2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications

Encrypted Keyword Search Mechanism Based on Bitmap Index for Personal Storage Services

Yong Ho Hwang, Jae Woo Seo, and Il Joo Kim
Software R&D Center, Samsung Electronics Co., LTD,
Suwon-si, Gyeonggi-do 443-742, Republic of Korea
Email: {yongh.hwang, jaewoo13.seo, ij00.kim}@samsung.com

***Abstract*—The remote storage service has been one of the most popular cloud services. However, outsourcing in the remote storage causes a privacy issue such as the exposure of personal data and the leakage of private information. In this paper, we focus on the encrypted keyword search problem, called searchable symmetric encryption in cryptography, to preserve the user privacy and the data confidentiality in cloud storage services. Even though many works have been introduced, they still suffer from limitations in their practical use due to high search costs and huge index size. We present a new keyword search mechanism based on the bitmap index that represents a set of files in the inverted index form. The proposed mechanism has fast search time and small index size compared with the previous works in practical use. We show that it can be directly applied to practical services by testing the known NoSQL database, Couchbase.**

I. Introduction

Outsourcing data to remote server systems such as public or private cloud storage services is a major industry trend. In the market, cloud services providing remote storage (e.g., DropBox, SkyDrive and iCloud) are already popular and many people use them. The privacy is realistic and sensitive issue for these services. As the privacy-preserving solution, we can consider the encryption of files in the cloud storages. However, normal encryption could not support the way to efficiently search them over encrypted files. The searching problem over encrypted files has been considered the main issue in (remote) storage services. In this paper, we present a new mechanism for the encrypted keyword search in personal storage services, called searchable symmetric encryption (SSE) in cryptography; the adversaries (e.g., honest-but-curious server and outsider attackers) can get very restricted information such as the file's sizes, the user's access and search patterns, and the encrypted files, but cannot learn any information about file's contents.

In the literature of SSE, Song et al. first proposed a non-interactive solution with linear search time on the total number of files [1]. After that, many research works have been presented [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. However, some of them do not address addition and deletion of files in cloud storage. We are only interested in *dynamic* SSE supporting addtion and deletion [5], [6], [7], [9], [10], [11]. The dynamic SSE is classified into two types according to their index structures; *forward index* [5], [6] and *inverted index* [7], [9], [10], [11]. In the forward index form, the files have their keywords as indices while, in the inverted index form, the unique keywords have the associated files as indices. In general, the schemes based on the inverted index form take shorter time than those based on the forward index structure for search, but requires more complex methods for addition and deletion. The schemes in [9], [10] are interactive for supporting dynamic updates (i.e., addition and deletion) with small leakage; the interactive dynamic SSE is beyond the scope of our work. The dynamic SSE of [7] is based on the linked list data structure and its searching complexity[1] is known as theoretically optimal. The scheme of [11] provides the same searching complexity as that of [7] with more restricted leakage information, which is based on the dictionary structure using counter. However, they still suffer from high search costs and huge index size in their practical use for personal storage services; to find the files with a searching keyword, they should perform decryption-related operations up to the number of files with the searching keyword while our proposal performs only one operation.

For practical use, we propose a new dynamic SSE based on the bitmap index that represents a set of files in the inverted index form. It is secure against the adaptive chosen keyword attacks of [7] in the random oracle model, which is the strong security requirement in the field of SSE. We call our mechanism bitmap-based SSE (B-SSE). The contributions of the proposed mechanism are as follows:

Efficiency. B-SSE achieves fast search time and small index size (see Section IV-A) by using a new indexing technique, bitmap index, for encrypted keyword search. While the bitmap indexing technique is quite simple, it satisfies strong security requirements and shows much better performance than the previous works in commonly-used cases of practical environment.[2]

Practical Test. We implement B-SSE on Couchbase, one of the most popular NoSQL database (see Section IV-B). In the field of SSE, several systems have been introduced but they only provide a methodology or test results in their local desktop; it could make developers have doubts about their practical usage. Our test gives clear criteria for using B-SSE.

II. Preliminary

A. Notation

We define $\mathbf{F}$ as the set of total files and $\mathbf{F}(w)$ as the set of files containing a keyword w. Similarly, $\mathbf{W}$ denote the set of unique keywords and $\mathbf{W}(f)$ be the set of unique keywords contained in a file f. For the number of elements of these sets, let $\mathbf{f}$ denote the size of $\mathbf{F}$, $\mathbf{f}_w$ be the size of $\mathbf{F}(w)$, $\mathbf{w}$ be the size of $\mathbf{W}$ and $\mathbf{w}_f$ be the size of $\mathbf{W}(f)$. We express the encryption of $\mathbf{F}$ and $\mathbf{F}(w)$ into $\mathbf{C}$ and $\mathbf{C}(w)$, respectively.

We write $\mathsf{id}(a)$ to indicate the identifier of a file or a keyword a, and $\mathsf{id}(A)$ to represent the set of identifiers of

[1] $O(\mathbf{f}_w)$ where $\mathbf{f}_w$ is the number of files with the search keyword

[2] In the specific cases that the number of files with the same keyword is very low, our scheme could have worse performance than the previous works. However, it is not common situation in personal storage services.

978-1-4799-6513-7/14 $31.00 © 2014 IEEE
DOI 10.1109/TrustCom.2014.22

elements composing a set A. The symbol $\{\ldots\}$ is a set representation and $[n]$ means the set of integers $\{1,\ldots,n\}$. $|a|$ means the length of a bit string a and $a||b$ is the concatenation of strings a and b. We use a simple table that consists of two column, $(key, value)$. The key is used to find the associated $value$ in the table; for a table T, $T[key]$ is the $value$ associated with the key.

B. Bitmap Index

We use a bitmap to represent the set of file identifiers. Our bitmap-based technique maps distinct elements of a set on each bit position of a bit array. For example, let $A = \{a_1,\ldots,a_m\}$ and there is a m-bit array where the length of the bit array m is determined by the number of the elements of the set A. Then, the i-th bit position in the m-bit array is associated with the element a_i and a subset $\tilde{A}(\subseteq A)$ is represented by setting the i-th bit position as 1 for all the integers i such that $a_i \in \tilde{A}$; the other bits are set as 0. We describe the m-bit array for the subset $\tilde{A}$ as $\mathsf{B_m}(\tilde{A})$ and $\mathsf{B_m}(\tilde{A})[i]$ for $1 \le i \le m$ means the value (0 or 1) of its i-th bit position.

When adding a new element $a_i \in A$, the i-th bit position of the m-bit array, $\mathsf{B_m}(\tilde{A})[i]$, is set as 1. When deleting an existing element $a_j \in \tilde{A}$, the j-th bit position of the m-bit array, $\mathsf{B_m}(\tilde{A})[j]$, is set as 0. The addition and deletion of elements can be easily implemented by the exclusive-OR operation.

III. Encrypted Keyword Search Mechanism

A. Bitmap-based Dynamic SSE

For the description of B-SSE, we use cryptographic operations and the following system parameters. Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme and $\mathsf{P} : \{0,1\}^k \times \{0,1\}^\omega \to \{0,1\}^{\ell_w}$ be a pseudo-random function where $\omega = |w_i|$ and $\ell_w = |\mathsf{id}(w_i)|$ for $w_i \in \mathbf{W}$; the keyword identifier $\mathsf{id}(w_i)$ is computed by a pseudo-random function P. We assume that the subscript i for a keyword w_i is counted by one in the order of appearance in $\mathbf{F}$. In the construction, the file identifiers are chosen in the range of 1 to m, where m encrypted files can be stored in cloud storage; for $1 \le j \le m$, the subscript j of a file f_j is its identifier $\mathsf{id}(f_j) = j$ and $|\mathsf{id}(f_j)| = \ell_f$. By the bitmap index of Section II-B, we can express the set of such file identifiers $\mathsf{id}(\mathbf{F}(w_i))$ into the m-bit array $\mathsf{B_m}(\mathsf{id}(\mathbf{F}(w_i)))$, but the expression is complex. For simplicity, we use $\mathsf{B_m}(w_i)$ instead of $\mathsf{B_m}(\mathsf{id}(\mathbf{F}(w_i)))$. Let $\mathsf{H} : \{0,1\}^{k+\ell_r} \to \{0,1\}^m$ be a hash function where ℓ_r is the length of random numbers taken by H. The construction of B-SSE is described in Fig 1

In the construction, the index $\mathbf{I}$ for search and dynamic update is composed of two tables (T_s, T_f). The search table T_s stores the bitmap $\mathsf{B_m}(w_i)$ for keywords $w_i \in \mathbf{W}$ in an encrypted form; the bitmap $\mathsf{B_m}(w_i)$ is masked by a hash value. The bitmap shows which file identifiers are associated with a keyword w_i. The Search algorithm recovers this bitmap by performing one m-bit hash evaluation and one m-bit exclusive-OR computation, and the Add and the Delete algorithms update them by performing m-bit exclusive-OR computations on each keyword. In B-SSE, the file table T_f stores the file identifiers and their encrypted files $\mathbf{C}$ and, by the Search algorithm, returns the encrypted files $\mathbf{C}(w_i)$ indicated by the bitmaps. The file identifier is assigned in the range of 1 to m. In the assignment of file identifiers, we re-use the file identifiers, which were assigned to deleted files, for newly added files.

B. Security

The security of the proposed scheme is proven in the dynamic CKA2 model described in [7]. We define the leakage functions for B-SSE as follows:

- The L_1 leakage function, given a file set $\mathbf{F}$, outputs the leakage information,

$$L_1(\mathbf{F}) = \left(\{\mathsf{id}(w_i)\}_{i\in[\mathbf{w}]}, \{\mathsf{id}(f_j), |f_j|\}_{j\in\mathsf{id}(\mathbf{F})}\right),$$

  where $\{\mathsf{id}(w_i)\}$ is the set of identifiers of unique keywords and $\{\mathsf{id}(f_j), |f_j|\}$ is the set of file identifiers and their file sizes.

- The L_2 leakage function, given a file set $\mathbf{F}$ and a search keyword w_i, outputs the leakage information,

$$L_2(\mathbf{F}, w_i) = \left(\mathsf{id}(w_i), \{\mathsf{id}(f_j)\}_{f_j\in\mathbf{F}(w_i)}\right),$$

  where $\mathsf{id}(w_i)$ is the identifier of a search keyword and $\{\mathsf{id}(f_j)\}$ is the set of file identifiers including the search keyword w_i.

- The L_3 leakage function, given a file set $\mathbf{F}$ and an added file f_j, outputs the leakage information,

$$L_3(\mathbf{F}, f_j) = \left(\mathsf{id}(f_j), \{\mathsf{id}(w_i)\}_{w_i\in\mathbf{W}(f_j)}, |f_j|\right),$$

  where $\mathsf{id}(f_j)$ is the identifier of the added file, $\{\mathsf{id}(w_i)\}$ is the set of keyword identifiers included in f_j and $|f_j|$ is the added file's size.

- The L_4 leakage function, given a file set $\mathbf{F}$ and a deleted file f_j, outputs the leakage information,

$$L_4(\mathbf{F}, f_j) = \left(\mathsf{id}(f_j), \{\mathsf{id}(w_i)\}_{w_i\in\mathbf{W}(f_j)}\right),$$

  where $\mathsf{id}(f_j)$ is the identifier of the deleted file and $\{\mathsf{id}(w_i)\}$ is the set of keyword identifiers included in the deleted file.

Definition 1: (CKA2-security) Let B-SSE be a dynamic searchable symmetric encryption system and consider the following probabilistic experiments, where A is a stateful adversary, S is a stateful simulator and (L_1, L_2, L_3, L_4) are stateful leakage functions:

- $\mathbf{Real}_{\mathsf{A}}(k)$: The challenger runs $\mathsf{Gen}(1^k)$ to generate a key K. The adversary A outputs a set of files $\mathbf{F}$ and receives $\mathbf{I} = (T_s, T_f) \leftarrow \mathsf{BuildIndex}(K, \mathbf{F})$ from the challenger. A makes q adaptive queries $\lambda_i \in \{w, f_a, f_d\}$ for $1 \le i \le q$ where q is polynomial and, for each query λ_i, receives from the challenger either a search token $t_s \leftarrow \mathsf{SearchToken}(K, w)$ if $\lambda_i = w$, an addition token $t_a \leftarrow \mathsf{AddToken}(K, f_a)$ and the encryption of f_a if $\lambda_i = f_a$, or a deletion token $t_d \leftarrow \mathsf{DeleteToken}(K, f_d)$ if $\lambda_i = f_d$. Finally, A returns a bit b that is output by the experiment.

- $\mathbf{Ideal}_{\mathsf{A},\mathsf{S}}(k)$: The adversary A outputs the set of files $\mathbf{F}$. Given $L_1(\mathbf{F})$, the simulator S generates and sends a simulated index $\mathbf{I} = (T_s, T_f)$ to A. The adversary

– $\mathsf{Gen}(1^k)$: Sample random keys $K_1, K_2 \leftarrow \{0,1\}^k$, generate a encryption key $K_e \leftarrow \mathsf{SKE.Gen}(1^k)$ and output $K = (K_1, K_2, K_e)$.

– $\mathsf{BuildIndex}(K, \mathbf{F})$: Scan $\mathbf{F}$, extract distinct keywords $\mathbf{W}$, assign the identifiers in the range of 1 to m to the files in $\mathbf{F}$, and generate an index $\mathbf{I} = (T_s, T_f)$,

- Building the search table T_s:
  - For all $i \in [\mathbf{w}]$,
    1) compute $\mathsf{id}(w_i) = \mathsf{P}(K_1, w_i)$,
    2) generate $\mathsf{id}(\mathbf{F}(w_i))$ that is the set of the file identifiers including the keyword w_i,
    3) make the bitmap $\mathsf{B_m}(w_i)$ where $\mathsf{B_m}(w_i)[j] = 1$ for all $j \in \mathsf{id}(\mathbf{F}(w_i))$,
    4) choose random $r_i \leftarrow \{0,1\}^{\ell_r}$ and compute $x_i = h_i \oplus \mathsf{B_m}(w_i)$ where $h_i = \mathsf{H}(K_{w_i}||r_i)$ and $K_{w_i} = \mathsf{P}(K_2, w_i)$,
    5) and store $(key, value) = (\mathsf{id}(w_i), x_i||r_i)$.
- Building the file table T_f:
  - For all $j \in \mathsf{id}(\mathbf{F})$, where $\mathbf{f} \leq m$,
    1) compute $c_j \leftarrow \mathsf{SKE.Enc}_{K_e}(f_j)$,
    2) and store $(key, value) = (j, c_j)$.

output the index $\mathbf{I} = (T_s, T_f)$.

– $\mathsf{SearchToken}(K, w_i)$: For a search keyword w_i, compute $\mathsf{id}(w_i) = \mathsf{P}(K_1, w_i)$ and $K_{w_i} = \mathsf{P}(K_2, w_i)$ and output the search token $t_s = (\mathsf{id}(w_i), K_{w_i})$.

– $\mathsf{AddToken}(K, f)$: For an added file f and its keywords $w_i \in \mathbf{W}(f)$,

1) compute $\mathsf{id}(w_i) = \mathsf{P}(K_1, w_i)$,
2) choose random $r_i \leftarrow \{0,1\}^{\ell_r}$ and compute $h_i = \mathsf{H}(K_{w_i}||r_i)$ where $K_{w_i} = \mathsf{P}(K_2, w_i)$.

output the addition token $t_a = (\{\mathsf{id}(w_i)\}, \{h_i\}, \{r_i\})$ and the encryption $c \leftarrow \mathsf{SKE.Enc}_{K_e}(f)$.

– $\mathsf{DeleteToken}(K, f_j)$: For a deleted file f_j and keywords $w_i \in \mathbf{W}(f_j)$, compute $\mathsf{id}(w_i) = \mathsf{P}(K_1, w_i)$. Then, output the deletion token $t_d = (\mathsf{id}(f_j), \{\mathsf{id}(w_i)\})$.

– $\mathsf{Search}(\mathbf{I}, t_s)$: Given an index $\mathbf{I}$ and a search token $t_s = (\mathsf{id}(w_i), K_{w_i})$, find the $value$, $T_s[\mathsf{id}(w_i)] = x_i||r_i$, recover the bitmap $\mathsf{B_m}(w_i) = x_i \oplus \mathsf{H}(K_{w_i}||r_i)$ and, for all j such that $\mathsf{B_m}(w_i)[j] = 1$, return the identifiers j and the encrypted files c_j.

– $\mathsf{Add}(\mathbf{I}, t_a, c)$: Given an index $\mathbf{I}$, an addition token $t_a = (\{\mathsf{id}(w_i)\}, \{h_i\}, \{r_i\})$ and an encrypted file c, update the file table T_f and the search table T_s as follows,

- Adding the encrypted file in T_f:
  - Scan the file table T_f and if the j-th row is empty ($T_f[j] = null$), assign the subscript j to the added file and store $(key, value) = (j, c_j)$ in the j-th row; if there is no empty row in T_f, return $\perp$ indicating an error.
- Updating the bitmaps for $\mathbf{W}(f)$ in T_s:
  - For given $\mathsf{id}(w_i)$ appeared for the first time,
    1) store $(key, value) = (\mathsf{id}(w_i), x_i||r_i)$ where $x_i = h_i \oplus \mathsf{B_m}(\{j\})$.
  - For given $\mathsf{id}(w_i)$ existed before,
    1) find the value $T_s[\mathsf{id}(w_i)] = x_i||r_i$,
    2) and compute $x_i = x_i \oplus \mathsf{B_m}(\{j\})$ where $\mathsf{B_m}(\{j\})$ is the m-bit array such that j-th bit is 1 and the other bits are 0.

– $\mathsf{Delete}(\mathbf{I}, t_d)$: Given an index $\mathbf{I}$ and a deletion token $t_d = (\mathsf{id}(f_j), \{\mathsf{id}(w_i)\})$, update the file table T_f and the update table T_s as follows,

- Deleting the encrypted file in T_f:
  - Find the row with $key = j$ and delete it in T_f.
- Updating the bitmaps for $\mathbf{W}(f_j)$ in T_s:
  - For all given $\mathsf{id}(w_i)$,
    1) find the value $T_s[\mathsf{id}(w_i)] = x_i||r_i$,
    2) and compute $x_i = x_i \oplus \mathsf{B_m}(\{j\})$.

– $\mathsf{Dec}(K, c)$: Output $f = \mathsf{SKE.Enc}_{K_e}(c)$.

Fig. 1. The Bitmap-based Dynamic SSE scheme: B-SSE

A makes q adaptive queries $\lambda_i \in \{w, f_a, f_d\}$ for $1 \leq i \leq q$ where q is polynomial and, for each query λ_i, S gets the leakage information $L_2(\mathbf{F}, w)$, $L_3(\mathbf{F}, f_a)$, or $L_4(\mathbf{F}, f_d)$. S returns an appropriate token t_s, t_a, or t_d and the index $\mathbf{I}$ simulated on the tokens to A, where S additionally returns a simulated encryption of f_a when returning the addition token. Finally, A returns a bit b that is output by the experiment.

We say B-SSE is (L_1, L_2, L_3, L_4)-secure against adaptive dynamic chosen-keyword attacks (dynamic CKA2) if there exists a PPT simulator S such that

$$|\Pr[\mathbf{Real}_{\mathsf{A}}(k) = 1] - \Pr[\mathbf{Ideal}_{\mathsf{A},\mathsf{S}}(k) = 1]$$

is negligible for all PPT adversaries A.

Theorem 1: If SKE is CPA-secure and if P is pseudo-random, then B-SSE is (L_1, L_2, L_3, L_4)-secure against dynamic adaptive chosen-keyword attacks in the random oracle model.

Proof. To show that $\mathbf{Real}_{\mathsf{A}}(k)$ and $\mathbf{Ideal}_{\mathsf{A},\mathsf{S}}(k)$ are computationally indistinguishable, we describe a polynomial-time simulator S that adaptively simulates an index $\mathbf{I} = (T_s, T_f)$ and a sequence of token queries $\mathbf{t} = (t_1, \ldots, t_q)$ for search, addition and deletion, where $t_i \in \{t_s, t_a, t_d\}$ for $1 \leq i \leq q$ and q is polynomial. The simulator S reconstructs a new index from the leakage information of $L_1(\cdot)$, computes tokens and updates the index from the leakage information of $L_2(\cdot,\cdot), L_3(\cdot,\cdot)$ and $L_4(\cdot,\cdot)$, where the reconstruction and the computed tokens should satisfy dependencies between the leakage information from $L_1(\cdot), L_2(\cdot,\cdot), L_3(\cdot,\cdot)$ and $L_4(\cdot,\cdot)$. If they do not satisfy the dependencies, the adversary A may distinguish the simulated environments from the real environments. The adversary A can access to the simulated index and confirm that the simulated tokens are valid on the simulated index. We express the simulations of a table T into $\tilde{T}$ and of a value α into $\tilde{\alpha}$; for example, the simulation of $\mathbf{I} = (T_s, T_f)$ and $\mathbf{t} = (t_1, \ldots, t_q)$ are expressed into $\tilde{\mathbf{I}} = (\tilde{T}_s, \tilde{T}_f)$ and $\tilde{\mathbf{t}} = (\tilde{t}_1, \ldots, \tilde{t}_q)$, respectively.

For the simulation, the simulator S maintains internal data structures $\left(T_{L(s)}, T_{K_w}, T_{\mathsf{H}}\right)$. The leakage information table for search $T_{L(s)}$ shows the leakage information given by $L_1(\cdot)$, $L_2(\cdot,\cdot)$, $L_3(\cdot,\cdot)$ and $L_4(\cdot,\cdot)$. The searching key table T_{K_w} associates keyword identifiers $\mathsf{id}(w_i)$ with the searching keys K_{w_i} (for $i \in [\mathbf{w}]$), and the random oracle tables T_{H} stores the oracle queries and their corresponding responses. Given the leakage information from $L_1(\cdot), L_2(\cdot,\cdot), L_3(\cdot,\cdot)$ and $L_4(\cdot,\cdot)$, the simulations are as follows:

Given the leakage information,

$$L_1(\mathbf{F}) = \left(\{\mathsf{id}(w_i)\}_{i\in[\mathbf{w}]}, \{\mathsf{id}(f_j), |f_j|\}_{j\in\mathsf{id}(\mathbf{F})}\right),$$

S generates a simulated encrypted index $\tilde{\mathbf{I}} = (\tilde{T}_s, \tilde{T}_f)$ as follows:

- $\tilde{T}_s$: The simulated search table stores $(key, value) = (\mathsf{id}(w_i), \tilde{x}_i || r_i)$ for $i \in [\mathbf{w}]$ where $\mathsf{id}(w_i)$ is given by $L_1(\mathbf{F})$, and $\tilde{x}_i$ and r_i are random chosen by S.
- $\tilde{T}_f$: The simulated file table stores $(key, value) = (j, \tilde{c}_j)$ for $j \in \mathsf{id}(\mathbf{F})$ where the file identifier $\mathsf{id}(f_j) = j$ is given by $L_1(\mathbf{F})$ and $\tilde{c}_j$ is the encryption simulated by S; S generates $K_e \leftarrow \mathsf{SKE.Gen}(k)$ and computes $\tilde{c}_j \leftarrow \mathsf{SKE.Enc}_{K_e}\left(0^{|f_j|}\right)$.

From the leakage information of $L_1(\mathbf{F})$, the simulator S outputs the simulated index $\tilde{\mathbf{I}} = (\tilde{T}_s, \tilde{T}_f)$. In the tables $\tilde{T}_s$ and $\tilde{T}_f$, the difference with the real environment is that S chooses random x_i instead of the output of the random oracle H and replaces the ciphertext c_j with the encryption of all zero string. The simulations on the other values are equivalent with those of the real environment. The assumptions of random oracle and CPA-security of SKE guarantee that the PPT adversary A cannot distinguish the difference between the simulation and the real game with non-negligible probability.

For the next simulation, the simulator S records the information given by the leakage function $L_1(\mathbf{F})$ in the internal data structures $\left(T_{L(s)}, T_{K_w}, T_{\mathsf{H}}\right)$. S sets the internal data structures as follows:

- $T_{L(s)}$: S stores $(key, value) = (\mathsf{id}(w_i), L_{x_i})$ where the leakage information for the keyword w_i, L_{x_i}, is the bitmap $\mathsf{B}_{\mathsf{m}}(w_i)$ that shows the file identifiers associated with the keyword identifier $\mathsf{id}(w_i)$; in the bitmap $\mathsf{B}_{\mathsf{m}}(\cdot)$, the order of bit positions is in the identity relation with the file identifiers. When searching a keyword w_i, the Search algorithm reveals which files are associated with the search keyword w_i. S records such access patterns in $T_{L(s)}$. At the beginning of the simulation, S just stores the keyword identifiers $\{\mathsf{id}(w_i)\}_{i\in[\mathbf{w}]}$ given by $L_1(\mathbf{F})$ as key and sets the corresponding $value = L_{x_i}$ as zero m-bit. After then, L_{x_i} will be filled by the leakage information from $L_2(\cdot,\cdot)$ and updated by those from $L_3(\cdot,\cdot)$ and $L_4(\cdot,\cdot)$.
- T_{K_w}: S stores $(key, value) = (\mathsf{id}(w_i), \tilde{K}_{w_i})$ where $\tilde{K}_{w_i}$ is k-bit random that substitutes the output of the pseudo-random function P in the real game. Under the assumption of pseudo-randomness of P, the simulation of the searching key is indistinguishable from that of the real game in the view of the adversary A. At the beginning of the simulation, S chooses k-bit random $\tilde{K}_{w_i}$ for all $\mathsf{id}(w_i)$ given by $L_1(\mathbf{F})$ and stores them.
- T_{H}: S simulates the random oracle $\mathsf{H} : \{0,1\}^{k+\ell_r} \rightarrow \{0,1\}^m$ by using T_{H}. The random oracle table T_{H} stores two types of values, $(query, response)$ where $query$ is a $(k + \ell_r)$-bit string and $response$ is a m-bit string. S returns the following responses $\tilde{h}$ for the queries $(s_1||s_2)$ where s_1 and s_2 are k-bit and ℓ_r-bit strings, respectively.
  1) If $(s_1||s_2)$ exists in the column of $query$, returns its corresponding response $\tilde{h}$.
  2) If $(s_1||s_2)$ does not exist in the column of $query$, for $i \in [\mathbf{w}]$,
     a) if $(s_1||s_2) \neq (\tilde{K}_{w_i}||r_i)$, pick m-bit random $\tilde{h}$,
     b) if $(s_1||s_2) = (\tilde{K}_{w_i}||r_i)$, compute $\tilde{h} = \tilde{x}_i \oplus L_{x_i}$ where $\tilde{x}_i$ and L_{x_i} are in $\tilde{T}_s$ and $T_{L(s)}$, respectively.

  S stores $(query, response) = (s_1||s_2, \tilde{h})$.

To successfully simulate the random oracle H, S should know the leakage information L_{x_i}. If not, S cannot simulate the random oracle H. This event occurs when A makes the queries without the knowledge of $\tilde{K}_{w_i}$. We ignore the effect by this event because the probability that the event occurs is 2^{-k}; it is negligible.

Given the leakage information,

$$L_2(\mathbf{F}, w_i) = \left(\mathsf{id}(w_i), \{\mathsf{id}(f_j)\}_{f_j \in \mathbf{F}(w_i)}\right),$$

S updates the internal data structures, computes a simulated token for search, and reflects the search results in the simulated index. The internal data structure $T_{L(s)}$ are updated as follows:

- $T_{L(s)}$: The leakage information of $L_2(\mathbf{F}, w_i)$ reveals that the files identifiers $\{\mathsf{id}(f_j)\}_{f_j \in \mathbf{F}(w_i)}$ are associated with the keyword identifier $\mathsf{id}(w_i)$. From this information, S sets a bitmap for $j \in \mathsf{id}(\mathbf{F}(w_i))$,

$$L_{x_i} = L_{x_i} \oplus \mathsf{B_m}(\{j\}).$$

After updating the internal data structure, S computes the simulated search token t_s for a keyword w_i,

$$t_s = (\mathsf{id}(w_i), K_{w_i}),$$

where S gets the information about $\mathsf{id}(w_i)$ and K_{w_i} from the leakage information $L_2(\mathbf{F}, w_i)$ and the searching key table T_{K_w}, respectively. When searching a keyword w_i in the simulated index, the search process on the token t_s returns the same leakage information as that of $L_2(\mathbf{F}, w_i)$. Finally, S reflects the search results in the simulated index $\tilde{\mathbf{I}} = (\tilde{T}_s, \tilde{T}_f)$ as follows:

- $\tilde{T}_s$: S sets $\tilde{x}_i = L_{x_i} \oplus \tilde{h}'$ and stores $(key, value) = (\mathsf{id}(w_i), \tilde{x}_i || r'_i)$.
- $\tilde{T}_f$: There is no change at this simulation.

Given the leakage information,

$$L_3(\mathbf{F}, f_j) = \left(\mathsf{id}(f_j), \{\mathsf{id}(w_i)\}_{w_i \in \mathbf{W}(f_j)}, |f_j|\right),$$

S updates the internal data structures, computes a simulated token for addition, and reflects the simulation results for addition in the simulated index. The internal data structures $\left(T_{L(s)}, T_{K_w}, T_{\mathsf{H}}\right)$ are updated as follows:

- $T_{L(s)}$: The leakage information of $L_3(\mathbf{F}, f_j)$ reveals that the file identifier $\mathsf{id}(f_j)$ is associated with the keywords identifiers $\{\mathsf{id}(w_i)\}_{w_i \in \mathbf{W}(f_j)}$. From the information, S updates L_{x_i} for $w_i \in \mathbf{W}(f_j)$. Among the set of keyword identifiers $\{\mathsf{id}(w_i)\}_{w_i \in \mathbf{W}(f_j)}$, if there are new keyword identifiers that are not in the leakage information table $T_{L(s)}$, S adds the rows such that $(key, value) = (\mathsf{id}(w_i), L_{x_i} = \mathsf{B_m}(\{j\}))$ where $\mathsf{B_m}(\{j\})$ is the m-bit array such that j-th bit is 1 and the other bits are 0. For the existing keyword identifiers, S updates the values as follows,

$$L_{x_i} = L_{x_i} \oplus \mathsf{B_m}(\{j\}).$$

- T_{K_w}: If a new keyword identifier $\mathsf{id}(w)$ appears in the set of $\{\mathsf{id}(w_i)\}_{w_i \in \mathbf{W}(f_j)}$, then S chooses a k-bit random number K_w and stores $(key, value) = (\mathsf{id}(w), K_w)$.
- T_{H}: For i such that $w_i \in \mathbf{W}(f_j)$, S chooses ℓ_r-bit random numbers r_i and m-bit random numbers $\tilde{h}_i$, and stores $(query, response) = (K_{w_i} || r_i, \tilde{h}_i)$; this simulation is for the generation of the addition token.

After updating the internal data structures, S computes the simulated token t_a, for i such that $w_i \in \mathbf{W}(f_j)$,

$$t_a = (\{\mathsf{id}(w_i)\}, \{\tilde{h}_i\}, \{r_i\}),$$

and $\tilde{c}_j \leftarrow \mathsf{SKE.Enc}_{K_e}(0^{|f_j|})$. The addition process on the token t_a returns the same leakage information as that of $L_3(\mathbf{F}, f_j)$. Finally, S reflects the simulation results in the simulated index $\tilde{\mathbf{I}} = (\tilde{T}_s, \tilde{T}_f)$ as follows:

- $\tilde{T}_s$: If there is no row with $key = \mathsf{id}(w_i)$ for $w_i \in \mathbf{W}(f_j)$, S stores $(key, value) = (\mathsf{id}(w_i), \tilde{x}_i || r_i)$ where $\tilde{x}_i = \tilde{h}_i \oplus \mathsf{B_m}(\{j\})$. If there is $key = \mathsf{id}(w_i)$ in $\tilde{T}_s$, S updates the corresponding value, $x_i = x_i \oplus \mathsf{B_m}(\{j\})$.
- $\tilde{T}_f$: S stores the file identifier $\mathsf{id}(f_j)$ and its associated encrypted file $\tilde{c}_j$.

Given the leakage information,

$$L_4(\mathbf{F}, f_j) = \left(\mathsf{id}(f_j), \{\mathsf{id}(w_i)\}_{w_i \in \mathbf{W}(f_j)}\right),$$

S updates the internal data structures, computes a simulated token for deletion, and reflects the simulation results for deletion in the simulated encrypted index. The internal data structure $T_{L(s)}$ is updated as follows:

- $T_{L(s)}$: The leakage information of $L_4(\mathbf{F}, f_j)$ reveals that the file identifier $\mathsf{id}(f_j)$ is associated with the keywords identifiers $\{\mathsf{id}(w_i)\}_{w_i \in \mathbf{W}(f_j)}$. From the information, S updates L_{x_i} for $\{\mathsf{id}(w_i)\}_{w_i \in \mathbf{W}(f_j)}$. S updates the values for $\{\mathsf{id}(w_i)\}_{w_i \in \mathbf{W}(f_j)}$ as follows,

$$L_{x_i} = L_{x_i} \oplus \mathsf{B_m}(\{j\}).$$

After updating the internal data structures, S computes the simulated token t_d, for i such that $w_i \in \mathbf{W}(f_j)$,

$$t_d = (\ id(f_j), \{id(w_i)\}).$$

The deletion process on the token t_d returns the same leakage information as that of $L_4(\mathbf{F}, f_j)$. Finally, S reflects the search results in the simulated index $\tilde{\mathbf{I}} = (\tilde{T}_s, \tilde{T}_f)$ as follows:

- $\tilde{T}_s$: For i such that $w_i \in \mathbf{W}(f_j)$, S updates the corresponding value, $x_i = x_i \oplus \mathsf{B_m}(\{j\})$.
- $\tilde{T}_f$: S deletes the file identifier $\mathsf{id}(f_j)$ and its corresponding encrypted file $\tilde{c}_j$.

In this proof, we describe the simulator S to simulate the real environments by using the given leakage information from (L_1, L_2, L_2, L_4). Assuming that SKE is CPA-secure, P is pseudo-random and H is a random oracle, the simulation, $\mathbf{Ideal}_{\mathsf{A},\mathsf{S}}(k)$, is indistinguishable from the real environments $\mathbf{Real}_{\mathsf{A}}(k)$ in the view of the adversary A. □

TABLE I. THE COMPUTATION COSTS WITH THE PREVIOUS SCHEME WHERE P: PSEUDO RANDOM EVALUATION, H: HASH EVALUATION, X: EXCLUSIVE-OR EVALUATION, H_m: m-BIT HASH EVALUATION AND X_m: m-BIT EXCLUSIVE-OR EVALUATION.

Scheme	Search	Add	Delete
KPR [7]	p+$\mathbf{f}_w$·h +($\mathbf{f}_w$+1)·x	5$\mathbf{w}_f$·x	$\mathbf{w}_f$·h +(11$\mathbf{w}_f$+1)·x
B-SSE	h_m+x_m	($\mathbf{w}_f$+1)·x_m	($\mathbf{w}_f$+1)·x_m

IV. ANALYSIS

A. Comparison

In the literature of dynamic SSE, there have been several schemes [5], [6], [7], [9], [10], [11]. The scheme of [5] has not been proven in a rigorous security model and the schemes of [6], [9], [10] are interactive in the updates (i.e., addition or deletion). The dynamic SSE schemes of [7] and [11] support non-interactive update and their security is proven in regorous security models; the scheme of [11] achieves higher level security than that of [7]. The two schemes are based on the inverted index form. The difference is to use different data structures, the linked list structure and the dictionary using counter, respectively. The search time and index size complexities in [11] are not significantly different from those of [7]. In [11] the scheme traverses $\mathbf{f}_w$ cells for recovering the file identifiers and the cost to traverse one cell is at least one pseudo random evaluation and one decryption. Its index size complexity is also the number of all document-key pairs (i.e., $\mathbf{f}_w \times \mathbf{w}$). For this reason, we only compare B-SSE with KPR [7].

Table I shows the computation costs in Search, Add and Delete. The measurement is considered by three cryptographic computations. In the case of B-SSE, we use the notations, h_m and x_m, for the m-bit hash evaluation and the m-bit exclusive-OR computation; for B-SSE, the hash and exclusive-OR computations are dependent on the size of bitmap. In Table I the other systematic operations (e.g., filesystem access) are not considered.

Search in Table I shows the cost to recover the file identifiers from each index of B-SSE and KPR. B-SSE uses one m-bit hash function and one m-bit exclusive-OR computation while KPR uses a pseudo random function, a hash function and the exclusive-OR computation for some fixed lengths. Since they use different measurements, we need to display the numerical values for more detailed comparison.

In Figure 2, we present numerical values (computed by Table I) on each search time according to the number of files ($\mathbf{f}$) and the keyword appearance rate ($\mathbf{f}_w/\mathbf{f}$). KPR gives similar performance in the case of the keyword appearance rate $\mathbf{f}_w/\mathbf{f} = 0.4\%$ with B-SSE though we do not draw the case in Figure 2, and the searching time of KPR increase by the keyword appearance rate. The searching time of B-SSE is more fast than KPR when the keyword appearance rate is higher than 0.4%.

As shown in Table I, KPR has different searching time according to the number of files with the search keyword $\mathbf{f}_w$ and B-SSE is affected by the number of files (i.e., $\mathbf{f} = m$). For this reason, to show the searching time by $\mathbf{f}_w$ and $\mathbf{f}$, we illustrate the searching time of KPR by the keyword appearance rate $\mathbf{f}_w/\mathbf{f}$ (=1%, 2%, 4%, 8%, 16%) as well as $\mathbf{f}$ while the searching time of B-SSE is only illustrated by $\mathbf{f}$. In real applications such as the gallery and the e-mail system, we expect that the keyword appearance rate is higher than 1% on average. In the gallery, the images do not include a lot of keywords. They usually have fields less than 50 for storing metadata and when assuming each field has at maximum 100 keywords, the number of unique keywords becomes at maximum 5,000. If the unique keywords exist uniformly in the files, the keyword appearance rate is 1% (=$50 \cdot \mathbf{f}/5,000/\mathbf{f} \times 100$). When considering that the metadata of the gallery for personal usage are restricted by some specific data, the keyword appearance rate will be higher than 1% on average. In the e-mail system, user also uses some specific keywords (e.g., sender and his interesting keywords) for searching. In our lab., the keyword appearance rate for the frequently-used keywords such as *sender*, *receiver* and *title* is investigated more than 5% on average. For the keyword appearance rate, we expect that this aspect is not significantly different in other many applications.

For getting numerical values, we measure the running times of three measurements as follows[3]. We assume that the pseudo random function is HMAC-SHA-256 and the hash function is SHA-256 where the running times on each algorithm are about 60 μs and about 25 μs, respectively. The m-bit hash evaluation is measured by iterating $\lceil m/256 \rceil$ hash function. The exclusive-OR computation takes 0.6 μs on 256-bit and linearly increases by the size of bits; for 5,000-bit, the exclusive-OR computation takes about 2 μs and for 10,000-bit, it takes about 4 μs.

In Figure 2, we show the searching time in the range of $\mathbf{f} = 5,000$ to $\mathbf{f} = 100,000$. In the market, the cloud storage services such as iCloud, GoogleDrive, Dropbox and SkyDrive provide at the maximum 50 GB storage. When storing 1 MB images in the cloud storages, we can store at the maximum 50,000 files and when storing 500 KB documents, we can store at the maximum 100,000 files. We think that $\mathbf{f} = 100,000$ is enough to cover the number of files stored in most cloud storage services.

Add and Delete in Table I show the cost to update the indices of B-SSE and KPR. For Add, B-SSE is less efficient than KPR when $\mathrm{x}_m > 5$·x; these cases occur when $\mathbf{f} > 7,000$. For Delete, B-SSE is more efficient than KPR when $\mathbf{f} < 80,000$. Assuming that $\mathbf{w}_f >> 1$ and h≈ 40·x, the costs of KPR and B-SSE become $51\mathbf{w}_f$·x and $\mathbf{w}_f$·x_m, respectively. From these approximation, we can know that B-SSE is more efficient than KPR if $\mathrm{x}_m < 51$x; these cases occur when $\mathbf{f} < 80,000$. The update algorithms of B-SSE provides better performance in some cases.

We do not analyze the performance of the other algorithms: BuildIndex, SearchToken, AddToken and DeleteToken. BuildIndex is performed at initial when user starts personal storage services and the token generation algorithms (i.e., SearchToken, AddToken and DeleteToken) do not require high computational costs on both B-SSE and KPR. We think that they are not essential elements to evaluate the performance of two schemes.

[3]See Section IV-B for the simulated environment.

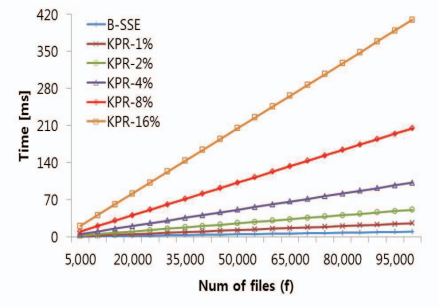

Fig. 2. The file identifier searching time when the keyword appearance rates ($\mathbf{f}_w/\mathbf{f}$) are 1%, 2%, 4%, 8% and 16%.

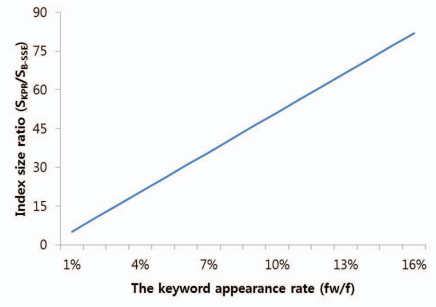

Fig. 3. The index size ratio between B-SSE and KPR according to the keyword appearance rate.

In B-SSE, the required space for the index T_s linearly increases according to the number of files ($\mathbf{f}$) and unique keywords ($\mathbf{w}$). For B-SSE, the index size in various cases can be computed according to the sizes of system parameters such as bitmap, hash value and keyword identifiers. The B-SSE requires $S_{\text{B-SSE}} = \mathbf{w}\times(\ell_\omega+m+\ell_r)$ ($\approx \mathbf{w}\cdot\mathbf{f}$ where $\mathbf{f}=m$) bits to maintain the search table T_s. For KPR, the index size is simply computed as $S_{\text{KPR}} = (\mathbf{f}_w\times\mathbf{w}\times k\times\#A)+(\mathbf{w}\times k\times\#T)$ ($\approx 2\cdot k\cdot\mathbf{w}\cdot\mathbf{f}_w$) where $\#A=2$ and $\#T=2$ is the number of arrays and tables, respectively, and k is the output size of the pseudo random function. Under the assumption of $k=256$, the required index size of B-SSE is smaller than that of KPR if the keyword appearance rate is higher than about 0.2% (i.e., $S_{\text{B-SSE}} < S_{\text{KPR}}$), where the keyword appearance rate 0.2% is much smaller value that we expected in the real applications such as the gallery and the e-mail system. The index size ratio ($S_{\text{KPR}}/S_{\text{B-SSE}}$) between B-SSE and KPR is shown in Figure 3 according to the keyword appearance rate in the range of 1% to 16%. When $\mathbf{f}=50,000$ and $\mathbf{w}=10,000$ under the assumption of $\ell_\omega=256$ and $\ell_r=256$, , the index size of B-SSE is $S_{\text{B-SSE}}=599$ MB and the index size of KPR is about 5 times of $S_{\text{B-SSE}}$ ($S_{\text{KPR}}=3,055$ MB) in the case of $\mathbf{f}_w/\mathbf{f}=1\%$. As shown in Figure 3, when the keyword appearance rate is higher than 1%, KPR requires much more index size in comparison with B-SSE.

B. Test

The test is performed on an Intel Core2 Quad 2.67 GHz with 3 GB memory where the operation system is Windows 7 (32 bits) and the program language is Java. To make sample files for the test, we collect $\mathbf{w}=10,000$ keywords and scatter them to $\mathbf{f}=10,000$ files, uniformly, where each file is composed of 100 keywords and its size is about 3.125 KB; it enables each bitmap (of the same size) to have the almost same number of file identifiers in the test. We first examine the performance on each algorithm of B-SSE.[4]

For the implementation of B-SSE, we set the system parameters of B-SSE and select the cryptographic algorithms at first. For the system parameters, we set the size of the secret key $k=256$ bits, the keyword size $\omega=256$ bits, the keyword identifier size $\ell_w=256$ bits, the file identifier size $\ell_f=64$ bits and the size of random numbers $\ell_r=256$ bits. The bitmap size m (i.e., the maximum number of storable files) is configured variously from $5,000$-bit to $40,000$-bit for the test. We assume that a user is storing at the maximum m files in the file table T_f, which means $\mathbf{f}=m$. For the cryptographic algorithms, we use SHA-256 for the hash function and the keyed-hash message authentication code, HMAC-SHA-256, for the pseudo-random function P. In B-SSE, the hash function H takes as input a $(k+\ell_r)$-bit string and outputs a m-bit string. We get the m-bit string by concatenating and truncating several hash values driven by the same input string. The pseudo-random function P is completely replaced with HMAC-SHA-256 without additional handling.

For practical analysis we test B-SSE with Couchbase (ver. 2.2.0). In Couchbase, indices are created via Views[5] and queries can be executed by using a suitable library to retrieve documents from DB. For the test of B-SSE, we construct an application server that is composed of an index server and a file server. The index server maintains the search table T_s. When users request search queries, it returns the associated file identifiers to the file server, and, when there is a request for addition and deletion from users, it updates the bitmap associated with the added/deleted files in the search table T_s. The file server maintains the file table T_f. It returns the encrypted files associated with the file identifiers given by the index server, and adds or deletes files in the file table T_f by the request of users. In our test, the index server and the file server interact with each other via their SDK (software development kit) that establishes a connection and supports read/write and other functions.

We compare the search time of B-SSE with that of the ordinary search mechanism (we call it Ord-Search) to examine the overheads by the security mechanisms of B-SSE. Ord-Search can be implemented by using Views. All the files are arranged by unique keywords and when searching files with a keyword, the server gets the file identifiers associated with the keyword via a View.

To show B-SSE is practically applicable to real cloud storage services, we measure the time from sending the search token to receiving encrypted files. The test result examines the time during users are waiting for the response from the server. Figure 4 shows the waiting time of B-SSE and Ord-Search from $\mathbf{f}=1,000$ to $\mathbf{f}=10,000$ when $\mathbf{w}=10,000$.

[4]The performance of KPR [7] depends on its implementation. For this reason, we do not compare B-SSE with KPR

[5]A View is the result set of a stored query (or map-and-reduce functions) on the data

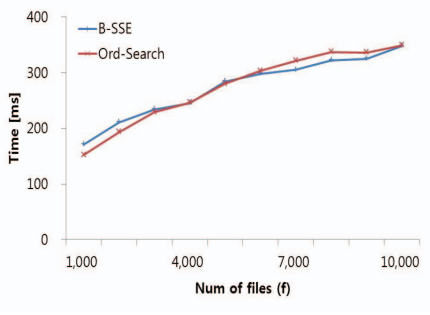

Fig. 4. The search time when integrated with Couchbase where $\mathbf{w} = 10,000$ and Ord-Search is an ordinary search mechanism in the inverted index form.

In Figure 4, B-SSE & Ord-Search cross each other at some points, which shows the search time of two methods is not significantly different. It also means that the overheads by security mechanisms do not affect to the performance in the view of users. The test result shows B-SSE can be sufficiently applied to the real cloud storage services.

V. Bitmap Expansion

In B-SSE of Section III-A, users could not store more than m files in cloud storages because the bitmap size m is fixed at the beginning. For this reason, to cover a large amount of files, B-SSE should set the bitmap size sufficiently large. However, it wastes the memory of cloud storage unnecessarily due to the increase of bitmap sizes. This limitation might be an obstacle to the usage of B-SSE in real services. If B-SSE can manage the size of the bitmap (i.e., the number of storable files) flexibly, it can reduce unnecessary memory used in the cloud storage. We introduce a simple method to expand the bitmap size when users store more than m files.

To store more than m files, the bitmap should be expanded into more than m-bit. However, simply to add additional bits to the m-bit array causes the change of the length of the system parameters such as hash values and random numbers. This is undesirable modification in our construction. We use the linked list data structure to expand the bitmap size.[6] When storing $(m+1)$ files, the server generates additional m-bit arrays, for the keywords included in the $(m+1)$-th file, whose bit positions are associated with $\{m+1, \ldots, 2m\}$, and binds two bitmaps in the linked list form. The linked list form includes the addresses to indicate the next link (i.e., bitmap). For hiding the relation between the $(m+1)$-th file and its keyword identifiers, we mask the address in the same manner of masking bitmaps.

In the above methodology, the search table T_s stores

$$(key, value) = (id(w_i), x_{i,j}||r_{x_{i,j}}||y_{i,j}||r_{y_{i,j}}),$$

where, for $1 \leq i \leq \mathbf{w}$, $j \geq 1$ and $K_3 \leftarrow \{0,1\}^k$,

$$x_{i,j} = \mathsf{H}(K_{w_i}||r_{x_{i,j}}) \oplus \mathsf{B}_{\mathsf{m}(j)}(w_i), \quad K_{w_i} = \mathsf{P}(K_2, w_i),$$
$$y_{i,j} = \mathsf{H}(K_{a_i}||r_{y_{i,j}}) \oplus addr_{i,j}, \quad K_{a_i} = \mathsf{P}(K_3, w_i).$$

The value $addr_{i,j}$ indicates the location of the $(j+1)$-th bitmap for the keyword w_i and each bitmap $\mathsf{B}_{\mathsf{m}(j)}$ is the set representation of file identifiers $\{(j-1)\cdot m+1, \ldots, j\cdot m\}$. When adding a new bitmap $\mathsf{B}_{\mathsf{m}(j+1)}$ for the keyword w_i, a random value is assigned to $addr_{i,j}$ by the exclusive-OR operation and becomes key to find the bitmap $\mathsf{B}_{\mathsf{m}(j+1)}$ where the $addr_{i,j+1}$ is set as zero bits. To search the files associated with the keyword w_i, we can consider the following $\mathsf{SearchToken}$,

$$t_s = \{\mathsf{id}(w_i), K_{w_i}, K_{a_i}\}.$$

The server first finds the bitmap $\mathsf{B}_{\mathsf{m}(1)}(w_i)$ from the keyword identifiers $\mathsf{id}(w_i)$ and recovers $addr_{i,j}$ by using the key K_{a_i} recursively. If $addr_{i,j}$ is zero bits, the server stops searching the bitmaps for the keyword w_i. Since the expanded B-SSE using the expanded bitmap shares the same search and update processes as that of Section III-A, we omit the detail of each algorithm.

[6] For the same keyword, B-SSE connects the set of files (i.e., bitmap) while KPR [7] connects individual files.

VI. Conclusion

We introduced a new keyword search mechanism, B-SSE, for personal storage services. B-SSE achieves fast search time and small index size in commonly-used cases where the keyword appearance rate is more than 0.4% on average. We tested B-SSE with the known NoSQL database, Couchbase. The result shows that B-SSE can be applied to real applications in practice. B-SSE can also flexibly deal with the bitmap size by expanding bitmap, which reduces the use of unnecessary memory for bitmap index.

References

[1] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000, pp. 44–55.

[2] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO (1)*, ser. Lecture Notes in Computer Science, vol. 8042. Springer, 2013, pp. 353–373.

[3] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *ASIACRYPT*, ser. Lecture Notes in Computer Science, vol. 6477. Springer, 2010, pp. 577–594.

[4] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *ACM Conference on Computer and Communications Security*. ACM, 2006, pp. 79–88.

[5] E.-J. Goh, "Secure indexes," *IACR Cryptology ePrint Archive*, vol. 2003, p. 216, 2003.

[6] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *FC*, ser. Lecture Notes in Computer Science, vol. 7859. Springer, 2013, pp. 258–274.

[7] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 965–976.

[8] K. Kurosawa and Y. Ohtaki, "Uc-secure searchable symmetric encryption," in *FC*, ser. Lecture Notes in Computer Science, vol. 7397. Springer, 2012, pp. 285–298.

[9] P. van Liesdonk, S. Sedghi, J. Doumen, P. H. Hartel, and W. Jonker, "Computationally efficient searchable symmetric encryption," in *SDM*, ser. Lecture Notes in Computer Science, vol. 6358. Springer, 2010, pp. 87–100.

[10] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *NDSS*. Internet Society, 2014 (to appear).

[11] D. Cash, J. Jaeger, S. Jarecki, and C. Jutla, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *NDSS*. Internet Society, 2014 (to appear).