Forward Secure Dynamic Searchable Symmetric Encryption with Efficient Updates

Kee Sung Kim
National Security Research Institute
keesung2137@nsr.re.kr

Minkyu Kim
National Security Research Institute
mkkim@nsr.re.kr

Dongsoo Lee
National Security Research Institute
letrhee@nsr.re.kr

Je Hong Park
National Security Research Institute
jhpark@nsr.re.kr

Woo-Hwan Kim
National Security Research Institute
whkim5@nsr.re.kr

ABSTRACT

The recently proposed file-injection type attacks are highlighting the importance of forward security in dynamic searchable symmetric encryption (DSSE). Forward security enables to thwart those attacks by hiding the information about the newly added files matching a previous search query. However, there are still only a few DSSE schemes that provide forward security, and they have factors that hinder efficiency. In particular, all of these schemes do not support actual data deletion, which increments both storage space and computational complexity. In this paper, we design and implement a forward secure DSSE scheme with optimal search and update complexity, for both computation and communication point of view. As a starting point, we propose a new, simple, theoretical data structure, called *dual dictionary* that can take advantage of both the inverted and the forward indexes at the same time. This data structure allows to delete data explicitly and in real time, which greatly improves efficiency compared to previous works. In addition, our scheme provides forward security by encrypting the newly inserted data with fresh keys not related with the previous search tokens. We implemented our scheme for Enron email and Wikipedia datasets and measured its performance. The comparison with Sophos shows that our scheme is very efficient in practice, for both searches and updates in dynamic environments.

CCS CONCEPTS

•Security and privacy → Management and querying of encrypted data;

KEYWORDS

Dynamic Searchable Symmetric Encryption, Forward Security

CCS'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.

ISBN 978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/10.1145/3133956.3133970

1 INTRODUCTION

With the advent of cloud computing, a number of cloud service providers have arisen to provide a digital storage on their own infrastructure. The basic approach for protecting the confidentiality of data stored in the untrusted cloud storage is to encrypt data using general symmetric encryption. While this approach provides strong security protection, semantic security precludes any ability to perform useful operations on encrypted data except decryption. It thereby induces inefficiency to provide functionality and is therefore inadequate for storage systems that handle large amounts of data. As a partial but practical solution to this problem, the notion of searchable symmetric encryption (SSE) which provides a practical search function on encrypted data, was introduced [30].

SSE enables a client to perform efficient keyword searches on the encrypted documents while preserving the privacy of both the database and the queries. A common tool to speed up the search process is an *index* which is a pre-built data structure made from documents, and there are two high-level approaches to designing an index for reasonably efficient and secure SSE schemes [3]. One is the *forward index*, which makes keyword lists per document, and it naturally requires search time proportional to the number of documents. The other is the *inverted index*, which maintains lists of document identifiers per keyword and so it achieves sub-linear search time $O(n_w)$, where n_w is the number of documents containing the keyword w in the database. Because the efficiency of search query processing is generally the most important factor in determining availability of SSE schemes, the inverted index has been preferred.

Although sub-linear search time is one of main requirements for practical SSE schemes, scalability guaranteeing efficient update (addition and deletion) of documents is also a required property of SSE schemes. But the *static* SSE schemes [1, 8, 9, 11, 13], which only consider a fixed number of document/keyword pairs, can provide scalability by either re-indexing the entire documents or making use of generic and relatively expensive techniques [13]. To remedy this problem, dynamic variants of SSE (DSSE) schemes [7, 23, 24, 27, 28, 31, 32] have been proposed.

With tradeoffs between security and practicality, almost all of the practical SSE schemes leak information about documents. Recent research on the real-world impact of these leakage [6, 22, 33], however, shows that even small leakage can be used to break the privacy of search queries. In particular, the file-injection attacks proposed by Zhang *et al.* [33] shows that it is possible to reveal the contents of past search queries of DSSE schemes with a few

injection of documents. This attack underlines the need for DSSE schemes with forward security.

1.1 Previous Forward Secure DSSE Schemes

Forward security of DSSE schemes means that the adversary cannot learn any information about added document containing the keyword that was previously queried. It guarantees that a search token for a keyword w cannot be used to retrieve documents added after issuing the token.

The object of file-injection attacks [33] is to learn a very high fraction of information about keywords from search tokens *issued by the client*. Once files sophisticatedly composed of keywords chosen from the keyword universe are injected to the encrypted database (EDB), the keywords corresponding to any future search tokens sent by the client could be revealed. However, forward security makes it possible to prevent determining the keywords issued before injecting malicious files, and so to thwart this type of attack.

Chang and Mitzenmacher [10] proposed the first forward secure DSSE scheme, but there was no solution with sub-linear search time before the work of Stefanov *et al.* [31]. Thereafter, a series of new forward secure DSSE schemes [4, 32] have been proposed, achieving better performance and/or security. While being asymptotically better, those schemes also have several drawbacks.

In the scheme proposed by Stefanov *et al.* [31], during their *interactive* update process, the client fetches a non-negligible amount of data from the server and requires $O(N^{\alpha})_{0<\alpha<1}$ working storage to run the oblivious sort algorithm. Furthermore, deletion of data is logically done in the form that it adds deletion information to the EDB and reflects them in the search process. The explicit deletion can be performed via entire EDB rebuilding. The author of [31] suggested such rebuilding when deletion information occupied about half of the EDB.

The scheme proposed by Yavuz and Guajardo [32] uses a simple matrix-based data structure of size $O(m \times n)$ where rows and columns denote keywords and document identifiers respectively. In their scheme, two hash tables are stored at the client-side and one of them has to be shared with the server in order to synchronize some states in real time. These hash tables are intended to represent an inverted index and a forward index respectively, and the matrix structure linking these two hash tables is introduced to provide search operations as the inverted index approach and update operations as the forward index approach. Note that, however, this data structure is required to determine in advance the parameters m and n representing respectively the maximum number of allowable keywords and documents. Therefore, their scheme can be seen as being closer to the static SSE.

Sophos [4] ensures forward security through the relationship between the search tokens and the update tokens based on trapdoor permutations. This relationship provides better theoretical efficiency in terms of computation and communication compared to the previous schemes. However, the trapdoor permutations based on public key operations such as RSA exponentiation degrade computational efficiency. Moreover, since the search and update operations involve iterative applications of the trapdoor permutation, Sophos does not support parallel processing of these operations. In addition, the data structure used in Sophos does not support actual deletion because it is specialized only for the search and add operations. Therefore, the dynamic nature of Sophos is provided by running two instances, one for addition and the other for deletion. During a search query processing for a keyword w, the server computes and returns the difference between the document identifiers corresponding to w in both instances. These factors degrade availability in terms of server storage and computation for the search operation.

Although there has been made a lot of progress in efficiency of ORAM techniques, the recent DSSE schemes based on ORAM such as [19] still require large bandwidth and multiple rounds. Note that the most attractive feature of the ORAM-based schemes is able to hide the access pattern (i.e., the identifiers of the documents matching search and update queries). However, this can only be satisfied by the assumption that documents are also stored in ORAM, and it results in more severe performance degradation [26].

1.2 Our Scheme

As mentioned above, most of the DSSE schemes use an inverted index-based data structure to guarantee efficient sub-linear search time. However, due to the structural difficulties in updating documents, especially deletion, rather complex and unnatural ways have been taken into account, resulting in overall degradation of efficiency. Some of the DSSE schemes [23, 28] adopted a forward index-based tree data structure to circumvent updating problem of the inverted index directly. But these schemes provide sub-linear search time depending on the total number of documents, not the number of documents matching the query.

In order to take all the advantages of both approaches, we propose a new data structure, called *dual dictionary*, consists of linked dictionaries to represent both inverted and forward indexes. To see how this works, let us consider a document containing keywords $w_1, \ldots, w_t$. We first generate t labels as $\mathsf{label}_i^{(1)} = H(\mathsf{key}^{(\mathsf{ind})}, i)$ for $i = 1, \ldots, t$, where $\mathsf{key}^{(\mathsf{ind})} \in \{0,1\}^{\lambda}$ is a secret key corresponding to the document identifier ind. We then generate another t labels as $\mathsf{label}_i^{(2)} = H(\mathsf{key}^{(w_i)}, \mathsf{cnt}^{(w_i)} + 1)$ for $i = 1, \ldots, t$, where $\mathsf{key}^{(w_i)} \in \{0,1\}^{\lambda}$ is a secret key corresponding to the keyword $w_i \in \mathsf{W}$, and $\mathsf{cnt}^{(w_i)}$ denotes the number of documents containing w_i in the EDB before adding the document. We then store $(\mathsf{label}_i^{(1)}, \mathsf{label}_i^{(2)})$ in a dictionary Dic_1 which reflects a forward index, and store $(\mathsf{label}_i^{(2)}, (\mathsf{label}_i^{(1)}, \mathsf{ind}))$ in another dictionary Dic_2 which reflects an inverted index. To retrieve documents containing a keyword w, we compute $\mathsf{label}_j^{(2)} = H(\mathsf{key}^{(w)}, j)$ for $1 \le j \le \mathsf{cnt}^{(w)}$ and search Dic_2. To delete a document with the identifier ind, we first compute $\mathsf{label}_i^{(1)} = H(\mathsf{key}^{(\mathsf{ind})}, i)$ for $i \ge 1$, and then find $(\mathsf{label}_i^{(1)}, \mathsf{label}^{(2)})$ in Dic_1 and $(\mathsf{label}^{(2)}, (\mathsf{label}_i^{(1)}, \mathsf{ind}))$ in Dic_2, and delete them.

Though the idea of simultaneous usage of both inverted and forward indexes was proposed by Hahn and Kerschbaum [21], the design of data structures for EDB management based on both indexes is totally different. Basically their scheme uses a forward index for EDB management and creates an inverted index for a keyword only after the keyword is searched. This makes it possible

to respond efficiently when the same keyword is searched again. Here, a key value of the inverted index made from a keyword is used as a search token and so newly added documents which containing the corresponding keyword are appended with respect to the search token. It means that their scheme cannot guarantee forward security because which keywords have been queried before is shared with the server when new document is added.

Although the dual dictionary requires about twice the amount of storage and computation, it guarantees sub-linear search time and explicit data deletion simultaneously. As shown in Table 1, our DSSE scheme provides optimal complexity in every point of view. Compared to Sophos, our scheme has optimal server storage complexity because it no longer owns the *deleted* data after processing the deletion query. This is a property that is required in an environment where update operations are likely to occur. The search time is also relatively efficient compared to the previous schemes since there is no need to perform additional searches on the deleted data. This is a very meaningful result, considering that search time is one of the most important measures of the practicality of SSE schemes. It also maximizes efficiency by handling most operations with hash functions only, and is designed to ensure the same level of security compared to previous schemes. Table 2 summarizes the number of occurrences of major internal functions to process a single (ind, w) in Sophos and our scheme.

Such real-time efficiency can be verified by comparing the implementation performance presented in Section 6 with Sophos which has similar complexities at the point of computation and communication.

Finally, forward security is achieved by designing our scheme to use a *fresh* key, after a search query is processed, to encrypt identifiers of newly added documents. This makes old search tokens unusable and so to mitigate the leakage from updates. Such key usage in the search operation is shown in Figure 1.

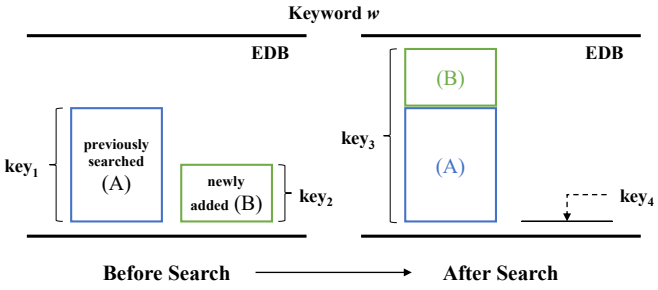

Figure 1: Key usage in the search operation

In the search process for a keyword w, our scheme encrypts identifiers of all searched documents with key_1. After that, identifiers of the documents containing w added to the EDB until the next search query for w is issued are encrypted with key_2. The new search token for w includes key_1 and key_2, and is used for searching identifiers of documents containing w in the EDB. As previous, our scheme re-encrypts identifiers of all searched documents with key_3. Thereafter, identifiers of the documents added up to the next search are encrypted with key_4. Since all of such keys are independently generated by the client, the search token containing key_1 and key_2 cannot be used to find the documents inserted after the search.

Our scheme is simple, easy to implement, non-interactive and practical for dealing with a large number of document/keyword pairs. To show such properties of our scheme, we implemented our scheme and ran experiments with the Enron email and Wikipedia datasets. Such experimental results show that our scheme works very efficiently in dynamic environments. We refer to Sections 6 and 7 for more information on experiments and performance.

2 PRELIMINARIES AND MODELS

2.1 Notations

Let $\lambda \in \mathbb{N}$ denote the security parameter and we will assume all algorithms take λ implicitly as input. Operators $\|$ and $|x|$ denote the concatenation and the bit length of variable x, respectively. For a finite set X, $|X|$ denotes the cardinality of it. And $x \xleftarrow{\$} X$ means that x is selected uniformly at random from the set X. The output x of an algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$ (or $\mathcal{A} \rightarrow x$). Let $\{0,1\}^\ell$ denote the set of all binary strings of length ℓ and $\{0,1\}^*$ be the set of all binary strings of finite length.

A function $\nu : \mathbb{N} \rightarrow \mathbb{N}$ is negligible in k if for every positive polynomial $p(\cdot)$ and all sufficiently large k, $\nu(k) < 1/p(k)$. We write $f(k) = \mathsf{negl}(k)$ to mean that there exists a negligible function $\nu(\cdot)$ such that $f(k) \leq \nu(k)$ for all sufficiently large k. Unless specified explicitly, the symmetric keys are strings of λ bits, and the key generation algorithm uniformly samples a key in $\{0,1\}^\lambda$. We only consider (possibly probabilistic) algorithms and protocols running in polynomial time in the security parameter λ. In particular, adversaries are probabilistic polynomial-time algorithms.

2.2 Pseudo-Random Function

Our construction is based on a pseudo-random function (PRF) that is a polynomial-time computable function and is indistinguishable from a true random function by any probabilistic polynomial-time adversary. We use the standard security definitions of *prf-advantage* of an adversary $\mathcal{A}$ for a PRF F and denote it by $\mathsf{Adv}^{\mathsf{prf}}_{\mathcal{A},F}(\lambda)$ [20]. We say that a PRF F is secure if $\mathsf{Adv}^{\mathsf{prf}}_{\mathcal{A},F}(\lambda)$ is negligible for all probabilistic polynomial-time algorithm $\mathcal{A}$.

2.3 (Dynamic) Searchable Symmetric Encryption

We follow the formalizations of Curtmola *et al.* [13] and Stepanov *et al.* [31] with some modifications.

A document $f = (\mathsf{ind}, \mathsf{DB}(\mathsf{ind}))$ consists of a document identifier $\mathsf{ind} \in \{0,1\}^\ell$ and a contained keyword-set $\mathsf{DB}(\mathsf{ind}) \subseteq \{0,1\}^*$. Then, a database DB can be defined as $(\mathsf{ind}_i, \mathsf{DB}(\mathsf{ind}_i))_{i=1}^n$. The set of all keywords of DB is $\mathsf{W} = \cup_{i=1}^n \mathsf{DB}(\mathsf{ind}_i)$, and the set of documents containing a keyword w is $\mathsf{DB}(w) = \{\mathsf{ind}_i \mid w \in \mathsf{DB}(\mathsf{ind}_i)\}$. Let n be the number of documents in DB, $m = |\mathsf{W}|$ be the total number of keywords, $n_w = |\mathsf{DB}(w)|$, and N be the number of document/keyword pairs (we identify documents with their identifier). Note that N can be written as $N = \sum_{i=1}^n |\mathsf{DB}(\mathsf{ind}_i)| = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$.

Table 1: Comparison with DSSE schemes supporting forward security

Scheme	Data		Communication		Computation	
	Client	Server	Search	Update	Search	Update
[31]	$O(N^{\alpha})_{(0<\alpha<1)}$	$O(N^{+})$	$O(n_w + \log N^{+})$	$O(\log N^{+})$	$O(\min\{a_w + \log N^{+}, n_w \log^3 N^{+}\})$	$O(\log^2 N^{+})$
[32]	$O(m+n)$	$O(m \times n)$	$O(n_w)$	$O(m)$	$O(n)$	$O(m)$
[19]	$O(1)$	$O(m+N)$	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(k \log^3 N)$	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(k \log^2 N)$
[4]	$O(m)$	$O(N^{+})$	$O(n_w)$	$O(k)$	$O(a_w + d_w)$	$O(k)$
Ours	$O(m)$	$O(N)$	$O(n_w)$	$O(k)$	$O(a_w)$	$O(k)$

The complexities are based on retrieving documents containing a keyword w or updating documents containing k unique keywords. The following notations are used throughout the paper. N is the total number of document/keyword pairs in the database, while m (resp. n) is the number of keywords (resp. documents) in the database. n_w is the size of search result set for keyword w, and a_w (resp. d_w) is the number of times the queried keyword w was historically *added to* (resp. *deleted from*) the database. N^{+} is the total number of document/keyword pairs historically stored in the database, i.e., $N^{+} = \sum_w (a_w + d_w)$. The notation $\tilde{O}$ hides the $\log \log N$ factors.

Table 2: Comparison of the number of major internal functions in Sophos and our scheme for a single pair (ind, w)

Scheme	C/S	Search		Add			
		T	H	F	T^{-1}	H	F
Sophos	Client	-	-	1	1	2	1
	Server	1	2	-	-	-	-
Ours	Client	-	-	-	-	3	1
	Server	-	4	-	-	-	-

T: trapdoor permutation, T^{-1}: inverse of trapdoor permutation, H: hash function, F: PRF

Let C and S denote a client and a server, respectively. Let P denote a two-party protocol between C and S such as

$$P(\mathsf{in}_C; \mathsf{in}_S) = (P_C(\mathsf{in}_C), P_S(\mathsf{in}_S))$$

meaning that P_C (resp. P_S) is executed by the client (resp. the server) with input in_C (resp. in_S). We write

$$P(\mathsf{in}_C; \mathsf{in}_S) \rightarrow (\mathsf{out}_C\ ;\ \mathsf{out}_S)$$

to mean that out_C and out_S are the outputs of the protocol P, involving C on input in_C and S on input in_S.

A dynamic searchable symmetric encryption (DSSE) scheme is a tuple of four polynomial-time protocols SE = (Setup, Search, Addition, Deletion) between a client and a server such that:

- SE.Setup$(\lambda\ ;\ \perp) \rightarrow (\sigma\ ;\ \mathsf{EDB})$**:** It takes as input a security parameter λ and creates a state σ for the client. It also initializes an encrypted database EDB for the server.
- SE.Search$((\sigma, w)\ ;\ \mathsf{EDB}) \rightarrow ((\sigma', \mathsf{DB}(w))\ ;\ \mathsf{EDB}')$**:** The client takes as input the state σ and a keyword w, and the server takes as input the encrypted database EDB. After execution of protocol, the client outputs an updated state σ' and the set $\mathsf{DB}(w)$ of document (identifiers) containing the keyword w. The server outputs an updated encrypted database EDB'.
- SE.Addition$((\sigma, f)\ ;\ \mathsf{EDB}) \rightarrow (\sigma'\ ;\ \mathsf{EDB}')$**:** The client takes as input the state σ and a document f, and the server takes as input the encrypted database EDB. After running the protocol, the client outputs an updated state σ' and the server outputs an updated encrypted database EDB' including f.
- SE.Deletion$((\sigma, \mathsf{ind})\ ;\ \mathsf{EDB}) \rightarrow (\emptyset\ ;\ \mathsf{EDB}')$**:** The client takes as input the state σ and a document identifier ind, and the server takes as input the encrypted database EDB. As a result, the server outputs an updated encrypted database EDB' excluding a document with identifier ind.

We say an DSSE scheme is *correct* if, for every keyword w, SE.Search returns $\mathsf{DB}(w)$ except with negligible probability. We also say a DSSE scheme is *non-interactive* if SE.Search is a two-round protocol and both SE.Addition and SE.Deletion are single round protocols.

2.4 Security Model

The standard security definition of a DSSE scheme follows the ideal or real simulation paradigm [7, 13, 24]. It is parameterized by a collection of *leakage functions*

$$\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Search}}, \mathcal{L}_{\mathsf{Addition}}, \mathcal{L}_{\mathsf{Deletion}})$$

describing what information the protocol leaks to the adversary, and formalized as stateful algorithms. The definition ensures that the scheme does not reveal any information beyond what can be inferred from the leakage functions.

More precisely, we define two games $\mathsf{Game}_{R,\mathcal{A}}$ and $\mathsf{Game}_{S,\mathcal{A}}$ as follows. In the real $\mathsf{Game}_{R,\mathcal{A}}$ the adversary $\mathcal{A}$ is given EDB generated by $\mathsf{Setup}(\lambda; \perp)$ as in the real case. In the ideal $\mathsf{Game}_{\mathcal{S},\mathcal{A}}$ the adversary $\mathcal{A}$ is given $\mathcal{S}(\mathcal{L}_{\mathsf{Setup}}(\lambda; \perp))$. The adversary then repeatedly performs search, addition, and deletion queries and receives the transcripts generated from $\mathsf{Search}(w)$, $\mathsf{Addition}(f)$, and $\mathsf{Deletion}(\mathsf{ind})$ algorithms in the real game, or receives the transcripts generated by the simulator $\mathcal{S}(\mathcal{L}_{\mathsf{Search}}(w))$, $\mathcal{S}(\mathcal{L}_{\mathsf{Addition}}(f))$, and $\mathcal{S}(\mathcal{L}_{\mathsf{Deletion}}(\mathsf{ind}))$ in the ideal game. Eventually, $\mathcal{A}$ outputs a bit 0 ($\mathsf{Game}_{R,\mathcal{A}}$) or 1 ($\mathsf{Game}_{\mathcal{S},\mathcal{A}}$).

Definition 2.1. A DSSE scheme is $\mathcal{L}$*-adaptively-secure* if for all probabilistic polynomial-time algorithm $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ such that the following equation holds:

$$\left|\Pr[\mathsf{Game}_{R,\mathcal{A}}(\lambda)=1]-\Pr[\mathsf{Game}_{\mathcal{S},\mathcal{A}}(\lambda)=1]\right|\leq \mathsf{negl}(\lambda).$$

Here, the leakage function collection $\mathcal{L}$ will keep the list Q of all search queries issued so far as a state. Each entry of the query list Q is of the form (i, w) where w is a queried keyword and the integer i is a timestamp, initially set to 0 and incremented at each query. For each keyword w, the search pattern $\mathsf{sp}(w)$ is defined as

$$\mathsf{sp}(w)=\{j \mid (j,w)\in Q\} \quad \text{(only matches search queries)}.$$

We also use the notation $\mathsf{HistDB}(w)$ described in [4]. It is the list of documents historically added to DB containing the keyword w in the order of insertion. In particular, it includes document identifiers that have been added and deleted later. For example, $\mathsf{HistDB}(w)=\langle \mathsf{ind}_1, \mathsf{ind}_2\rangle$ means that ind_1 and ind_2 have been added sequentially so far, but it does not tell that which documents have been deleted.

Forward Security. As mentioned before, forward security means that an addition query does not leak any keyword information of the newly added documents. We will follow the definition of [4] due to its clarity.

Definition 2.2. A DSSE scheme is *forward secure* if there exists a leakage function $\bar{\mathcal{L}}$ such that its $\mathcal{L}_{\mathsf{Addition}}$ can be written as

$$\mathcal{L}_{\mathsf{Addition}}(\mathsf{ind},\mathsf{W})=\bar{\mathcal{L}}(\mathsf{ind},|\mathsf{W}|).$$

The definition means that the addition operation of any forward secure DSSE scheme does not leak more than the identifier and the number of keywords of the newly added document.

Backward Security. The authors of [31] have described backward security that prevents search queries to be performed over deleted data. If a document containing w have been deleted before a search query for w is issued, the server should not be sure that if the deleted document contains w from the subsequent search queries. To our best knowledge, the only existing DSSE schemes providing both forward and backward securities are based on ORAM techniques.

3 THE PROPOSED DSSE SCHEME

Most of the practical SSE schemes developed thus far configure data using an inverted index for efficient search and store it using a single dictionary data structure. However, when the inverted index is used, it is inevitably difficult to update documents. The problem of document addition could be solved by the client retaining states about the size of $\mathsf{DB}(w)$, but there is no known SSE schemes that supports substantial document deletion. In order to overcome these limitations and utilize structural advantages of both index structures, we propose a new data structure called *dual dictionary* that combines the inverted index to guarantee optimal search time and the forward index for efficient update.

3.1 Building Blocks

We first recall the single dictionary data structure.

A *dictionary* Dict is a data structure that maintains a set of $(\mathsf{label}, \mathsf{data}) \in \{0,1\}^*\times\{0,1\}^*$ pairs where each possible label appears at most once in the set, and that is given the following operations to manage these pairs.

- $\mathsf{Dict.Create}(\{(\mathsf{label}_i, \mathsf{data}_i) \mid 1\leq i\leq N\}) \rightarrow \mathsf{Dic}$
- $\mathsf{Dict.Get}(\mathsf{Dic}, \mathsf{label}) \rightarrow \mathsf{data}$ or $\perp$
- $\mathsf{Dict.Insert}(\mathsf{Dic}, (\mathsf{label}, \mathsf{data})) \rightarrow \mathsf{Dic}'$
- $\mathsf{Dict.Remove}(\mathsf{Dic}, \mathsf{label}) \rightarrow \mathsf{Dic}'$ or $\perp$

We now introduce the *dual dictionary* data structure which manages a data set with two independent labels by linking two different dictionaries Dict_1 and Dict_2.

Definition 3.1. The *dual dictionary* DLDict is a data structure that maintains a set $\{(\mathsf{label}_i^{(1)}, \mathsf{label}_i^{(2)}, \mathsf{data}_i) \in \{0,1\}^*\times\{0,1\}^*\times\{0,1\}^* \mid \mathsf{label}_i^{(1)}\neq \mathsf{label}_j^{(1)} \wedge \mathsf{label}_i^{(2)}\neq \mathsf{label}_j^{(2)} \Leftrightarrow i\neq j\}$, and that is given the following operations to manage these tuples:

- $\mathsf{DLDict.Create}\big(\{(\mathsf{label}_i^{(1)}, \mathsf{label}_i^{(2)}, \mathsf{data}_i) \mid 1\leq i\leq N\}\big) \rightarrow \mathsf{DLDic} = (\mathsf{Dic}_1, \mathsf{Dic}_2)$:
  $\mathsf{Dic}_1 \leftarrow \mathsf{Dict.Create}\big(\{(\mathsf{label}_i^{(1)}, \mathsf{label}_i^{(2)}) \mid 1\leq i\leq N\}\big)$
  $\mathsf{Dic}_2 \leftarrow \mathsf{Dict.Create}\big(\{(\mathsf{label}_i^{(2)}, (\mathsf{label}_i^{(1)}, \mathsf{data}_i)) \mid 1\leq i\leq N\}\big)$
  return $\mathsf{DLDic} \leftarrow (\mathsf{Dic}_1, \mathsf{Dic}_2)$
- $\mathsf{DLDict.Insert}(\mathsf{DLDic}, (\mathsf{label}, \mathsf{data})) \rightarrow \mathsf{DLDic}' = (\mathsf{Dic}_1', \mathsf{Dic}_2')$:
  $\mathsf{Dic}_1' \leftarrow \mathsf{Dict.Insert}(\mathsf{Dic}_1, (\mathsf{label}^{(1)}, \mathsf{label}^{(2)}))$
  $\mathsf{Dic}_2' \leftarrow \mathsf{Dict.Insert}(\mathsf{Dic}_2, (\mathsf{label}^{(2)}, (\mathsf{label}^{(1)}, \mathsf{data})))$
  return $\mathsf{DLDic}' \leftarrow (\mathsf{Dic}_1', \mathsf{Dic}_2')$
- $\mathsf{DLDict.Get}(\mathsf{Dic}, \mathsf{label}^{(1)}) \rightarrow (\mathsf{label}^{(2)}, \mathsf{data})$ or $\perp$:
  if $\mathsf{Dict.Get}(\mathsf{Dic}_1, \mathsf{label}^{(1)}) = \perp$ **then**
  return $\perp$
  end if
  $\mathsf{label}^{(2)} \leftarrow \mathsf{Dict.Get}(\mathsf{Dic}_1, \mathsf{label}^{(1)})$
  $(\mathsf{label}^{(1)}, \mathsf{data}) \leftarrow \mathsf{Dict.Get}(\mathsf{Dic}_2, \mathsf{label}^{(2)})$
  return $(\mathsf{label}^{(2)}, \mathsf{data})$
- $\mathsf{DLDict.Get}(\mathsf{Dic}, \mathsf{label}^{(2)}) \rightarrow (\mathsf{label}^{(1)}, \mathsf{data})$ or $\perp$:
  if $\mathsf{Dict.Get}(\mathsf{Dic}_2, \mathsf{label}^{(2)}) = \perp$ **then**
  return $\perp$
  end if
  return $(\mathsf{label}^{(1)}, \mathsf{data}) \leftarrow \mathsf{Dict.Get}(\mathsf{Dic}_2, \mathsf{label}^{(2)})$
- $\mathsf{DLDict.Remove}(\mathsf{Dic}, \mathsf{label}^{(1)}) \rightarrow \mathsf{DLDic}' = (\mathsf{Dic}_1', \mathsf{Dic}_2')$ or $\perp$:
  if $\mathsf{Dict.Get}(\mathsf{Dic}_1, \mathsf{label}^{(1)}) = \perp$ **then**
  return $\perp$
  end if
  $\mathsf{label}^{(2)} \leftarrow \mathsf{Dict.Get}(\mathsf{Dic}_1, \mathsf{label}^{(1)})$
  $\mathsf{Dic}_1' \leftarrow \mathsf{Dict.Remove}(\mathsf{Dic}_1, \mathsf{label}^{(1)})$
  $\mathsf{Dic}_2' \leftarrow \mathsf{Dict.Remove}(\mathsf{Dic}_2, \mathsf{label}^{(2)})$
  return $\mathsf{DLDic}' \leftarrow (\mathsf{Dic}_1', \mathsf{Dic}_2')$
- $\mathsf{DLDict.Remove}(\mathsf{Dic}, \mathsf{label}^{(2)}) \rightarrow \mathsf{DLDic}' = (\mathsf{Dic}_1', \mathsf{Dic}_2')$ or $\perp$:
  if $\mathsf{Dict.Get}(\mathsf{Dic}_2, \mathsf{label}^{(2)}) = \perp$ **then**
  return $\perp$

end if
$(\mathsf{label}^{(1)}, \mathsf{data}) \leftarrow \mathsf{Dict.Get}(\mathsf{Dic}_2, \mathsf{label}^{(2)})$
$\mathsf{Dic}_1' \leftarrow \mathsf{Dict.Remove}(\mathsf{Dic}_1, \mathsf{label}^{(1)})$
$\mathsf{Dic}_2' \leftarrow \mathsf{Dict.Remove}(\mathsf{Dic}_2, \mathsf{label}^{(2)})$
return $\mathsf{DLDic}' \leftarrow (\mathsf{Dic}_1', \mathsf{Dic}_2')$

A dual dictionary expands a single dictionary in a double manner, so it needs about twice the amount of storage and computation. However, from an asymptotic point of view, it is possible to manage data only by $O(1)$ operation, like a single dictionary data structure.

In the proposed scheme, an EDB is configured by the dual dictionary DLDict as follows[1]: for each (ind, w) where $w \in \mathsf{DB}(\mathsf{ind})$, there are two tuples $(\mathsf{label}^{(1)}, \mathsf{label}^{(2)})$ and $(\mathsf{label}^{(2)}, (\mathsf{label}^{(1)}, \mathsf{ind}))$ in the dual dictionary so that one can restore (ind, w) using $\mathsf{label}^{(1)}$ derived from ind or using $\mathsf{label}^{(2)}$ derived from w. When data is stored in such a manner, substantial deletion of data becomes possible using $\mathsf{label}^{(1)}$. In addition, $\mathsf{label}^{(2)}$ enables efficient data addition and search. In our construction, we denote $\mathsf{label}^{(1)}$ and $\mathsf{label}^{(2)}$ by $\mathsf{label}^{(\mathsf{ind})}$ and $\mathsf{label}^{(w)}$ respectively.

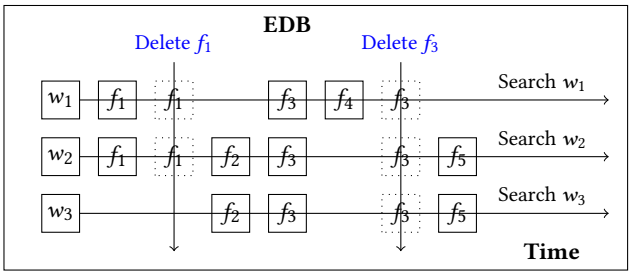

Figure 2: Basic idea

Figure 2 shows the basic idea of our scheme. We configure an EDB in an inverted index structure to efficiently perform the search operation. In addition, a data-centric forward index is constructed and combined into the dual dictionary data structure to enable the actual deletion of data.

3.2 Construction

We now introduce our main scheme. In the following pseudo codes, $H_i : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^{\mu(=2\lambda)}$ $(i = 1, 2.3)$ are keyed hash functions and $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ is a PRF.

SE.Setup. The client should maintain a (single) dictionary $\mathsf{Dict}_{\mathsf{kwd}}$ to manage the secret key and counter information corresponding to each keyword. Independently, the client should also maintain a master secret key $\mathsf{key}_{\mathsf{prf}}$ for the PRF F. The client's storage for maintaining the state $\sigma = (\mathsf{key}_{\mathsf{prf}}, \mathsf{Dict}_{\mathsf{kwd}})$ is thus $O(m)$. The client creates an EDB as a dual dictionary data structure and sends it to the server. Algorithm 1 gives a formal description of SE.Setup operation.

The dictionary $\mathsf{Dict}_{\mathsf{kwd}}$ stores a tuple $(\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$ for each keyword w. The tuple is initialized as $(\emptyset, 0, \mathsf{ukey}^{(w)}, 0)$ where $\mathsf{ukey}^{(w)} \xleftarrow{\$} \{0,1\}^\lambda$ is generated by the client when the first time a document containing the keyword w is uploaded to EDB (see lines 5 - 8 in Algorithm 2). The binary strings $\mathsf{key}^{(w)}$ and $\mathsf{ukey}^{(w)}$ are secret keys used to encrypt identifiers of documents containing w. More precisely, $\mathsf{key}^{(w)}$ is the key used to encrypt document identifiers in the EDB and $\mathsf{ukey}^{(w)}$ is the key prepared to encrypt document identifiers to be inserted. The number $\mathsf{cnt}^{(w)}$ represents the number of documents in the EDB containing w and $\mathsf{ucnt}^{(w)}$ is used to count the number of added documents containing w before querying w.

[1]It is not exactly the same as the propose scheme since we omit the role of counter for convenience of explanation.

Algorithm 1 $\mathsf{SE.Setup}(\lambda\ ;\ \perp) \to (\sigma\ ;\ \mathsf{EDB})$

Client$(\lambda) \to (\sigma, \mathsf{EDB})$

1: $\mathsf{Dict}_{\mathsf{kwd}} \leftarrow \mathsf{Dict.Create}(\emptyset)$; $\mathsf{key}_{\mathsf{prf}} \xleftarrow{\$} \{0,1\}^\lambda$
2: $\sigma \leftarrow (\mathsf{key}_{\mathsf{prf}}, \mathsf{Dict}_{\mathsf{kwd}})$
3: $\mathsf{EDB} \leftarrow \mathsf{DLDict.Create}(\emptyset)$
4: **Send** EDB **to** Server

SE.Addition. To add a document f in the EDB, the client first computes a secret key $\mathsf{key}^{(\mathsf{ind})} = F(\mathsf{key}_{\mathsf{prf}}, \mathsf{ind})$ where ind denotes the identifier of f. For each keyword $w \in \mathsf{DB}(\mathsf{ind})$ in a random order, the client generates two labels $\mathsf{label}^{(\mathsf{ind})} = H_1(\mathsf{key}^{(\mathsf{ind})}, \mathsf{cnt}^{(\mathsf{ind})})$ and $\mathsf{label}^{(w)} = H_2(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$, where the counter $\mathsf{cnt}^{(\mathsf{ind})}$ reflects the order of w in $\mathsf{DB}(\mathsf{ind})$, and $\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)}$ are keyword-specific values obtained from $\mathsf{Dict}_{\mathsf{kwd}}$. The key/counter pair $(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$ is also used to encrypt the document identifier ind as $\mathsf{data} = \mathsf{ind} \oplus H_3(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$. Finally, the client uploads the set $\mathsf{AddSet} = \{(\mathsf{label}^{(\mathsf{ind})}, \mathsf{label}^{(w)}, \mathsf{data}) \mid w \in \mathsf{DB}(\mathsf{ind})\}$ to the server.

Note that the dictionary $\mathsf{Dict}_{\mathsf{kwd}}$ keeps two key/counter pairs for each keyword $w \in \mathsf{W}$ and only $(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$ pair is used for encrypting the document identifier in the SE.Addition operation. On the other hand, the key/counter pair $(\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)})$ is used by the server to encrypt all document identifiers stored just before the most recently requested search query for the keyword w. By separating encryption keys for previously searched data and newly added data, the server cannot determine the association between the data retrieved in the past and the newly added data, until it must be revealed by the next search query. For more information, see the description of the SE.Search operation below. Algorithm 2 gives a formal description of SE.Addition operation.

Both the computational and communication complexities of SE.Addition operation are $O(k)$ because labels are generated and transmitted corresponding to the number of keywords included in the document f.

SE.Deletion. The delete operation for removing document identifiers from the EDB is very simple and intuitive. For the identifier ind of a document to be deleted, the key $\mathsf{key}^{(\mathsf{ind})}$ is derived from $\mathsf{key}_{\mathsf{prf}}$ and transmitted to the server as a deletion token. The server calculates $\mathsf{label}^{(\mathsf{ind})} = H_1(\mathsf{key}_{\mathsf{prf}}, \mathsf{cnt}^{(\mathsf{ind})})$ and deletes the corresponding document identifier from the EDB by calling $\mathsf{DLDict.Remove}(\mathsf{EDB}, \mathsf{label}^{(\mathsf{ind})})$. The server repeats this process by incrementing the counter $\mathsf{cnt}^{(\mathsf{ind})}$, starting with 1, until $\perp$ is returned by $\mathsf{DLDict.Remove}(\mathsf{EDB}, \mathsf{label}^{(\mathsf{ind})})$.

Algorithm 2 SE.Addition$((\sigma, f)\,;\ \mathsf{EDB}) \rightarrow (\sigma'\,;\ \mathsf{EDB}')$

Client$(\sigma, f = (\mathsf{ind}, \mathsf{DB}(\mathsf{ind}))) \rightarrow (\sigma', \mathsf{AddSet})$

1: $\mathsf{key}^{(\mathsf{ind})} \leftarrow F(\mathsf{key}_{\mathsf{prf}}, \mathsf{ind})\,;\quad \mathsf{cnt}^{(\mathsf{ind})} \leftarrow 0\,;\quad \mathsf{AddSet} \leftarrow \emptyset$
2: $\mathsf{RefSet} \leftarrow \mathsf{DB}(\mathsf{ind})$
3: **while** $|\mathsf{RefSet}| \neq 0$ **do**
4: $\quad w \xleftarrow{\$} \mathsf{RefSet}\,;\quad \mathsf{RefSet} \leftarrow \mathsf{RefSet} \setminus \{w\}$
5: $\quad$ **if** $\mathsf{Dict}_{\mathsf{kwd}}.\mathsf{Get}(w) = \perp$ **then**
6: $\quad\quad \mathsf{key}^{(w)} \leftarrow \emptyset\,;\quad \mathsf{cnt}^{(w)} \leftarrow 0\,;$
7: $\quad\quad \mathsf{ukey}^{(w)} \xleftarrow{\$} \{0,1\}^{\lambda}\,;\quad \mathsf{ucnt}^{(w)} \leftarrow 0$
8: $\quad\quad \mathsf{Dict}_{\mathsf{kwd}} \leftarrow \mathsf{Dict}_{\mathsf{kwd}}.\mathsf{Insert}(w, (\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)}))$
9: $\quad$ **else**
10: $\quad\quad (\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)}) \leftarrow \mathsf{Dict}_{\mathsf{kwd}}.\mathsf{Get}(w)$
11: $\quad$ **end if**
12: $\quad \mathsf{cnt}^{(\mathsf{ind})} \leftarrow \mathsf{cnt}^{(\mathsf{ind})} + 1\,;\quad \mathsf{label}^{(\mathsf{ind})} \leftarrow H_1(\mathsf{key}^{(\mathsf{ind})}, \mathsf{cnt}^{(\mathsf{ind})})$
13: $\quad \mathsf{ucnt}^{(w)} \leftarrow \mathsf{ucnt}^{(w)} + 1\,;\quad \mathsf{label}^{(w)} \leftarrow H_2(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$
14: $\quad \mathsf{data} \leftarrow \mathsf{ind} \oplus H_3(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$
15: $\quad \mathsf{AddSet} \leftarrow \mathsf{AddSet} \cup \{(\mathsf{label}^{(\mathsf{ind})}, \mathsf{label}^{(w)}, \mathsf{data})\}$
16: $\quad \mathsf{Dict}_{\mathsf{kwd}} \leftarrow \mathsf{Dict}_{\mathsf{kwd}}.\mathsf{Insert}(w, (\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)}))$
17: **end while**
18: $\sigma' \leftarrow (\mathsf{key}_{\mathsf{prf}}, \mathsf{Dict}_{\mathsf{kwd}})$
19: **send** AddSet **to** Server

Server$(\mathsf{EDB}, \mathsf{AddSet}) \rightarrow \mathsf{EDB}'$

1: **for** each $(\mathsf{label}^{(\mathsf{ind})}, \mathsf{label}^{(w)}, \mathsf{data}) \in \mathsf{AddSet}$ **do**
2: $\quad \mathsf{EDB} \leftarrow \mathsf{DLDict.Insert}(\mathsf{EDB}, (\mathsf{label}^{(\mathsf{ind})}, \mathsf{label}^{(w)}, \mathsf{data}))$
3: **end for**
4: $\mathsf{EDB}' \leftarrow \mathsf{EDB}$

Algorithm 3 SE.Deletion$((\sigma, \mathsf{ind})\,;\ \mathsf{EDB}) \rightarrow (\emptyset\,;\ \mathsf{EDB}')$

Client$(\sigma, \mathsf{ind}) \rightarrow \mathsf{dtoken}^{(\mathsf{ind})}$

1: $\mathsf{key}^{(\mathsf{ind})} \leftarrow F(\mathsf{key}_{\mathsf{prf}}, \mathsf{ind})$
2: $\mathsf{dtoken}^{(\mathsf{ind})} \leftarrow \mathsf{key}^{(\mathsf{ind})}$
3: **send** $\mathsf{dtoken}^{(\mathsf{ind})}$ **to** Server

Server$(\mathsf{dtoken}^{(\mathsf{ind})}, \mathsf{EDB}) \rightarrow \mathsf{EDB}'$

1: $\mathsf{cnt}^{(\mathsf{ind})} \leftarrow 1$
2: **while** (1) **do**
3: $\quad \mathsf{label}^{(\mathsf{ind})} \leftarrow H_1(\mathsf{dtoken}^{(\mathsf{ind})}, \mathsf{cnt}^{(\mathsf{ind})})$
4: $\quad$ **if** $\mathsf{DLDict.Remove}(\mathsf{EDB}, \mathsf{label}^{(\mathsf{ind})}) = \perp$ **then**
5: $\quad\quad$ **break**
6: $\quad$ **else**
7: $\quad\quad \mathsf{EDB} \leftarrow \mathsf{DLDict.Remove}(\mathsf{EDB}, \mathsf{label}^{(\mathsf{ind})})$
8: $\quad\quad \mathsf{cnt}^{(\mathsf{ind})} \leftarrow \mathsf{cnt}^{(\mathsf{ind})} + 1$
9: $\quad$ **end if**
10: **end while**
11: $\mathsf{EDB}' \leftarrow \mathsf{EDB}$

Since the value of the corresponding location completely removed from the EDB, it can support substantial document deletion. Therefore the size of the EDB is optimal as $O(N)$, which is the size of data that should be maintained by the client's current request excluding the deleted data from the inserted data. The computational complexity is $O(k)$ because the server must perform DLDict.Remove as many times as the number of keywords the corresponding data contains, and the communication complexity is $O(1)$ because the deletion token is composed of only one key.

SE.Search. To find documents containing a keyword w, the client first retrieves $(\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$ from $\mathsf{Dict}_{\mathsf{kwd}}$. At the same time, the client creates a new secret key $\mathsf{nkey}^{(w)}$. And then sends them to the server as a search token $\mathsf{token}^{(w)}$. At this point, all the relevant document identifiers stored on the EDB are encrypted with $(\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)})$ or $(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$ pair. As described in the SE.Addition operation, all document identifiers stored before the previous search query for the keyword w is encrypted with $(\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)})$, and the newly added document identifiers thereafter are encrypted with $(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$.

Given a search token

$$\mathsf{token}^{(w)} = (\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)}, \mathsf{nkey}^{(w)}),$$

the server uses $\mathsf{key}^{(w)}$ to search document identifiers by increasing a counter from 1 to $\mathsf{cnt}^{(w)}$. In the same way, $\mathsf{ukey}^{(w)}$ and $\mathsf{ucnt}^{(w)}$ are applied to further search another document identifiers. If there are deleted documents, it may happen that the extraction of associated document identifiers fails in this process (lines 3 - 5 in the **SUEdb** function), but the retrieval proceeds up to $\mathsf{cnt}^{(w)} + \mathsf{ucnt}^{(w)}$. One of the most important features of our scheme is re-encrypting and storing the search results. Whenever a document identifier ind is extracted, the server creates a label $\mathsf{nlabel}^{(w)}$ and a data ndata using the new key $\mathsf{nkey}^{(w)}$ contained in the search token and a new inner counter $\mathsf{ncnt}^{(w)}(= j)$ (line 10 in the **SUEdb** function), and replaces old data in the EDB to it. In this re-encryption process, the counter $\mathsf{ncnt}^{(w)}(= j)$ associated to the key $\mathsf{nkey}^{(w)}$ is set by reflecting the current number of document identifiers stored in the EDB, and so an unnecessary process of searching for labels of the deleted documents in the later search process can be omitted. The server shares this new key/counter pair $(\mathsf{nkey}^{(w)}, \mathsf{ncnt}^{(w)})$ with the client by sending $\mathsf{ncnt}^{(w)}$ with the search result, and then the client update $\mathsf{Dict}_{\mathsf{kwd}}$ for the keyword w by putting $(\mathsf{nkey}^{(w)}, \mathsf{ncnt}^{(w)})$ at the location of first key/counter pair in the associated data. Figure 3 shows the changes of $\mathsf{Dict}_{\mathsf{kwd}}$ and EDB before and after a search operation.

As noted in [32], this approach provides privacy if the server is compromised by an outsider after a search operation occurs (the server deletes the search token from the memory after the search processing is completed). And it keeps DLDict consistent for consecutive search operations performed on the same keyword. Algorithm 4 gives a formal description of SE.Search operation.

As an optimization technique of the search operation, it is possible to delay the rewriting process after the search result is returned. This can minimize the influence of the write process and can be utilized especially when the size of an EDB is larger than that of memory.

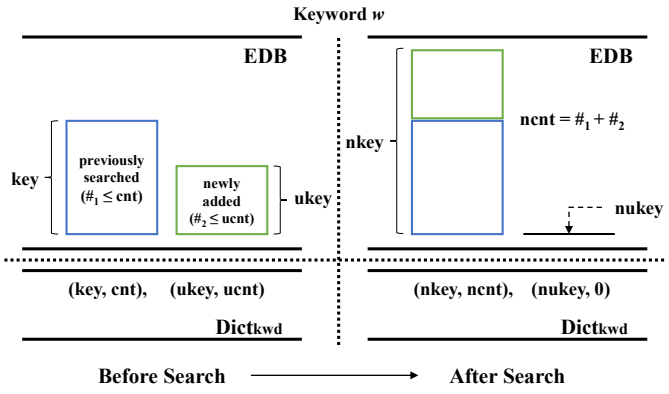

Figure 3: Search & re-encryption in SE.Search

The amount of data required in the search operation is $O(n_w)$ because it is exactly the number of documents containing the keyword w, and the computational complexity is $O(a_w)$ because the search operation must be performed the number of times that the documents containing w is added. However, since the deleted number of documents is reflected to the counter in the search process, the optimal search time $O(n_w)$ is satisfied from the second search operation unless additional deletion occurs.

3.3 Correctness

The client has a dictionary $\mathsf{Dict}_{\mathsf{kwd}}$ for managing the keys used to encrypt document identifiers. This dictionary $\mathsf{Dict}_{\mathsf{kwd}}$ is updated by SE.Search and SE.Addition operations, and when there is no information about the key corresponding to the keyword in the search query, SE.Search stops itself. Thus, it can be seen that SE.Addition must be performed prior to SE.Search. The dictionary $\mathsf{Dict}_{\mathsf{kwd}}$ maps each keyword to two pairs of key and counter. The counter paired with each key indicates the number of times that the corresponding key is used to encrypt document identifiers.

The data added by DLDict.Insert(EDB, $(\mathsf{label}^{(\mathsf{ind})}, \mathsf{label}^{(w)}, \mathsf{data})$) until just before the first SE.Search$((\sigma, w)\,;\ \mathsf{EDB})$ runs have been encrypted using the second key ukey managed by $\mathsf{Dict}_{\mathsf{kwd}}$, and the corresponding counter ucnt is incremented sequentially. A search token consists of the randomly generated key nkey and the value in $\mathsf{Dict}_{\mathsf{kwd}}$ corresponding to the keyword to be searched. Given the search token, the server handles the search process using each of the two key/counter pairs. If the keyword is searched for the first time, the counter cnt of the first key pair is 0, so only the second key pair is used for the search. While processing the search query, the server performs re-encryption using the new key nkey included in the search token and returns the corresponding counter ncnt to the client along with the search result. The client then substitutes (nkey, ncnt) for (key, cnt) corresponding to the keyword, generates a new key nukey, and replaces ukey with nukey in $\mathsf{Dict}_{\mathsf{kwd}}$. It can be seen that the data added after the execution of SE.Search$((\sigma, w)\,;\ \mathsf{EDB})$ are encrypted using a key that is independent of the key used in the past. And the server always has data encrypted with up to two keys (See Figure 3).

Algorithm 4 SE.Search$((\sigma, w)\,;\ \mathsf{EDB}) \to ((\sigma', \mathsf{DB}(w))\,;\ \mathsf{EDB}')$

Client$(\sigma, w) \to \mathsf{token}^{(w)}$

1: $\mathsf{nkey}^{(w)} \xleftarrow{\$} \{0,1\}^{\lambda}$
2: **if** $\mathsf{Dict.Get}(\mathsf{Dict}_{\mathsf{kwd}}, w) = \perp$ **then**
3: **return** $\emptyset$
4: **end if**
5: $(\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)}) \leftarrow \mathsf{Dict.Get}(\mathsf{Dict}_{\mathsf{kwd}}, w)$
6: $\mathsf{token}^{(w)} \leftarrow (\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)}, \mathsf{nkey}^{(w)})$
7: **send** $\mathsf{token}^{(w)}$ **to** Server

Server$(\mathsf{EDB}, \mathsf{token}^{(w)}) \to (\mathsf{EDB}', \mathsf{ncnt}^{(w)}, \mathsf{ResultSet})$

1: **parse** $\mathsf{token}^{(w)}$ **as** $(\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)}, \mathsf{nkey}^{(w)})$
2: $\mathsf{ResultSet} \leftarrow \emptyset\,;\quad j \leftarrow 0$
3: **SUEdb**$(\mathsf{EDB}, \mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{nkey}^{(w)}, j, \mathsf{ResultSet}) \to$ $(\mathsf{EDB}, j, \mathsf{ResultSet})$
4: **SUEdb**$(\mathsf{EDB}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)}, \mathsf{nkey}^{(w)}, j, \mathsf{ResultSet}) \to$ $(\mathsf{EDB}, j, \mathsf{ResultSet})$
5: $\mathsf{EDB}' \leftarrow \mathsf{EDB}\,;\quad \mathsf{ncnt}^{(w)} \leftarrow j$
6: **send** $(\mathsf{ncnt}^{(w)}, \mathsf{ResultSet})$ **to** Client

Client$(\mathsf{nkey}^{(w)}, \mathsf{ncnt}^{(w)}) \to \sigma'$

1: $\mathsf{nukey}^{(w)} \xleftarrow{\$} \{0,1\}^{\lambda}$
2: $\mathsf{Dict}'_{\mathsf{kwd}}$ $\leftarrow \mathsf{Dict.Insert}(\mathsf{Dict}_{\mathsf{kwd}}, (w, (\mathsf{nkey}^{(w)}, \mathsf{ncnt}^{(w)}, \mathsf{nukey}^{(w)}, 0)))$
3: $\sigma' \leftarrow (\mathsf{key}_{\mathsf{prf}}, \mathsf{Dict}'_{\mathsf{kwd}})$

Subroutine: **SUEdb**$(\mathsf{EDB}, \mathsf{key}, \mathsf{cnt}, \mathsf{nkey}, j, \mathsf{ResultSet})$

1: **for** $i = 1$ to cnt **do**
2: $\mathsf{label}^{(w)} \leftarrow H_2(\mathsf{key}, i)$
3: **if** $\mathsf{DLDict.Get}(\mathsf{EDB}, \mathsf{label}^{(w)}) = \perp$ **then**
4: **continue**
5: **end if**
6: $j \leftarrow j + 1$
7: $(\mathsf{label}^{(\mathsf{ind})}, \mathsf{data}) \leftarrow \mathsf{DLDict.Get}(\mathsf{EDB}, \mathsf{label}^{(w)})$
8: $\mathsf{ind} \leftarrow \mathsf{data} \oplus H_3(\mathsf{key}, i)$
9: $\mathsf{ResultSet} \leftarrow \mathsf{ResultSet} \cup \{\mathsf{ind}\}$
10: $\mathsf{nlabel}^{(w)} \leftarrow H_2(\mathsf{nkey}, j)\,;\quad \mathsf{ndata} \leftarrow \mathsf{ind} \oplus H_3(\mathsf{nkey}, j)$
11: $\mathsf{EDB} \leftarrow \mathsf{DLDict.Remove}(\mathsf{EDB}, \mathsf{label}^{(w)})$
12: $\mathsf{EDB} \leftarrow \mathsf{DLDict.Insert}(\mathsf{EDB}, (\mathsf{label}^{(\mathsf{ind})}, \mathsf{nlabel}^{(w)}, \mathsf{ndata}))$
13: **end for**
14: **return** $(\mathsf{EDB}, j, \mathsf{ResultSet})$

The delete operation is set as a method of controlling the dual dictionary EDB stored in the server by using a deletion token. This deletion token is the key corresponding to the document added by SE.Addition, and it is generated once and is not affected by other operations. Therefore, in SE.Deletion, the server sequentially rebuilds the label while incrementing the counter, and removes the value corresponding to the label directly from the EDB. This process directly controls EDB, and hence the data removed from the EDB by SE.Deletion do not appear in subsequent searches.

The only issue is collision among the labels generated from H_1 and H_2. We can reduce the correctness to the collision resistance

of H_1 and H_2. Similar to [4], we will set $\mu = 2\lambda$ ($\lambda \geq 2\log N$) in practice because we need to choose μ to guarantee collision resistance where N hash values are revealed.

4 SECURITY PROOF

Theorem 4.1. *Let F be a secure PRF. Then our scheme is $\mathcal{L}$-adaptively-secure in the (programmable) random oracle model, where the leakage function collection $\mathcal{L}$ is defined as follows:*

- $\mathcal{L}_{\mathsf{Setup}}(\lambda) = \emptyset$,
- $\mathcal{L}_{\mathsf{Search}}(w) = (\mathsf{sp}(w), \mathsf{HistDB}(w))$,
- $\mathcal{L}_{\mathsf{Addition}}(\mathsf{ind}, \mathsf{DB}(\mathsf{ind})) = (\mathsf{ind}, |\mathsf{DB}(\mathsf{ind})|)$,
- $\mathcal{L}_{\mathsf{Deletion}}(\mathsf{ind}) = \mathsf{ind}$.

Proof. Given the leakage function collection

$$\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Search}}, \mathcal{L}_{\mathsf{Addition}}, \mathcal{L}_{\mathsf{Deletion}}),$$

we can build a simulator $\mathcal{S}$ as follows:

Random Oracles H. The hash functions $H_i, 1 \leq i \leq 3$ behave like random oracles in the proof. For each $1 \leq i \leq 3$, $\mathcal{S}$ maintains a hash table $H_{i,\mathsf{table}}$ consists of tuples $(\mathsf{counter}, \mathsf{ind}, \mathsf{in}, \mathsf{out}) \in \mathbb{N} \times \{0,1\}^{\ell} \times (\{0,1\}^{\lambda} \times \mathbb{N}) \times \{0,1\}^{\mu}$ as described in Algorithm 5.

Algorithm 5 Programming the Random Oracles(in)

1: **if** $\exists\, (h_{i,1}, h_{i,2}, h_{i,3}, h_{i,4}) \in H_{i,\mathsf{table}}$ such that $h_{i,3} = \mathsf{in}$ **then**
2: **return** $h_{i,4}$
3: **else**
4: $\mathsf{out} \xleftarrow{\$} \{0,1\}^{\mu}$
5: add $(\emptyset, \emptyset, \mathsf{in}, \mathsf{out})$ to $H_{i,\mathsf{table}}$
6: **return** out
7: **end if**

Setup. Simulation of Setup is identical to SE.Setup, the only difference being that, it does not generate a $\mathsf{key}_{\mathsf{prf}}$.

Addition Token. We describe how the simulator $\mathcal{S}$ generates the transcript AddSet when an addition query Addition(f) is issued.

All values $\mathsf{key}^{(\mathsf{ind})}, \mathsf{label}^{(1)}, \mathsf{label}^{(2)}, \mathsf{mask}$ are selected uniformly at random, and appended to the hash tables $H_{i,\mathsf{table}}$ as described in Algorithm 6. The simulator $\mathcal{S}$ outputs AddSet = $\{(\mathsf{label}^{(1)}, \mathsf{label}^{(2)}, \mathsf{mask} \oplus \mathsf{ind})\}$ as the transcript. Note that $\mathcal{S}$ chooses random strings instead of calling F to generate $\mathsf{label}^{(1)}$. If there exists an adversary $\mathcal{B}$ who can distinguish between the real and simulated values $\mathsf{label}^{(1)}$, one can build a reduction that enables us to distinguish between F and a random function. Therefore, the probability to distinguish $\mathsf{label}^{(1)}$ from outputs of a PRF F is bounded by $\mathsf{Adv}^{\mathsf{prf}}_{F,\mathcal{B}}(\lambda)$.

Deletion Token. During the simulation of the addition token, the deletion token $\mathsf{key}^{(\mathsf{ind})}$ corresponding to the document identifier ind has been generated. As described in Algorithm 7, the simulator $\mathcal{S}$ just searches it in $H_{1,\mathsf{table}}$ and sets it as the deletion token for the document with identifier ind.

Algorithm 6 Simulation of the Addition Token(ind, |DB(ind)|)

1: $\mathsf{AddSet} \leftarrow \emptyset$; $\mathsf{key}^{(\mathsf{ind})} \xleftarrow{\$} \{0,1\}^{\lambda}$
2: **for** $i = 1$ **to** $|\mathsf{DB}(\mathsf{ind})|$ **do**
3: **if** $(h_{1,1} = \emptyset, h_{1,2} = \emptyset, (\mathsf{key}^{(\mathsf{ind})}, i), \mathsf{out}) \in H_{1,\mathsf{table}}$ **then**
4: $h_{1,1} \leftarrow i$; $h_{1,2} \leftarrow \mathsf{ind}$
5: **else**
6: $\mathsf{label}^{(1)} \xleftarrow{\$} \{0,1\}^{\mu}$
7: $H_{1,\mathsf{table}} \leftarrow H_{1,\mathsf{table}} \cup (i, \mathsf{ind}, (\mathsf{key}^{(\mathsf{ind})}, i), \mathsf{label}^{(1)})$
8: **end if**
9: $(\mathsf{label}^{(2)}, \mathsf{mask}) \xleftarrow{\$} \{0,1\}^{\mu} \times \{0,1\}^{\mu}$
10: $H_{2,\mathsf{table}} \leftarrow H_{2,\mathsf{table}} \cup (i, \mathsf{ind}, \emptyset, \mathsf{label}^{(2)})$
11: $H_{3,\mathsf{table}} \leftarrow H_{3,\mathsf{table}} \cup (i, \mathsf{ind}, \emptyset, \mathsf{mask})$
12: $\mathsf{AddSet} \leftarrow \mathsf{AddSet} \cup \{(\mathsf{label}^{(1)}, \mathsf{label}^{(2)}, \mathsf{mask} \oplus \mathsf{ind})\}$
13: **end for**
14: **return** AddSet

Algorithm 7 Simulation of the Deletion Token(ind)

1: find any tuple $(h_{i,1}, h_{i,2} = \mathsf{ind}, h_{i,3}, h_{i,4}) \in H_{1,\mathsf{table}}$
2: parse $h_{i,3}$ as $(\mathsf{key}^{(\mathsf{ind})}, *)$
3: **return** $\mathsf{key}^{(\mathsf{ind})}$

Search Token. We first describe how the simulator $\mathcal{S}$ generates the transcript $(\mathsf{key}^{(w)}, \mathsf{cnt}^{(w)}, \mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)}, \mathsf{nkey}^{(w)})$ when a search query Search(w) is issued.

In order to simulate the search token corresponding to an unknown keyword w, the simulator $\mathcal{S}$ first computes $\overline{w} = \min(\mathsf{sp}(w))$ which is used to distinguish different keywords, and then generates random keys $\mathsf{skey}^{(\overline{w})}, \mathsf{nkey}^{(\overline{w})}$.

The random oracles H_2 and H_3 are then programmed as follows. Recall that, while simulating AddSet of (ind, DB(ind)), $\mathcal{S}$ does not specify inputs $(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$ of the random oracle H_2, H_3 and only stores tuples $(i, \mathsf{ind}, \emptyset, \mathsf{label}^{(2)})$ in $H_{2,\mathsf{table}}$ and $(i, \mathsf{ind}, \emptyset, \mathsf{mask})$ in $H_{3,\mathsf{table}}$, because it does not know which keywords are in DB(ind). To reflect the fact that an unknown keyword w is in DB(ind), $\mathcal{S}$ stores tuples $(i, \mathsf{ind}, (\mathsf{skey}^{(\overline{w})}, \mathsf{scnt}^{(\overline{w})}), \mathsf{label}^{(2)})$ in $H_{2,\mathsf{table}}$ and $(i, \mathsf{ind}, (\mathsf{skey}^{(\overline{w})}, \mathsf{scnt}^{(\overline{w})}), \mathsf{mask})$ in $H_{3,\mathsf{table}}$, where $\mathsf{scnt}^{(\overline{w})}$ denotes the corresponding counter for each $\mathsf{ind} \in \mathsf{HistDB}(w)$. On the other hand, it is possible that the adversary $\mathcal{A}$ has already made random oracles queries H_2 or H_3 with inputs $(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$ and got responses $\mathsf{out}_2 = H_2(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$ or $\mathsf{out}_3 = H_3(\mathsf{ukey}^{(w)}, \mathsf{ucnt}^{(w)})$. In this case, if $\mathsf{out}_2 \neq \mathsf{label}^{(2)}$ or $\mathsf{out}_3 \neq \mathsf{mask}$, the simulation fails. Note that the probability that such event occurs is $\mathsf{poly}(\lambda)/2^{\lambda}$, since the simulation is limited to polynomial time in λ.

The simulator $\mathcal{S}$ also maintains a dictionary $\mathsf{Dict}_{\mathsf{kwd}}$ which stores tuples $(\mathsf{key}^{(\overline{w})}, \mathsf{cnt}^{(\overline{w})}, \mathsf{ukey}^{(\overline{w})}, \mathsf{ucnt}^{(\overline{w})})$ for $\overline{w} \in \mathbb{N}$ as the proposed algorithm. The only difference is that the simulator $\mathcal{S}$ updates $\mathsf{Dict}_{\mathsf{kwd}}$ only when simulating search queries, not addition queries. If $\mathsf{Dict.Get}(\mathsf{Dict}_{\mathsf{kwd}}, \overline{w}) = \perp$ and $\mathsf{scnt}^{(\overline{w})} = 0$ (or equivalently $|\mathsf{sp}(w)| = 1$), it means that the search query for w is issued for the first time and so $\mathcal{S}$ outputs the transcript $(\emptyset, 0, \mathsf{skey}^{(\overline{w})}, \mathsf{scnt}^{(\overline{w})}, \mathsf{nkey}^{(\overline{w})})$. Otherwise, since the search query for w has been

issued before, $\mathcal{S}$ outputs the transcript $(\mathsf{key}^{(\overline{w})}, \mathsf{cnt}^{(\overline{w})}, \mathsf{skey}^{(\overline{w})}, \mathsf{scnt}^{(\overline{w})}, \mathsf{nkey}^{(\overline{w})})$ for $(\mathsf{key}^{(\overline{w})}, \mathsf{cnt}^{(\overline{w})}, *, *) \leftarrow \mathsf{Dict.Get}(\mathsf{Dict}_{\mathsf{kwd}}, \overline{w})$.

The search result $\mathsf{DB}(w)$ can be constructed from $\mathsf{HistDB}(w)$ and document identifiers revealed during the deletion token simulation. The rest parts of the protocol, including the re-encryption of the EDB and the updating $\mathsf{Dict}_{\mathsf{kwd}}$ can be simulated similarly because all the necessary inputs are given to $\mathcal{S}$, and thus we do not described in detail here.

Algorithm 8 Simulation of the Search Token($\mathsf{sp}(w)$, $\mathsf{HistDB}(w)$)

$\overline{w} \leftarrow \min(\mathsf{sp}(w))$
$\mathsf{nkey}^{(\overline{w})} \xleftarrow{\$} \{0,1\}^\lambda$; $\mathsf{skey}^{(\overline{w})} \xleftarrow{\$} \{0,1\}^\lambda$; $\mathsf{scnt}^{(\overline{w})} \leftarrow 0$
for ind $\in \mathsf{HistDB}(w)$ (choose sequentially) **do**
  $\mathsf{scnt}^{(\overline{w})} \leftarrow \mathsf{scnt}^{(\overline{w})} + 1$
  if $(\emptyset, \emptyset, (\mathsf{nkey}^{(\overline{w})}, \mathsf{scnt}^{(\overline{w})}), *) \in H_2$ or H_3 **then**
    abort
  else
    choose one element of the form $(h_{2,1}, \mathsf{ind}, h_{2,3} = \emptyset, *)$ randomly from $H_{2,\mathsf{table}}$
    choose one element of the form $(h_{3,1} = h_{2,1}, \mathsf{ind}, h_{3,3} = \emptyset, *)$ randomly from $H_{3,\mathsf{table}}$
    $h_{2,3} \leftarrow (\mathsf{skey}^{(\overline{w})}, \mathsf{scnt}^{(\overline{w})})$; $h_{3,3} \leftarrow (\mathsf{skey}^{(\overline{w})}, \mathsf{scnt}^{(\overline{w})})$
    $\mathsf{HistDB}(w) \leftarrow \mathsf{HistDB}(w) \setminus \{\mathsf{ind}\}$
  end if
end for
if $\mathsf{Dict.Get}(\mathsf{Dict}_{\mathsf{kwd}}, \overline{w}) = \perp$ and $\mathsf{scnt}^{(\overline{w})} = 0$ **then**
  return $\emptyset$
end if
if $\mathsf{Dict.Get}(\mathsf{Dict}_{\mathsf{kwd}}, \overline{w}) = \perp$ **then**
  $\mathsf{token}^{(\overline{w})} \leftarrow (\emptyset, 0, \mathsf{skey}^{(\overline{w})}, \mathsf{scnt}^{(\overline{w})}, \mathsf{nkey}^{(\overline{w})})$
else
  $(\mathsf{key}^{(\overline{w})}, \mathsf{cnt}^{(\overline{w})}, *, *) \leftarrow \mathsf{Dict.Get}(\mathsf{Dict}_{\mathsf{kwd}}, \overline{w})$
  $\mathsf{token}^{(\overline{w})} \leftarrow (\mathsf{key}^{(\overline{w})}, \mathsf{cnt}^{(\overline{w})}, \mathsf{skey}^{(\overline{w})}, \mathsf{scnt}^{(\overline{w})}, \mathsf{nkey}^{(\overline{w})})$
end if
return $\mathsf{token}^{(w)} \leftarrow \mathsf{token}^{(\overline{w})}$

Conclusion. By combining all simulation results, we can say that, for any probabilistic polynomial-time adversary $\mathcal{A}$, there exists a prf-adversary $\mathcal{B}$ such that

$$\left|\Pr[\mathsf{Game}_{R,\mathcal{A}}(\lambda) = 1] - \Pr[\mathsf{Game}_{\mathcal{S},\mathcal{A}}(\lambda) = 1]\right| \leq \mathsf{Adv}^{\mathsf{prf}}_{F,\mathcal{B}}(\lambda) + \mathsf{poly}(\lambda)/2^\lambda.$$

We thus conclude the resulting probability is $\mathsf{negl}(\lambda)$ by assuming that the PRF F is secure. Note that our construction provides *forward security* because $\mathcal{L}_{\mathsf{Addition}}$ leaks only $(\mathsf{ind}, |\mathsf{DB}(\mathsf{ind})|)$, thus $\bar{\mathcal{L}}$ can be defined as $\mathcal{L}_{\mathsf{Addition}}$ itself.

□

5 SOME CONSIDERATIONS

5.1 Leakage Comparison with Previous Schemes

In this section, we analyze the leakage level of our scheme compared to the previous forward secure DSSE schemes [4, 31]. The leakage functions of the scheme in [31] are defined as

$$\mathcal{L}^{\mathsf{SPS}} = (\mathcal{L}^{\mathsf{SPS}}_{\mathsf{Addition}}, \mathcal{L}^{\mathsf{SPS}}_{\mathsf{Deletion}}, \mathcal{L}^{\mathsf{SPS}}_{\mathsf{Search}}),$$

where

- $\mathcal{L}^{\mathsf{SPS}}_{\mathsf{Addition}}(\mathsf{ind}, \mathsf{DB}(\mathsf{ind})) = (\mathsf{ind}, |\mathsf{DB}(\mathsf{ind})|)$,
- $\mathcal{L}^{\mathsf{SPS}}_{\mathsf{Deletion}}(\mathsf{ind}, \mathsf{DB}(\mathsf{ind})) = (\mathsf{ind}, |\mathsf{DB}(\mathsf{ind})|)$,
- $\mathcal{L}^{\mathsf{SPS}}_{\mathsf{Search}}(w) = (\mathsf{sp}(w), \mathsf{HistDB}(w))$.

The addition and search leakage functions are exactly the same as ours. Although there is a slight difference in the deletion leakage due to the difference in the way the delete operations are performed, it does not affect overall update leakage level.

The leakage functions of Sophos are defined as

$$\mathcal{L}^{\mathsf{B}} = (\mathcal{L}^{\mathsf{B}}_{\mathsf{Addition}}, \mathcal{L}^{\mathsf{B}}_{\mathsf{Deletion}}, \mathcal{L}^{\mathsf{B}}_{\mathsf{Search}}),$$

where

- $\mathcal{L}^{\mathsf{B}}_{\mathsf{Addition}}(\mathsf{ind}, w) = \mathcal{L}^{\mathsf{B}}_{\mathsf{Deletion}}(\mathsf{ind}, w) = \perp$,
- $\mathcal{L}^{\mathsf{B}}_{\mathsf{Search}}(w) = (\mathsf{sp}(w), \mathsf{Hist}(w))$

and $\mathsf{Hist}(w)$ lists all the modifications made to $\mathsf{DB}(w)$. For example, $(2, \mathsf{Deletion}, \mathsf{ind}) \in \mathsf{Hist}(w)$ means the pair (ind, w) was deleted at the second update. Note that Sophos is designed to handle a pair (ind, w) of inputs. According to the security analysis in [4], $\mathcal{L}^{\mathsf{B}}$ is comparable with $\mathcal{L}^{\mathsf{SPS}}$ if Sophos is limited to only support the update of a whole document $(\mathsf{ind}, \mathsf{DB}(\mathsf{ind}))$ not a single pair (ind, w) where $w \in \mathsf{DB}(\mathsf{ind})$. More specifically, they showed that $\mathsf{Hist}(w)$ can be constructed from $\mathsf{HistDB}(w)$ and the time information of update queries. Now, the only possible difference compared to our scheme is that the update function of Sophos reveals nothing (even ind) when a single (ind, w) pair is updated while our scheme leaks $(\mathsf{ind}, |\mathsf{DB}(\mathsf{ind})|)$ when all (ind, w) pairs in the document are updated. From this, one might think that our scheme reveals more information than Sophos. However, revealing the document identifier in the document-level update is inevitable except ORAM-based schemes because the document also needs to be added or deleted when the update operation is performed.

5.2 Security against Malicious Adversaries

The security model (Section 2.4) considers only passive adversaries who will not deviate from the defined protocol, but will attempt to learn all possible information from the protocol transcripts. Thus our scheme does not guarantee security against malicious adversaries that can return incorrect answers to the clients. To remedy this problem, we can consider the generic solutions for constructing *verifiable* DSSE scheme [5]. The notion of verifiability enables a client to verify that the results by search queries are indeed correct. In [5], the authors showed how to add "verifiability" to the non-verifiable scheme in [31] while keeping the performance. Basic idea is to maintain the hash values of $\mathsf{DB}(w)$ for each keyword w, and then to use these values to check the validity of search results. In more detail, they applied the *multiset hashing* technique [12] to avoid re-hash the full set of $\mathsf{DB}(w)$ every time updating for w is performed. By applying their result, we can easily get the verifiable version of our scheme without any loss in asymptotic performance.

5.3 Easy Deletion

The delete operation described in Algorithm 3 requires only an identifier of the document to be deleted as a client's input and the computation and communication costs for generating and sending the deletion token are only $O(1)$. However, since the previous forward secure DSSE schemes perform the deletion in the same way as the addition, at least $O(|\mathrm{DB(ind)}|)$ computation and communication costs are required. It is also unrealistic for the client to know all keywords contained in the document to be deleted in advance. Furthermore, if the delete operation is repeated for the same document, meaningless data are continually added to the EDB unless DBMS prevents it. Note that our scheme does not suffer from this problem.

6 EVALUATION ON A SMALL DATASET

The evaluation of our scheme targets the performance of each operation and changes in storage space. To evaluate our scheme, we wrote a code with C++14 that works standalone on Windows and POSIX. Because we are interested in measuring the latency to see how fast the operations are performing, our code is designed to run as a single program without RPC (Remote procedure call) or separate DBMS. Both search and update operations are implemented so that they can be parallelized using a thread pool. Although our program does not have a network communication function, it is designed to be configured as a server-client environment through RPC.

We chose LSH [25] as the underlying hash function to instantiate the PRF F as HMAC, and also used it to instantiate the hash functions H_i $(i = 1, 2, 3)$.

And we use RocksDB [17] as the data structure of our scheme. RocksDB is optimized for random access devices such as SSD, and thus is considered to be suitable as a client and server data structures. Since a dual dictionary has two keys, we implemented it to input two objects in RocksDB. Note that one object has a form of (`document label`, `keyword label`) pair, which serves as the forward index, and the other object has a form of (`keyword label`, (`document label`, `encrypted value`)) tuple that serves as the inverted index. The label uses the upper 128 bits of the hash digest (H_1, H_2), and the encrypted value is a 64-bit value of the result of bitwise xor of the document ID and the hash digest (H_3).

All our experiments were performed on a desktop computer with a single Intel Core i7 4790 3.6 GHz CPU, 16 GB of DDR3 RAM, and Samsung SSD 850 PRO 512 GB running Linux Mint 18 (64-bit). For comparison, we used the open source code of Sophos in GitLab and added a code that implements the search and update that do not use the RPC. To minimize I/O access time all files used in our experiments are loaded into main memory before starting measurements. This is done by activating the memory-mapped file read (`mmap read`) function in RocksDB.

6.1 Dataset

We used the well-known Enron email dataset [15] to create an EDB. The Enron email dataset is a dataset that converts the contents of the mail server to text and is provided as 517,491 plaintext files of 1.32 GiB when decompressed. We used Python to write a code that removes embedded attachments from text files in BASE64 encoding and removes unnecessary special characters, HTML tags, and so on. We used the PorterStemmer provided by the NLTK library [29] to extract keywords from the original text. The extracted keyword count is 400,087, and the total number of document/keyword pairs is 62,018,878.

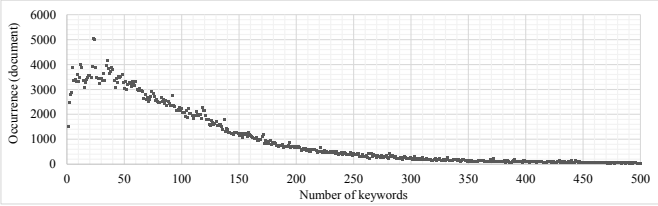

Figure 4: Occurrence of documents by keyword count

Figure 4 depicts the frequency of documents according to the number of keywords in the forward index view. Most documents have fewer than 300 keywords and have a distribution similar to the log-normal distribution.

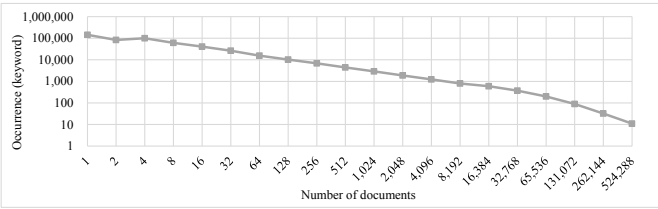

Figure 5: Occurrence of keywords by document count

Figure 5 depicts the frequency of keywords according to the number of documents in terms of inverted index. For example, the number of keywords with only one search result is 143,992. In addition, it can be inferred that a small number of documents are returned as a result of the search operation for most keywords.

6.2 EDB Creation

The initial EDB was created to store the encrypted contents of the Enron email dataset. Our scheme uses document-level (forward) input, and we measured the time of completion of EDB creation by bulk-insertion three times to obtain the average. The computational works and I/O latency required for the insertion process are parallelized using the thread pool. For Sophos, it provides a document/keyword pair input function to update an EDB, and provides an example of keyword-level input for bulk-insertion. For consistent experimentation, we added a code to allow document-level insertion for EDB creation, and it is parallelized and optimized to work in multi-threaded mode. We also tried to experiment keyword-level bulk-insertion as used in Sophos code. Table 3 shows the time taken to input all documents in the Enron email dataset and the storage capacity just after EDB creation.

To understand the effect of the underlying hash function on the speed of our scheme, we measured the operation speed from two implementations, one using LSH and the other using SHA of CRYPTO++ [14]. For reference, LSH-256 was 7.76 cycles/byte,

Table 3: Comparison with EDB creation using Enron email dataset

Implementation	Time (ms)	Pairs per sec.	Storage (KiB)	
			Client	Server
Our scheme (with LSH-256)	451,824	137,263.4	86,093	5,243,625
Our scheme (with SHA-256)	469,039	132,225.3	86,089	5,245,237
Sophos (with RSA-2048, Document-level)	9,494,200	6,532.3	272,360	2,242,700
Sophos (with RSA-2048, Keyword-level)	9,628,816	6,441.0	272,364	2,241,436
Sophos (with RSA-512, Document-level)	1,085,146	57,152.6	46,453	2,242,712

and SHA-256 was 25.03 cycles/byte in the evaluation environment. An interesting point is that the time difference between the LSH based and the SHA based implementations is negligible. Given the speed difference between the two hash functions as noted above, the implementation appears to be heavily influenced by the I/O bottleneck.

In the case of the Sophos implementations, the insertion rate due to RPC removal is faster than 4,300 pairs per second presented in [4], but still more inefficient than our scheme. We estimate that the RSA decryption operation used in Sophos is a more delayed factor in the EDB generation than I/O latency. As a basis for this estimation, we implemented Sophos using 512-bit RSA instead of 2,048 bits, and the performance is dramatically improved as shown in Table 3.

From Table 3, we can see that our scheme has less CPU load and less client storage space. Considering that the addition operation including EDB generation is performed at the client, even if the performance of the client is not sufficient, the operation can be sufficiently performed by using our scheme.

6.3 EDB Search

To compare the search speed just after EDB creation, we searched all the keywords extracted from the Enron email dataset and measured the time taken.

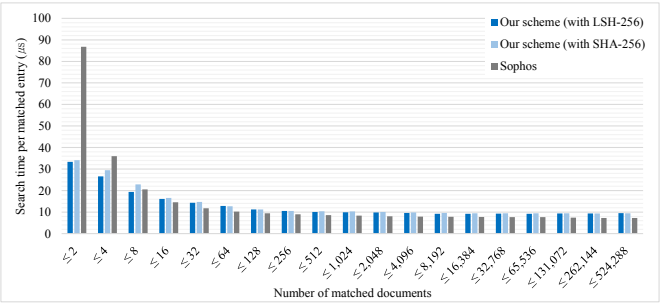

Figure 6: Comparison with search time per matched document

Figure 6 compares the average time taken to search based on the number of documents returned in the search results. Here, by average time we mean the time taken to search divided by the number of matched documents. Both implementations of our scheme show that the I/O latency is the bottleneck as the EDB creation. When compared with Sophos, however, our scheme is superior in speed for the case of a small number of matched documents, but the efficiency is relatively low on other cases. This is because our scheme deletes old labels and adds new labels in the dual dictionary each time it is searched, thereby increasing the load of I/O.

6.4 EDB Update

To evaluate the effect of update operations to the EDB, we randomly chose 200,000 documents from the Enron mail dataset to create an EDB and used remained 300,000 documents for the update. We measured the writing and deleting time by repeating the addition and deletion operations, and measured the searching time in units of 200,000 times. Note that Sophos does not provide an explicit deletion operation, but instead recommends creating and maintaining a separate EDB for deletion. For Sophos, we therefore created two EDB instances for storing added document/keyword pairs and deleted pairs respectively and implemented the procedure for a search query by computing the difference between the matched documents in both EDB instances.

Table 4: Comparison of operation performance during add-delete-search iterations (unit: pairs per sec.)

Iteration	Our scheme			Sophos		
	Add	Delete	Search	Add	Delete	Search
Init.	132,870	-	91,265	6,898	-	98,315
200k	124,941	117,406	71,140	6,903	6,968	37,825
400k	127,804	118,227	72,910	6,903	6,915	22,461
600k	130,321	117,977	71,620	6,924	6,934	17,153
800k	127,631	118,150	72,683	6,953	6,960	13,479

6.4.1 Storage Capacity. Based on the EDB generated using the Enron email dataset, we checked the change in capacity by repeatedly adding and deleting 10 randomly chosen documents from the dataset. Figure 7 shows the results. Note that RocksDB maintains a certain amount of space after deleting data for performance. Although the data structure of our scheme keeps the amount of data unchanged in this evaluation, the capacity can be increased partially due to the empty space caused by the repeated addition and

deletion operations. Specifically, our scheme makes the capacity of the EDB start from 2.0 GiB and increases to 2.5 GiB. In the case of Sophos, it can be seen that the capacity increases linearly in the evaluation process.

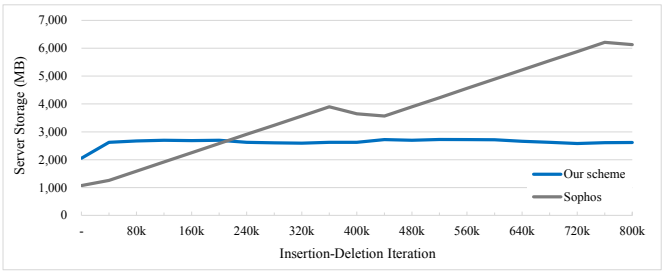

Figure 7: Server storage usage by repeated addition-deletion operations

6.4.2 Searching Time. Similar to the experiment for the search efficiency in Section 6.3, we searched all the keywords extracted from the Enron email dataset and measured the average value for three repeated iterations of the experiments. Each entry in the "Search" column of Table 4 is the number of matched pairs in the operation divided by the total time spent. And each entry in the "Add" and "Delete" columns is the number of document/keyword pairs processed per one second in the operation.

Our scheme requires only one iteration of the search operation for the deleted document/keyword pair, and thus it can be seen that the search speed is decreased in the subsequent search compared to the search speed measured on the initial EDB. However, since the index of the deleted document/keyword pair is strictly removed by the additional one-time search procedure, it can be confirmed that the search speed is not influenced later. In the case of Sophos, deletion performs the same operation as addition, and thus shows poor performance as, noted in Section 6.3. Also, since all the deleted data must be searched in the search query processing, the search speed gradually decreases. These results show that our scheme is feasible without degrading capacity and performance in environments where updates are frequent.

7 EVALUATION ON A LARGE DATASET

In Section 6, we evaluate performance of our scheme on the Enron email dataset. To check performance in a more realistic situation, however, we further evaluate performance of our scheme on Wikipedia. Due to time constraints, this evaluation is limited to the EDB creation and searching on the initial EDB, and no performance changes during the update are measured.

The data structure and data configuration based on RocksDB, coding tools and techniques, and the adopted cryptographic algorithms are the same as those for the Enron email dataset experiments. Considering the size of Wikipedia, however, all our experiments were performed on a desktop computer with a single Intel Core i7 6850K 3.6 GHz CPU, 128 GB of DDR4 RAM, and Samsung SSD 850 PRO 512 GB running Linux Mint 18 (64-bit).

7.1 Dataset and EDB Creation

We used the Wikipedia Extractor [2] to remove the wiki syntax from entire Wikipedia articles[2] [18] and used NLTK to extract keywords from the original texts. The size of the uncompressed xml files is 52.4 GiB, and the size of the plaintexts with the wiki syntax removed was 11.8 GiB. The number of extracted keywords was 4,225,457, the number of documents is 5,078,194, and the total number of document/keyword pairs is 698,371,776, which is about 10 times of those of Enron email dataset.

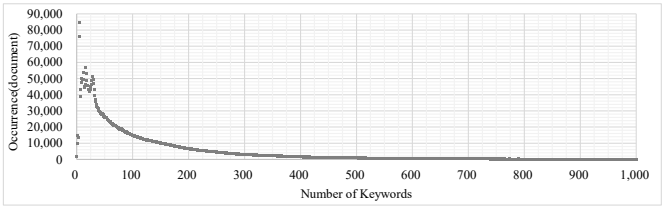

Figure 8: Occurrence of documents by keyword count

Figure 8 depicts the frequency of documents according to the number of keywords in the forward index view.

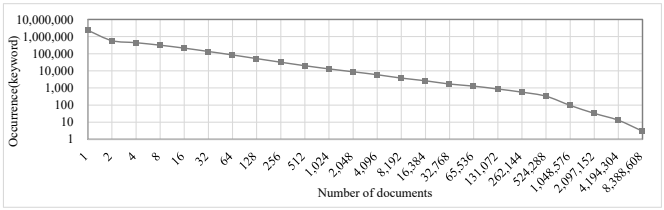

Figure 9: Occurrence of keywords by document count

Figure 9 depicts the frequency of keywords according to the number of documents in terms of inverted index.

To create an EDB based on the Wikipedia dataset, both our scheme and Sophos use document-level input and we measured the time of completion of EDB generation by bulk-insertion. Table 5 shows the time taken to input all documents in the Wikipedia dataset and the storage capacity just after EDB creation.

Table 5: Comparison with EDB creation using Wikipedia dataset

Scheme	Time (ms)	Pairs per sec.	Storage (KiB)	
			Client	Server
Ours	5,268,861	132,547.0	738,524	65,950,212
Sophos	64,456,708	10,834.7	2,198,504	20,954,360

Especially, it is confirmed that the speed of EDB generation of Sophos increased about 50% compared to the case for Enron email dataset (See Table 3). Considering the performance characteristic

[2] `enwiki-20160204-pages-articles.xml.bz2`

of the addition operation of Sophos which strongly depends on the CPU performance, it is estimated to be influenced by the 6-core CPU instead of the 4-core CPU used for the Enron email dataset experiments.

Anyway, as observed in Section 6.2, we can see that our scheme has still less CPU load and less client storage space. This implies that the addition operation can be sufficiently performed by using our scheme even if the performance of the client is not sufficient and the size of a dataset is big.

7.2 EDB Search

Note that the EDB based on the Wikipedia dataset has a very large capacity compared to the Enron email dataset. In the case of such large-scale data, it may not be possible to load and handle the entire EDB in memory. Therefore, we measured the search performance in two cases when using all 128 GB RAM that can hold the entire EDB and when using only 16 GB RAM. To configure the 16 GB RAM environment, we added `MEM=16384M` command to the GRUB bootloader. The result is depicted in Figure 10.

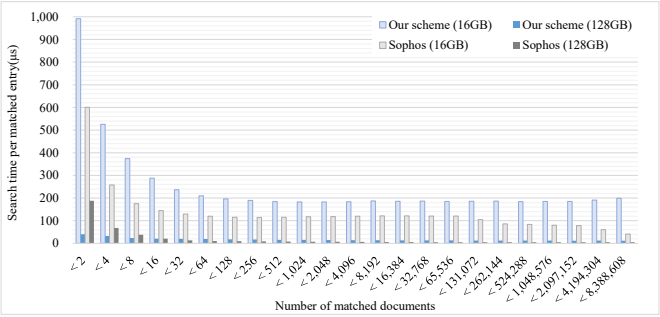

Figure 10: Comparison with search time per matched documents on 128 GB and 16 GB RAM environments

When the entire EDB is loaded in the memory, as observed in Section 6.3, our scheme is superior in speed for the case of a small number of matched documents but the efficiency is relatively low on other cases. When only 16 GB RAM can be used, however, thrasing occurs because the capacity of the EDB is much larger than memory size. This has caused serious performance degradation, and the performance of Sophos with smaller EDB size has also been significantly reduced although relatively less.

It is known that random read/write speed of SSD can achieve optimal performance when issuing more than 32 instructions in parallel [16]. Since our scheme is designed to be optimized for parallel implementation, we evaluate the performance of our scheme in the environment where `mmap read` function is disabled and the SSD is used to read and write directly. Since our scheme has already been optimized to be well operated at relatively slow storages, no additional configuration was required except for `mmap read` function disabling. In this case, our scheme shows a big performance improvement compared to the result of 16 GB RAM environment depicted in Figure 10. If the number of search results is less than 32, SSD seems not to be utilized effectively. For 32 or more search results, the best performance is shown, which is similar to the case of full memory load as shown in Figure 11.

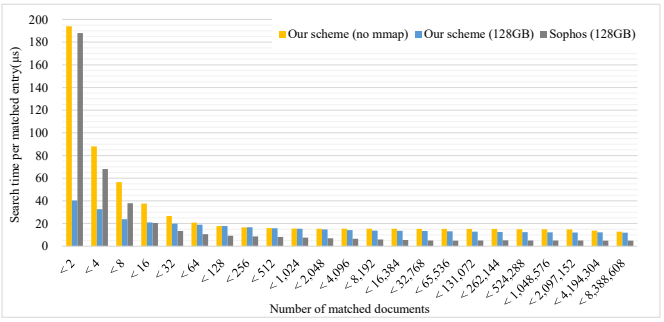

Figure 11: Comparison with search time per matched documents on 128 GB and nommap environments

As stated above, due to the difference in size of the initial EDBs, our scheme requires much more work in the search operation. Nevertheless, for the cases of the entire EDB loaded in the memory, our scheme does not show significantly less performance compared to Sophos. And our scheme shows similar performance when the entire EDB is loaded in the memory or `mmap read` function is disabled. Furthermore, it is expected that the change of the storage capacity due to frequent update operations and the resulting change in the search speed will be the same as the result in Section 6.4. Those imply that our scheme can be fully utilized in large datasets.

ACKNOWLEDGEMENT

We are grateful to anonymous reviewers of the CCS 2017 for very detailed and thoughtful reviews. We would also like to thank our shepherd Muhammad Naveed for his advice and help. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean government (MSIT) (No.R0101-16-0301).

REFERENCES

[1] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. 2016. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing (STOC '16)*. ACM, New York, NY, USA, 1101–1114. https://doi.org/10.1145/2897518.2897562

[2] Giuseppe Attardi. 2016. Wikipedia Extractor. (2016). http://medialab.di.unipi.it/wiki/Wikipedia_Extractor

[3] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. 2014. A survey of provably secure searchable encryption. *ACM Comput. Surv.* 47, 2, Article 18 (Aug. 2014), 51 pages. https://doi.org/10.1145/2636328

[4] Raphael Bost. 2016. $\Sigma o \phi o \varsigma$ - Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1143–1154. https://doi.org/10.1145/2976749.2978303

[5] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. 2016. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. Cryptology ePrint Archive, Report 2016/062. (2016). http://eprint.iacr.org/2016/062

[6] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 668–679. https://doi.org/10.1145/2810103.2813700

[7] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2014. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium (NDSS'14)*. Internet Society, Reston, VA, U.S.A, 23–26.

[8] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2013. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology – CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Springer Berlin Heidelberg, Berlin, Heidelberg, 353–373. https://doi.org/10.1007/978-3-642-40041-4_20

[9] David Cash and Stefano Tessaro. 2014. The locality of searchable symmetric encryption. In *Advances in Cryptology – EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, Phong Q. Nguyen and Elisabeth Oswald (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–368. https://doi.org/10.1007/978-3-642-55220-5_20

[10] Yan-Cheng Chang and Michael Mitzenmacher. 2005. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security: Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005. Proceedings*, John Ioannidis, Angelos Keromytis, and Moti Yung (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 442–455. https://doi.org/10.1007/11496137_30

[11] Melissa Chase and Seny Kamara. 2010. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, Masayuki Abe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 577–594. https://doi.org/10.1007/978-3-642-17373-8_33

[12] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. 2003. Incremental multiset hash functions and their application to memory integrity checking. In *Advances in Cryptology - ASIACRYPT 2003: 9th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, November 30 – December 4, 2003. Proceedings*, Chi-Sung Laih (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 188–207. https://doi.org/10.1007/978-3-540-40061-5_12

[13] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/1180405.1180417

[14] Wei Dai. 2016. Crypto++ Library 5.6.5. (2016). https://www.cryptopp.com

[15] Enron Email Dataset. 2015. (2015). https://www.cs.cmu.edu/~./enron

[16] Samsung Electronics. 2013. Samsung Solid State Drive White Paper. (2013). http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/850pro.html

[17] Facebook. 2016. RocksDB: A persistent key-value store for fast storage environment. (2016). http://rocksdb.org

[18] Wikimedia Foundation. 2016. Wikimedia Downloads. (2016). Retrieved February 4, 2016 from https://dumps.wikimedia.org

[19] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In *Advances in Cryptology – CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, Matthew Robshaw and Jonathan Katz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 563–592. https://doi.org/10.1007/978-3-662-53015-3_20

[20] Shafi Goldwasser and Mihir Bellare. 2008. *Lecture Notes on Cryptography*. https://cseweb.ucsd.edu/~mihir/papers/gb.html

[21] Florian Hahn and Florian Kerschbaum. 2014. Searchable encryption with secure and efficient updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 310–320. https://doi.org/10.1145/2660267.2660297

[22] Mohammad Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proceedings of the 2012 Network and Distributed System Security (NDSS) Symposium*. Internet Society, Reston, VA, U.S.A.

[23] Seny Kamara and Charalampos Papamanthou. 2013. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, Ahmad-Reza Sadeghi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 258–274. https://doi.org/10.1007/978-3-642-39884-1_22

[24] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 965–976. https://doi.org/10.1145/2382196.2382298

[25] Dong-Chan Kim, Deukjo Hong, Jung-Keun Lee, Woo-Hwan Kim, and Daesung Kwon. 2015. LSH: A new fast secure hash function family. In *Information Security and Cryptology - ICISC 2014: 17th International Conference, Seoul, South Korea, December 3-5, 2014, Revised Selected Papers*, Jooyoung Lee and Jongsung Kim (Eds.). Springer International Publishing, Cham, 286–313. https://doi.org/10.1007/978-3-319-15943-0_18

[26] Muhammad Naveed. 2015. The fallacy of composition of oblivious RAM and searchable encryption. Cryptology ePrint Archive, Report 2015/668. (2015). http://eprint.iacr.org/2015/668

[27] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. 2014. Dynamic searchable encryption via blind storage. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. 639–654. https://doi.org/10.1109/SP.2014.47

[28] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. 2014. Blind Seer: A scalable private DBMS. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. 359–374. https://doi.org/10.1109/SP.2014.30

[29] NLTK Project. 2016. Natural Language Toolkit. (2016). http://www.nltk.org

[30] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (SP '00)*. IEEE Computer Society, Washington, DC, USA, 44–55. http://dl.acm.org/citation.cfm?id=882494.884426

[31] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical dynamic searchable encryption with small leakage. In *Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium*. Internet Society, Reston, VA, U.S.A.

[32] Attila A. Yavuz and Jorge Guajardo. 2016. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In *Selected Areas in Cryptography – SAC 2015: 22nd International Conference, Sackville, NB, Canada, August 12–14, 2015, Revised Selected Papers*, Orr Dunkelman and Liam Keliher (Eds.). Springer International Publishing, Cham, 241–259. https://doi.org/10.1007/978-3-319-31301-6_15

[33] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 707–720. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang