrypt the *value*, *key* and *ptr*. Next, it sends the block b to the server, and updates the state of keyword w as $\mathbf{W}[w].id = b.id$ and $\mathbf{W}[w].key = key^*$. Finally, the server will insert this block into the tree $\mathcal{T}$ to finish this process. Figure 3 shows the details of inserting a new keyword-document pair into the inverted list of $\mathbb{W}_3$.

Search operation. When the client performs a search query on w, it will issue a search token t that will allow the server to retrieve the document identifiers matching w. To do so, the client can only generate the search token as the keyword state stored in the map $\mathbf{W}[w]$, i.e., $t \leftarrow (\mathbf{W}[w].id, \mathbf{W}[w].key)$. After receiving the search token t, the server can retrieve the blocks chain one by one by running KBBC.$Retrieve(\mathbb{L}_w, \cdot)$ repeatedly, until the *tail block* is visited. For each node in the chain, the server decrypts its *value* and obtains the stored document identifier. If a document identifier has both the *add* and *del* operators, it should be ignored. In Algorithm 1, a function $Merge$ is defined to merge the same document identifier with different operations. In the end, the server returns all the document identifiers matching w to the client to finish this process.

Correctness. The correctness of FFSSE is quite straightforward. Like that in $\Sigma o\phi o\varsigma$, the only issue is collision among the data identifier id generated from $\mathbf{H}$ with input (uk, r). We can relate the correctness to the collision resistance of $\mathbf{H}$. The output of $\mathbf{H}$ is μ bits long. In practice, μ should satisfy the condition that $N^2/2^\mu$ is negligible in the security parameter λ. Let N_{max} be the maximum number of keyword-document pairs which the database can support. Therefore, setting $\mu = \lambda + 2\log N_{max}$ is thoughtful.

4.3 Security analysis

The adaptive security of FFSSE can be proven in the Random Oracle Model.

Theorem 2. *(Adaptive security of FFSSE). Define* $\mathcal{L}_S = (\mathcal{L}_S^{Search}, \mathcal{L}_S^{Update})$, *where*
$\mathcal{L}_S^{Search} = (\boldsymbol{sp}(w), \boldsymbol{Hist}(w))$, $\mathcal{L}_S^{Update}(op, w, ind) = \perp$.
FFSSE is $\mathcal{L}_S - adaptive - secure$.

Proof. Let λ be the security parameter. Deriving some games from real world game will help to prove the theorem.

Game G_0. The G_0 is the real world FFSSE security game $SSEReal_A^{FFSSE}(\lambda)$. That is to say,

$$P[SSEReal_A^{FFSSE}(\lambda) = 1] = P[G_0 = 1].$$

Game G_1. Algorithm 2 formally describes G_1, and introduces an intermediate game $\overline{G}_1$, by including the additional boxed lines. In the **Update** protocol, instead of calling hash function $\mathbf{H}$ to generate the new id^*, we pick random strings when it is confronted to a new keyword w. For hash function $\mathbf{H}$, if an entry is accessed for the first time, it is first randomly chosen and then returned; if an entry has been accessed, then it will return the result which has returned before. Formally, for the first time, $id' \leftarrow \mathbf{H}(uk, r)$, id' is a random result; when access r for next time, it will still return the result id'. The point of $\overline{G}_1$ is to ensure the consistency of $\mathbf{H}'s$ transcript. In $\overline{G}_1$, hash function $\mathbf{H}$ will always program to two same results for the same input. If random number r doesn't appear in hash function $\mathbf{H}'s$ transcript, $\overline{G}_1$ randomly chooses id, otherwise, id will be set to the related value in the $\mathbf{H}'s$ transcript. Then, $\overline{G}_1$ lazily programs the RO when needed by the search protocol or by an adversary's query, so it makes sure the consistence of output. Because of it, in $\overline{G}_1$ and G_0, the output of hash function $\mathbf{H}$ is indistinguishable. Therefore,

$$P[G_0 = 1] = P[\overline{G}_1 = 1],$$

G_1 and $\overline{G}_1$ are also indistinguishable unless the flag is set to true, formally speaking,

$$P[\overline{G}_1 = 1] - P[G_1 = 1] \leq P[flag\ is\ set\ to\ true\ in\ \overline{G}_1].$$

The possibility of random number r gets the same random number is negligible. Furthermore, symmetric encryption $E_k(m)$ and decryption $D_k(c)$ are IND-CPA. Therefore,
$P[flag\ is\ set\ to\ true\ in\ \overline{G}_1] = Adv_H^{IND-CPA}(\lambda)$ is negligible.
Based on the above analysis,

$$\begin{aligned} P[G_0 = 1] - P[G_1 = 1] = P[\overline{G}_1 = 1] - P[G_1 = 1] \\ \leq Adv_H^{IND-CPA}(\lambda). \end{aligned}$$

Algorithm 2 *Game* G_1 **and** *Game* $\overline{G}_1$. **Boxed code is included in** $\overline{G}_1$ **only**

```
Setup()
 1: $uk \xleftarrow{\$} \{0,1\}^\lambda$
 2: $\mathbf{W} \leftarrow empty\ map,\ \mathcal{T} \leftarrow empty\ tree$
 3: $flag = false$
Search($w,\ \sigma$; EDB)
    Client:
 1: $(id_0, \ldots, id_n, cnt) \leftarrow \mathbf{W}[w]$
 2: $if (id_0, \ldots, id_n, cnt) = \perp$
 3:     $return\ \emptyset$
 4: $(ind_0 \| op_0, \ldots, ind_{cnt} \| op_{cnt}) \leftarrow \mathbf{Hist}(w)$
 5: $key \leftarrow \mathbf{W}[w].key$
 6: Send token $(id, key)$ to the server.
    Server:
 7: $\mathbf{S} \leftarrow empty\ set$
 8:     $\mathtt{H}(uk, r_i) \leftarrow \mathbf{W}[w].id$
 9: $for\ i = 0\ to\ cnt$
10:     $blk_i \leftarrow D_{key}(\mathcal{T}[id].data)$
11:     $id_i \leftarrow blk_i.ptr$
12:     $key_i \leftarrow blk_i.key$
13:     $ind \| op \leftarrow blk_i.value$
14:     $\mathbf{S}[i] \leftarrow ind \| op$
15:     $\mathtt{H}(uk, r_i) \leftarrow id_i$
16: $\mathbf{S} \leftarrow Merge(\mathbf{S})$
17: Send each $ind$ in $\mathbf{S}$ to the client.
Update($op,\ w,\ ind,\ \sigma$; EDB)
    Client:
 1: $id \leftarrow \mathbf{W}[w].id$
 2: $key \leftarrow \mathbf{W}[w].key$
 3: $r \xleftarrow{\$} \{0,1\}^\lambda$
 4: $id^* \leftarrow \{0,1\}^\mu$
 5: $if\ \mathtt{H}(uk, r) \neq \perp\ then$
 6:     $\boxed{flag \leftarrow true, id^* \leftarrow \mathbf{H}(uk, r)}$
 7: $end\ if$
 8: $key^* \leftarrow KeyGen(1^\lambda)$
 9: $newBlk \leftarrow (id^*, E_{key^*}(\text{“}ind \| op\text{”} \| key \| id))$
10: $\mathbf{W}[w].id \leftarrow id^*$
11: $\mathbf{W}[w].key \leftarrow key^*$
12: $\mathbf{W}[w].cnt{+}{+}$
13: Send $newBlk$ to the server.
    Server:
14: Insert $newBlk$ into the tree $\mathcal{T}$.
H($uk, r$)
 1: $t \leftarrow \mathtt{H}(uk, r)$
 2: $if\ t = \perp\ then$
 3:     $t \xleftarrow{\$} \{0,1\}^\lambda$
 4:     $if\ \exists id \in \mathbf{W}[w]$
 5:         $\boxed{flag \leftarrow true, t \leftarrow \mathbf{H}(uk, r)}$
 6:     $end\ if$
 7: $end\ if$
 8: $return\ t$
```

Game G_2. In Algorithm 3, we remove the code which has nothing to do with the hash function $\mathbf{H}$. Therefore, Algorithm 3 are single roundtrip protocols.

For each **update** operation, we output fresh random strings. For **search** operation, we get next id from the current block. For key, if an entry of $\mathbf{W}$ is accessed for the first time, G_2 will randomly pick it, otherwise, we will use the stored key. Therefore, we can conclude that

$$P[G_1 = 1] - P[G_2 = 1] = 0.$$

Algorithm 3 *Game G_2*

```
Setup()
1: $uk \xleftarrow{\$} \{0,1\}^\lambda$
2: $\mathbf{W} \leftarrow$ empty map, $\mathcal{T} \leftarrow$ empty tree
3: $u \leftarrow 0$
4: Updates $\leftarrow$ empty map
Search($w$, $\sigma$; EDB)
    Client:
1: $id_0 \leftarrow \mathbf{W}[w]$
2: $key \leftarrow \mathbf{W}[w].key$
3: $[(u_0, ind_0), \ldots, (u_{cnt}, ind_{cnt})] \leftarrow$ Updates$[w]$
4: if $cnt = 0$ then
5:     return $\emptyset$
6: end if
7: for $i = 0$ to $cnt$ do
8:     Program $\mathbf{H}$ s.t. $\mathbf{H}(uk, r_i) \leftarrow id_i$
9:     $id_{i+1} \leftarrow blk_i.ptr$
10: end for
11: Send token $(id, key)$ to the server.
Update($op$, $w$, $ind$, $\sigma$; EDB)
    Client:
1: Append $(u, ind)$ to Updates$[w]$
2: $id \leftarrow \mathbf{W}[w].id$
3: $key \leftarrow \mathbf{W}[w].key$
4: $r \xleftarrow{\$} \{0,1\}^\lambda$
5: $id^* \xleftarrow{\$} \{0,1\}^\mu$
6: $key^* \leftarrow KeyGen(1^\lambda)$
7: $newBlk \leftarrow (id^*, E_{key^*}(\text{"}ind\|op\text{"}\|key\|id))$
8: $\mathbf{W}[w].id \leftarrow id^*$
9: $\mathbf{W}[w].key \leftarrow key^*$
10: $\mathbf{W}[w].cnt{+}{+}$
11: Send $newBlk$ to the server.
12: $u = u + 1$
```

The Simulator. we can divide the code of G_2 into two parts, one is the leakage function and the other is the simulator. Algorithm 4 describes the simulator. $SSEIdeal^{FFSSE}_{A,S,\mathcal{L}_S}$ and G_2 are identical. Therefore,

$$P[G_2 = 1] - P[SSEIdeal^{FFSSE}_{A,S,\mathcal{L}_S}(\lambda) = 1] = 0.$$

Conclusion. Combine the contributions which come from G_0, G_1, G_2 and S, we can conclude that $P[SSEReal^{FFSSE}_{A}(\lambda) = 1] - P[SSEIdeal^{FFSSE}_{A,S,\mathcal{L}_S}(\lambda) = 1] \leq Adv^{IND-CPA}_{H}(\lambda)$,

by stating that $\mathbf{H}$ is a hash function, symmetric encryption $E_k(m)$ and decryption $D_k(c)$ are IND-CPA. □

4.4 Comparison with $\Sigma o\phi o\varsigma$

In this section, we briefly discuss the comparison between FFSSE and $\Sigma o\phi o\varsigma$-B whose client storage is not reduced.

Computational performance. For the search and update operation in the server, both the FFSSE and $\Sigma o\phi o\varsigma$-B have the same computational complexities: $O(a_w)$ for search, $O(1)$ for update and token generation. But from a practical point of view, FFSSE has a better performance than $\Sigma o\phi o\varsigma$-B, because their primitives are different: symmetric encryption scheme is in FFSSE while public key encryption scheme is in $\Sigma o\phi o\varsigma$-B.

Support of add and delete operations. In FFSSE, we can use a unique instance to support both add and delete operations by storing the document identifier and operators together. However, $\Sigma o\phi o\varsigma$-B utilizes two instances for add and delete respectively. So that, FFSSE is more flexible for dynamic merging or other operations.

Algorithm 4 *Simulator S*

```
Setup()
 1: $uk \xleftarrow{\$} \{0,1\}^{\lambda}$
 2: $\mathbf{W} \leftarrow empty\ map$, $\mathcal{T} \leftarrow empty\ tree$
 3: $u \leftarrow 0$
Search($w$, $\sigma$; EDB)
    Client:
 1: $id \leftarrow \mathbf{W}[w].id$
 2: $key \leftarrow \mathbf{W}[w].key$
 3: if $cnt = 0$ then
 4:     return $\emptyset$
 5: end if
 6: Send token $(id, key)$ to the server.
Update($op$, $w$, $ind$, $\sigma$; EDB)
    Client:
 1: $id \leftarrow \mathbf{W}[w].id$
 2: $key \leftarrow \mathbf{W}[w].key$
 3: $r \xleftarrow{\$} \{0,1\}^{\lambda}$
 4: $id^* \xleftarrow{\$} \{0,1\}^{\mu}$
 5: $key^* \leftarrow KeyGen(1^{\lambda})$
 6: $newBlk \leftarrow (id^*, E_{key^*}(``ind\|op" \|key\|id))$
 7: $\mathbf{W}[w].id \leftarrow id^*$
 8: $\mathbf{W}[w].key \leftarrow key^*$
 9: $\mathbf{W}[w].cnt{+}{+}$
10: Send $newBlk$ to the server.
11: $u = u + 1$
```

5 An improved construction

To further reduce the client storage and improve the performances of search operation, we introduce an improved FFSSE scheme, called "FFSSE-ε".

5.1 Intuitions

Reducing client storage. In $\Sigma o\phi o\varsigma$-B, the client storage is $O(\mathcal{W}(\log|\mathcal{M}| + \log \mathcal{D}))$, where $\mathcal{D}$ is number of documents and $\mathcal{M}$ is the length of search token. When $\mathcal{M}$ is big, which is a 2048 bits integer for a reasonable level of security, this can be a problem on constrained devices. Thus, $\Sigma o\phi o\varsigma$ scheme reduces the client storage to $O(\mathcal{W} \log \mathcal{D})$ by using a re-computation method based on trapdoor permutation (TDP) π. In FFSSE, the client storage is $O(\mathcal{W}(\mu + \lambda))$, that is, for each keyword w, there are μ-bit data identifier and λ-bit encryption key in its client state. This can also be a problem on constrained devices. We can reduce the client state to $O(\mathcal{W} \log \mathcal{D})$ by preserving only the counter, like that in $\Sigma o\phi o\varsigma$. For the data identifier and encryption key, we can re-compute them instead of storing them.

Challenge. In order to remove the search token (*id* and *key* of *head block*) for storage reduction, we must re-generate them. However, this will lead to the contradiction with *index generation rule independence* of FFSSE. Even if we have to provide re-generation rule, we can't use TDP technique, because of its performance problem that we try to solve in this paper. It comes to the challenge then: *how can we design a similar TDP technique with both the efficient performance and the ability of token re-generation*?

Improving search performance. For a keyword w, there are total a_w nodes storing the document identifier and its operator in the tree $\mathcal{T}$. So, a **search** protocol contains total a_w tree traverse operations and symmetric decryption operations. The traverse of the tree is also an expensive operation. Aiming at solving this issue, we can dynamically merge all the nodes matching w into a small number of nodes. Because the search operation in most SSE schemes allows the leakage of search results, the merge operation will not leak other information.

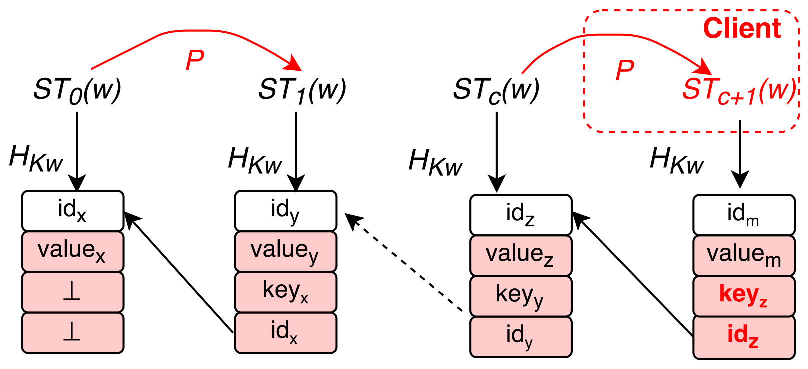

Figure 4: The ε-sTDP. A one-way permutation (red) is performed in the client, while its inverse is simulated by the key-based blocks chain. It is similar to the usage of TDP, so that we call it as "an efficient permutation similar to TDP".

5.2 The ε-sTDP technique

Basic idea. In order to obtain a good performance and forward security, we still apply the KBBC to link the blocks and hide their relations. But for token re-generation, because the security requires the output domain to be the same as the input domain in the random oracle (in Theorem 2), we utilize a one-way permutation $\mathcal{P}$ (not necessary to have a trapdoor) instead of TDP to recompute the tokens. Obviously, this is not a strict trapdoor permutation, but it is very similar to the usage of TDP in $\Sigma o\phi o\varsigma$. That is why we call it as "ε-sTDP", which means "an efficient permutation similar to TDP".

Considering the performance, we suggest to instantiate the permutation $\mathcal{P}$ by using symmetric cryptographic primitive (such as block cipher). For convenience, we define the one-way permutation $\mathcal{P}$ as $\mathcal{P} : \{0,1\}^\mu \rightarrow \{0,1\}^\mu$, and denote $\mathcal{P}_{k_\rho}(x)$ as a keyed permutation with the key k_ρ and message x as inputs. Thus, for each $x \overset{\$}{\leftarrow} \{0,1\}^\mu$, we have $|\mathcal{P}_{k_\rho}(x)| = |x|$.

The ε-sTDP. Figure 4 shows the details of ε-sTDP. For each keyword w, the client randomly generates a $ST_0(w)$ and stores a counter c with default value 0. When adding a data block, the client will first compute its identifier by: $ST_{c+1}(w) \leftarrow \mathcal{P}_{k_\rho}(ST_c(w))$; $id \leftarrow H_{K_w}(ST_{c+1}(w))$. Then, based on KBBC, the client sets the data identifier and encryption key of the next block and encrypts them. The search token is still the (id, key) of the *head block*, which is re-computed by the client.

5.3 Our construction

Algorithm 5 gives the formal description of our improved forward private scheme, "FFSSE-ε". We still use $\mathbf{W}[w]$ to store the keyword state, but there is only a counter *cnt* (default value is -1) after client storage reduction. We define a constant value CT for dynamic adjustment.

Re-generation of search token. For re-computing the search token, we first pseudo-randomly generate $ST_0(w)$ from w (or a unique identifier $i_w \leq \mathcal{W}$). Like that in $\Sigma o\phi o\varsigma$, it is easy to do from a PRF $\mathcal{G} : \mathcal{W} \rightarrow \{0,1\}^\mu$ by computing $ST_0(w) \leftarrow \mathcal{G}(i_w)$.

Then, to recompute the search token with the counter c, we can easily compute $ST_c(w)$ from $ST_0(w)$ and c by iterating $\mathcal{P}$ for c times, that is, $ST_c(w) \leftarrow \mathcal{P}^c_{k_\rho}(ST_0(w))$. After obtaining the $ST_c(w)$, we can generate the *id* of search token as $\mathbf{H}_{K_w}(ST_c(w))$, where K_w is computed by the user's secret key *uk* and w. For the other part *key* of search token, we can generate hash key K^*_w by user's secret key uk^* and utilize it to compute the *key* as $\mathbf{H}_{K^*_w}(ST_c(w))$. As a result, the search token is $(\mathbf{H}_{K_w}(ST_c(w)), \mathbf{H}_{K^*_w}(ST_c(w)))$. We denote this re-generation process as $\mathbf{ReGen}(w, c)$ which takes w and c as inputs.

Dynamic adjustment. After a certain amount CT of operations on keyword w, the client can require the server to delete all the nodes matching w. Then, it dynamically merges them into one (or more) nodes according to some rules pre-defined in FFSSE-ε. Finally, it uploads the merged nodes to the server along with a **search** operation, without loss of the security for the usage of "key-based blocks chain". In the

Algorithm 5 FFSSE-ε: Forward private SSE scheme supporting dynamic adjustment

```
Setup()
 1: $uk \xleftarrow{\$} \{0,1\}^\lambda,\ uk^* \xleftarrow{\$} \{0,1\}^\lambda$
 2: $k_\rho \leftarrow \text{KeyGen}(1^\lambda)$
 3: $\mathbf{W} \leftarrow$ empty map, $\mathcal{T} \leftarrow$ empty tree
Search($w,\ \sigma$; EDB)
    Client:
 1: $(id, key) \leftarrow \mathbf{ReGen}(w, \mathbf{W}[w].cnt)$
 2: Send token $(id, key)$ to the server.
    Server:
 3: $\mathbf{S} \leftarrow$ empty set, $i \leftarrow 0$
 4: repeat
 5:    $b \leftarrow D_{key}(\mathcal{T}[id].data)$
 6:    $id \leftarrow b.ptr$
 7:    $key \leftarrow b.key$
 8:    $ind\|op \leftarrow b.value$
 9:    $\mathbf{S}[i{+}{+}] \leftarrow ind\|op$
10: until $(b.ptr == \perp)$
11: $\mathbf{S} \leftarrow Merge(\mathbf{S})$
    // merge the same ind with add and del operations
12: Send each ind in $\mathbf{S}$ to the client.
    Client:
13: if $\mathbf{W}[w].cnt \bmod CT == 0$ then
14:    $DynamicAdjust(w)$
15: end if
Update($op,\ w,\ ind,\ \sigma$; EDB)
    Client:
 1: $(id, key) \leftarrow \mathbf{ReGen}(w, \mathbf{W}[w].cnt)$
 2: $\mathbf{W}[w].cnt{+}{+}$
 3: $(id^*, key^*) \leftarrow \mathbf{ReGen}(w, \mathbf{W}[w].cnt)$
 4: $b \leftarrow (id^*, E_{key^*}(\text{“}ind\|op\text{”}\|key\|id))$
 5: Send block $b$ to the server.
    Server:
 6: Insert block $b$ into the tree $\mathcal{T}$.
ReGen($w,\ c$)
 1: if $\mathbf{W}[w].cnt$==-1 then
 2:    $id \leftarrow \perp,\ key \leftarrow \perp$
 3: else
 4:    $K_w \leftarrow \mathcal{F}(uk, w)$
 5:    $K_w^* \leftarrow \mathcal{F}(uk^*, w)$
 6:    $ST_0(w) \leftarrow \mathcal{G}(i_w)$
 7:    if $\mathbf{W}[w].cnt$==0 then
 8:       $id \leftarrow \mathbf{H}(K_w, ST_0(w)),\ key \leftarrow \mathbf{H}(K_w^*, ST_0(w))$
 9:    else
10:       $ST_c(w) \leftarrow \mathcal{P}_{k_\rho}^c(ST_0(w))$
11:       $id \leftarrow \mathbf{H}(K_w, ST_c(w)),\ key \leftarrow \mathbf{H}(K_w^*, ST_c(w))$
12:    end if
13: end if
14: Return the id and key.
```

algorithm 5, we only represent the above client-server protocol as the function *DynamicAdjust* with the input w.

Obviously, based on dynamic adjustment, the FFSSE-ε can freely eliminate the same document identifier with both *add* and *delete* operators in different nodes, reduce the total number of nodes in the tree $\mathcal{T}$ from $O(\mathcal{W}a_w)$ to $O(\mathcal{W})$, and improve the search performance by reducing about a_w traverse operations of the tree.

Theorem 3. *FFSSE-ε is $\mathcal{L}_S - adaptive - secure$.*

5.4 Comparison with $\Sigma o\phi o\varsigma$

In this section, we briefly discuss the comparison between FFSSE-ε and $\Sigma o\phi o\varsigma$, whose client storages are both reduced to $O(\mathcal{W}\log\mathcal{D})$.

Dynamic adjustment. This character is used in this paper, to emphasize that the SSE scheme can adjust the elements stored in the cloud server for the consideration of efficiency. It is obvious that FFSSE-ε supports dynamic adjustment for search performance, while $\Sigma o\phi o\varsigma$ does not.

Computational performance. Although the same computational complexities, $O(a_w)$ for search and $O(1)$ for update, FFSSE-ε has a better performance than $\Sigma o\phi o\varsigma$ caused by the adopted symmetric cryptographic primitive and dynamic adjustment. As for token generation with complexity of $O(a_w)$, FFSSE-ε is obviously better than $\Sigma o\phi o\varsigma$ caused by the ε-sTDP mechanism.

Functionalities and leakages. Compare to $\Sigma o\phi o\varsigma$, FFSSE-ε utilizes ε-sTDP instead of traditional TDP to hide the connection between the newly added nodes and the existing nodes. However, the **search** and **update** protocols of these two schemes still have the same processes. They both store the keyword states in the client and send the search or update token to the server. And thus, they actually have the same functionalities and leakages.

6 Experiments and evaluations

The goal of our experiments is to compare the performance between FFSSEs (FFSSE and FFSSE-ε) and the first efficient forward private SSE scheme $\Sigma o\phi o\varsigma$. Generally, a complete search/update operation consists of search/update in server side and token generation in client side. Considering token generation has its own rule in different scheme, we separate a complete operation into two parts, that in server and that in client.

6.1 Implementation details

We implemented FFSSEs' core functions and benchmarks in C/C++, using about 1000 lines of codes. All of the cryptographic primitives implementation in FFSSEs use the third party's code provided by $\Sigma o\phi o\varsigma$ [21]'s open source code. For keyed hash function, we use HMAC. The keyed permutation $\mathcal{P}$ and symmetric encryption are instantiated using AES in counter mode. For the FFSSE-ε, we set CT as 75.

Experiment environments. To provide the same experimental environments, we use the same keyword-document pairs generation rules that $\Sigma o\phi o\varsigma$ does. Also, we use RocksDB as underlying server side's storage to store tree $\mathcal{T}$ since map $\mathbf{T}$ in $\Sigma o\phi o\varsigma$ is stored using this database. To note that, aside from the chosen for pairs generation rules and cryptographic components, we drop RPC machinery in implementation and don't take timings of underlying database operations into account. These considerations attempt to guarantee the accurate measurements and comparison of search, update and token generation operations for both our schemes and $\Sigma o\phi o\varsigma$'s.

We run our experiments on a desktop computer with a single Inter Core i7-7700 3.60HZ CPU, 2GB of RAM running on ubuntu 14.0.4. Our codes have been opened in GitHub: *https://github.com/liuzheli/FFSE.*

Parameter. For our schemes, we set λ, the length of symmetric keys, to 256 bits. The N_{max}, the maximum number of keyword-document pairs, is determined by concrete benchmarks ranging from 140 to 14000000. The length of identifiers, μ, can be set from λ and N_{max} according to the above-mentioned descriptions.

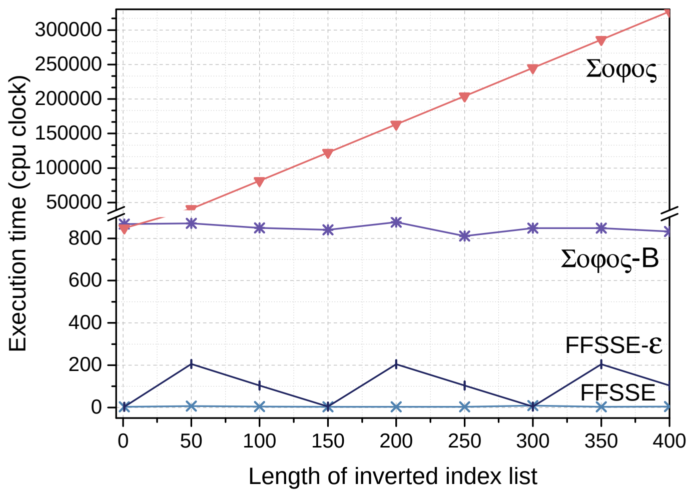

Figure 5: Comparison of token generation in update operation. X-axis is the number of times that keyword w has been queried, i.e., a_w.

6.2 Evaluation

Token generation. We evaluated the performances of token generation algorithms in update operation. Figure 5 shows the experiment results. We can conclude that our token generation algorithms have significantly better performance than $\Sigma o\phi o\varsigma$s'. For two basic schemes, FFSSE is at least 300× faster than $\Sigma o\phi o\varsigma$-B, because $\Sigma o\phi o\varsigma$-B requires RSA-based private key encryption operation but FFSSE only uses symmetric encryption operations. For two improved schemes with the same size of client storage, FFSSE-ε is at least 200× faster than $\Sigma o\phi o\varsigma$, since it is based on symmetric encryption scheme. Moreover, as the number of documents in the indexed list increases, the execution time of FFSSE-ε does not increase, owing to the dynamic adjustment (CT=75). Specially, we can see that the token generation in the client is very expensive for $\Sigma o\phi o\varsigma$ when the length of indexed list is long.

SSE operations. We evaluated execution times of completed search and update operations to compare performances of different schemes. We tested each scheme on the basis of the same keyword-document pairs, the same keyword sets and the same benchmarks. Concretely, we first run the update operation benchmarks and construct the databases at the same time. Then run the search operation benchmarks on them. Actually, our benchmarks are the same as those in $\Sigma o\phi o\varsigma$. The experiment results can be shown in Figure 6 and Figure 7. We can conclude that no matter search or update operation, FFSSEs have significantly better performance than $\Sigma o\phi o\varsigma$s, derived from their different cryptographic primitives.

About search operation, FFSSE is average 4× better than $\Sigma o\phi o\varsigma$-B. Despite our search algorithm only contains symmetric key encryption, the reason why our advantage is not obvious is due to the facts: there is no token generation operation, and RSA public key-based evaluation in $\Sigma o\phi o\varsigma$-B is only 5× slower than symmetric encryption in FFSSE. But for two improved schemes, FFSSE-ε is average 60× better than $\Sigma o\phi o\varsigma$. After reducing client storage, token generation operation must be executed in the client. Because $\Sigma o\phi o\varsigma$ uses RSA private key-based token generation, it is much slower than that in FFSSE-ε.

About update operation, the differences in their performances stem mainly from the token generation and block encryption. Two basic schemes require less computation about token generation, but two improved schemes must re-generate them for client storage reduction. We can see that FFSSE is average 300× better than $\Sigma o\phi o\varsigma$-B, FFSSE-ε is average 200× better than $\Sigma o\phi o\varsigma$.

Distribution of search operation. We also evaluated the execution time in client and that in server of the search operation. Figure 8 shows the experiment results. Obviously, because of no need to re-

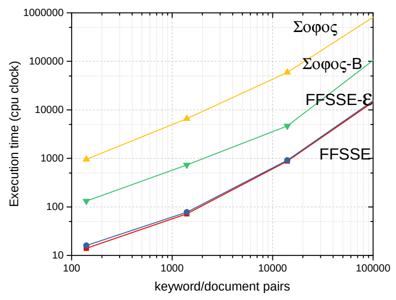

Figure 6: Comparison of search operation. Average execution time of search operation over database with 140000 keyword-document pairs.

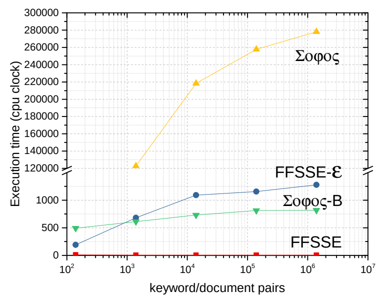

Figure 7: Comparison of update operation. Average execution time of update operation over database with 140000 keyword-document pairs.

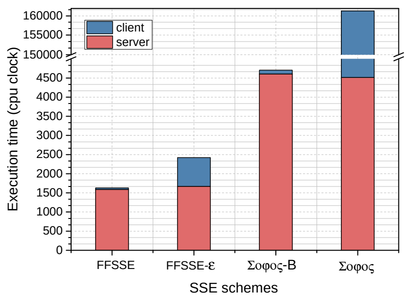

Figure 8: Execution time distribution of search operation. Search time ($|\mathbb{L}_w| = 130$) over database with 140000 keyword-document pairs.

generate tokens in the client, two basic schemes take more than 95% of the time to perform the server-side operations. For FFSSE-ε, the client-side operations take less time than server-side operations, because dynamic adjustment of FFSSE-ε reduces the number of encryption in token generation process. But for $\Sigma o\phi o\varsigma$, it takes lots of time in the client for the RSA private key-based token generation.

7 Conclusion

We focus on how to improve the performance of forward secure searchable encryption schemes. Until now, the state-of-the-art forward private SSE scheme is $\Sigma o\phi o\varsigma$ proposed in CCS 2016, however, it relies on the asymmetric cryptographic primitive. In this paper, we exploit a key-based blocks chain technique which is based on symmetric cryptographic primitive, and apply it to design a more efficient forward secure SSE scheme, named FFSSE. The proposed key-based blocks chain is independent of index generation rule, so that it is more flexible than TDP in $\Sigma o\phi o\varsigma$. It may be valuable for some other designs with the consideration of forward security. In our future work, we plan to study how to design the forward secure order-preserving encryption scheme based on key-based blocks chain technique.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 61672300), and National Natural Science Foundation of Tianjin (No. 16JCYBJC15500).

References

[1] Z. Fu, X. Wu, Q. Wang, K. Ren, "Enabling Central Keyword-Based Semantic Extension Search Over Encrypted Outsourced Data", in *IEEE Transactions on Information Forensics and Security*, 12(12), pp. 2986-2997, 2017.

[2] Z. Fu, X. Wu, C. Guan, X. Sun, K. Ren, "Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement", in *IEEE Transactions on Information Forensics and Security*, 11(12), pp. 2706-2716, 2016.

[3] P. Xu, Q. Wu, W. Wang, W. Susilo, J. Domingo-Ferrer, H. Jin, "Generating searchable public-key ciphertexts with hidden structures for fast keyword search", in *IEEE Transactions on Information Forensics and Security*, 10(9), pp. 1993-2006, 2015.

[4] R. A. Popa, C. Redfield, N. Zeldovich, H. Balakrishnan, “CryptDB: protecting confidentiality with encrypted query processing”, in *SOSP*, pp. 85-100, 2011.

[5] R. Poddar, T. Boelter, R. A. Popa, “Arx: A Strongly Encrypted Database System”, in *IACR Cryptology ePrint Archive*, 2016.

[6] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, “Rich queries on encrypted data: Beyond exact matches”, in *ESORICS*, 2015.

[7] Z. Fu, F. Huang, K. Ren, J. Weng, C. Wang, “Privacy-Preserving Smart Semantic Search Based on Conceptual Graphs Over Encrypted Outsourced Data”, in *IEEE Transactions on Information Forensics and Security*, 12(8), pp. 1874-1884, 2017.

[8] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. C. Rosu, M. Steiner, “Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation”, in *NDSS*, Vol. 14, pp. 23-26, 2014.

[9] D. X. Song, D. Wagner, A. Perrig, “Practical techniques for searches on encrypted data”. In *S&P*, 2000.

[10] R. Li, A. X. Liu, “Adaptively secure conjunctive query processing over encrypted data for cloud computing”, in *ICDE*, pp. 697-708, 2017.

[11] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, M. Garofalakis, “Practical private range search revisited”, in *SIGMOD*, pp. 185-198, 2016.

[12] E. Stefanov, C. Papamanthou, E. Shi, “Practical Dynamic Searchable Encryption with Small Leakage”, in *NDSS*, pp. 23-26, 2014.

[13] R. Curtmola, J. Garay, S. Kamara, R. Ostrovsky, “Searchable symmetric encryption: improved definitions and efficient constructions”, in *CCS*, 2006.

[14] M. S. Islam, M. Kuzu, M. Kantarcioglu, “Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation”, in *NDSS*, 2012.

[15] D. Cash, P. Grubbs, J. Perry, T. Ristenpart, “Leakage-abuse attacks against searchable encryption”, in *CCS*, pp. 668-679, 2015.

[16] Y. Zhang, J. Katz, C. Papamanthou, “All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption”, in *USENIX Security*, 2016.

[17] G. Kellaris, G. Kollios, K. Nissim, A. O’Neill, “Generic attacks on secure outsourced databases”, in *CCS*, pp. 1329-1340, 2016.

[18] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, “Constants Count: Practical Improvements to Oblivious RAM”, in *USENIX Security*, 2015.

[19] X. Wang, H. Chan, E. Shi, “Circuit oram: On tightness of the goldreich-ostrovsky lower bound”, in *CCS*, pp. 850-861, 2015.

[20] Y. C. Chang, M. Mitzenmacher, “Privacy preserving keyword searches on remote encrypted data”, in *ACNS*, pp. 442-455, 2005.

[21] R. Bost, “$\Sigma o\phi o\varsigma$−Forward Secure Searchable Encryption”, in *CCS*, 2016.

[22] S. Garg, P. Mohassel, C. Papamanthou, “TWORAM: Round-optimal oblivious RAM with applications to searchable encryption”, in *Crypto*, 2016.

[23] R. W. Lai, S. S. Chow, “Forward-Secure Searchable Encryption on Labeled Bipartite Graphs”, in *ACNS*, pp. 478-497, 2017.

[24] S. Kamara, C. Papamanthou, T. Roeder, “Dynamic searchable symmetric encryption”, in *CCS*, pp. 965-976, 2016.

[25] K. Kurosawa, Y. Ohtaki, “UC-Secure Searchable Symmetric Encryption”, in *Financial Cryptography*, Vol. 7397, pp. 285-298, 2012.

[26] M. Naveed, M. Prabhakaran, C. A. Gunter, “Dynamic searchable encryption via blind storage”, in *S&P*, pp. 639-654, 2014.

[27] A. Lysyanskaya, S. Micali, L. Reyzin, H. Shacham, “Sequential aggregate signatures from trapdoor permutations”, in *Eurocrypt*, Vol. 3027, pp. 74-90, 2004.

[28] Q. Wang, M. He, M. Du, S. S. Chow, R. W. Lai, Q. Zou, “Searchable encryption over feature-rich data”, in *IEEE Transactions on Dependable and Secure Computing*, 2016.

[29] Y. Yang, X. Liu, R. H. Deng, Y. Li, “Lightweight Sharable and Traceable Secure Mobile Health System”, in *IEEE Transactions on Dependable and Secure Computing*, 2017.

[30] K. Liang, X. Huang, F. Guo, J. K. Liu, “Privacy-preserving and regular language search over encrypted cloud data”, in *IEEE Transactions on Information Forensics and Security*, 11(10), pp. 2365-2376, 2016.