Secure Keyword Search Using Bloom Filter with Specified Character Positions

Takanori Suga[1], Takashi Nishide[2], and Kouichi Sakurai[2]

[1] Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University
[2] Department of Informatics, Faculty of Information Science and Electrical Engineering, Kyushu University

Abstract. There are encryption schemes called searchable encryption which enable keyword searches. Traditional symmetric ones support only full keyword matches. Therefore, both a data owner and data searcher have to enumerate all possible keywords to realize a variety of searches. It causes increases of data size and run time.We propose searchable symmetric encryption which can check characters in the specified position as we perform search on plaintexts. Our scheme realizes a variety of searches such as fuzzy keyword search, wildcard search, and so on.

Keywords: Symmetric encryption, searchable encryption, Bloom filter.

1 Introduction

1.1 Background

In recent years, cloud computing is spreading rapidly and widely due to advance in computer and telecommunication technology. In cloud computing, we outsource not only data but also processing to cloud servers.

The users cannot carry out investigations into the cause of security incidents and the measures for preventing the recurrence of such incidents because the users cannot know how cloud providers manage their servers. Therefore, we need the measures for preventing the leakages before sending data to the cloud servers. However, traditional encryption schemes prevent not only the leakages but also the conveniences like searches. There are encryption schemes which enable us to search without decryption. These schemes are called searchable encryption and attract a great deal of attention.

There exist symmetric searchable encryption schemes and asymmetric ones. The asymmetric ones can be used when it is difficult for us to share a secret key securely as we send an e-mail. On the other hand, we can use symmetric ones if and only if we can share a secret key securely, for example, when we share files in the same organization. However, the symmetric ones can be executed faster than asymmetric encryption in general. Therefore, we focus on symmetric searchable encryption in this work.

An information flow of searchable encryption schemes is shown in Figure 1.

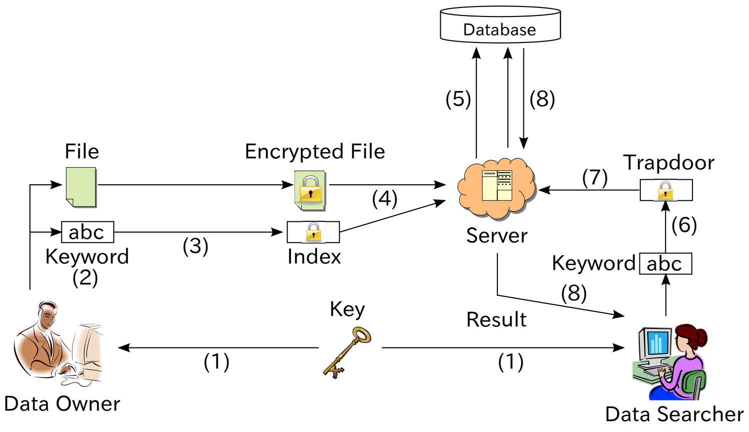

Fig. 1. Flow of Searchable Encryption

1. A data owner shares a key with data searchers before encryption.
2. The data owner specifies the keywords $\mathrm{Kw_A}$ which represent the contents of the file.
3. The data owner computes indexes $\mathrm{Idx_A}$ of $\mathrm{Kw_A}$. "Index" means a data structure which enables the server to perform an encrypted keyword search if and only if the server obtains a corresponding trapdoor that is explained later.
4. The data owner sends encrypted data C_{A} and $\mathrm{Idx_A}$ to a server.
5. The server registers C_{A} and $\mathrm{Idx_A}$ to the database.
6. The data searcher computes trapdoors $\mathrm{Td_B}$ of keywords $\mathrm{Kw_B}$. "Trapdoor" means a data structure which enables the data searcher to query the keyword the data searcher wants to search in the server without revealing it.
7. The data searcher sends $\mathrm{Td_B}$ to the server.
8. The server searches indexes in the database by using $\mathrm{Td_B}$ and returns the result to the data searcher.

$\mathrm{Idx_A}, \mathrm{Td_B}$ do not reveal $\mathrm{Kw_A}, \mathrm{Kw_B}$ to the server. Therefore, we can retrieve data without letting the server know the keywords.

The types of searches are as follows.

Equality search. The keywords which match the query completely hit.
Prefix search. The keywords which contain the query in the head hit.
Suffix search. The keywords which contain the query in the tail hit.
Partial matching search. The keywords which contain the query anywhere hit.
Wildcard search. The keywords which match any character other than wildcard characters hit. Wildcard characters represent any characters.
Fuzzy keyword search. The keywords within a certain edit distance from the query hit.

Full-text search. We search the full content of a document for specified several keywords.

Relational database search. We search a relational database for some values (keywords, numbers, and so on). We usually use a query language like SQL in this search.

1.2 Motivation

The searches for plaintexts such as search engines, searches in the computer, searches for e-mails in a mailbox are realized in various ways and very convenient. However, the available types of searches in searchable encryption schemes have been restricted because the server searches hidden plaintexts for hidden keywords. Therefore, our motivation is to enable to compare keywords in the trapdoor per character with the hidden keyword in the index to realize more advanced searches.

1.3 Related Works

Song et al. proposed the first practical searchable symmetric encryption scheme in 2000 [15]D This scheme supports only equality search and thus is not so convenient. After that, many searchable symmetric encryption schemes were proposed [2,6,7,17]. These schemes aim to enhance the situations where they are usable or to improve security and do not support any method other than equality search.

In recent years, Li et al. proposed the first searchable encryption scheme which supports a fuzzy keyword search [11]. This scheme is based on wildcard realized by enumerating keywords like "?apple", "?pple", ..., and "apple?" for "apple". However, we cannot use any other wildcard patterns than prepared patterns for fuzzy keyword search because this wildcard is realized by whole keyword match as equality search. Furthermore, we cannot search any keyword other than those which are enumerated by the data owner.

Boneh et al. proposed the first searchable asymmetric encryption scheme [4]. Abdalla et al. proposed anonymous IBE [1] and mentioned the relationship to searchable asymmetric encryption scheme. After that, many searchable asymmetric encryption schemes were proposed [5]. Sedghi et al. proposed a searchable asymmetric encryption scheme which supports wildcard search [14]. This scheme support a fixed wildcard search as our scheme and is realized with bilinear pairing. In symmetric key settings, we can execute symmetric key encryption schemes or hash functions faster than asymmetric key encryption schemes in general. However, there is no searchable symmetric encryption scheme which supports general wildcard searches.

Goh proposed a searchable symmetric encryption scheme for full-text search with Bloom filter [8] and Watanabe et al. proposed a searchable symmetric encryption scheme for relational database with Bloom filter [16]. These schemes use Bloom filter for efficiency.

1.4 Challenging Issues

After the first searchable encryption scheme was proposed, many searchable encryption schemes were proposed. As an example, consider the wildcard search like "2011/??/??" to find the dates from "2011/01/01" to "2011/12/31". These schemes make data searchers enumerate all possible keywords like "2011/01/01", "2011/01/02", ..., and "2011/12/31" or the data owner enumerates all possible patterns the data searcher may query. In this case, the data searcher has to enumerate 365 keywords because we know wildcard characters represent the dates. However, the data searcher has to enumerate more keywords when the data searcher cannot know what wildcard character can be because we have to enumerate all possible characters.

We can perform search by testing whether any character other than wildcard characters matches if we can compare not keywords but characters. Therefore, our challenging task is designing a searchable encryption which supports searches based on character comparison in a secure manner.

1.5 Our Contribution

In this work, we propose a searchable symmetric encryption scheme which supports searches based on the comparison per character. In this paper, we call this search position-specific keyword search.

This search brings us the following advantages:

- We can realize wildcard search efficiently. For example, in [11], we can perform a search for "2011/??/??" by comparing any character other than wildcard characters. In this example, we need 365 trapdoors with the traditional searchable symmetric encryption schemes but only one trapdoor with our scheme.
- We can also realize the wildcard-based fuzzy keyword search proposed by Li et al. [11] efficiently. Given keyword length ℓ and edit distance d, we can decrease index data size from $\mathcal{O}(\ell^d)$ to $\mathcal{O}(1)$.
- We can realize partial matching search with a few indexes because the data searcher can ignore characters other than specified characters. We can realize it with a single index, but we can realize it efficiently if we put the same number of indexes as the length of the specified keyword. For example, we can perform a partial matching search for "dog" by sending several trapdoors "dog", "?dog", ..., "?...?dog" when a single index "housedog" is put in the cloud servers. In this search, given the upper bound of the keyword length u and the keyword length ℓ, the number of trapdoors that must be sent to the cloud server is $u - \ell$. On the other hand, we can perform a partial matching search for "dog" by sending a single trapdoor if eight indexes "housedog", "ousedog", "usedog", ..., "g" are put in the cloud servers because the indexes contain "dog". Another example is the indexes for "doggy". In this example, we put five indexes "doggy", ..., "y", in the cloud server and we can also perform a partial matching search for "dog" by sending a single trapdoor because "doggy" matches "dog" by ignoring characters other than specified characters.

Table 1. Comparison of Search Functions

Name	Key	Search type
our scheme	symmetric	searches with specified characters
Song et al's scheme [15]	symmetric	equality search
Li et al.'s scheme [11]	symmetric	fuzzy keyword search
Z-IDX [8]	symmetric	full-text search
Watanabe et al.'s scheme [16]	symmetric	relational database search
PEKS [4]	asymmetric	equality search
Sedghi et al.'s scheme [14]	asymmetric	wildcard search

We can execute our scheme faster than asymmetric ones because we do not need complex computations like pairing used by asymmetric ones. Furthermore, we can perform searches efficiently without testing all indexes on the server side as explained in Section 5.1.

In our construction, we use pseudo-random functions for security improvement. The pseudo-random functions can conceal the information because no efficient algorithm can distinguish an output of pseudo-random functions from an output of random functions. We also use the Bloom filter to decrease the data size of the output of pseudo-random functions because pseudo-random functions needs some bits (e.g., 256 bits with HMAC-SHA256) for a single input, but one Bloom filter with the same length can contain multiple outputs of pseudo-random functions.

Note that Goh's scheme [8] and Watanabe et al.'s scheme [16] already use the Bloom filter only for an index, whereas we use the Bloom filter not only for an index but also for a trapdoor. We cannot apply this our technique to the Goh's original scheme and Watanabe et al.'s scheme, so the data size of the trapdoor cannot be decreased. Although we can apply this our technique to Goh's second scheme described in the appendix of [8], its applicability was not mentioned in [8] and the security proof of the second scheme was not given explicitly in [8].

1.6 Comparison with Existing Works

Let the keyword length be ℓ, and edit distance d for fuzzy keyword search.

We show the comparison of search functions in Table 1, and the comparison of data size and run time to generate an index and a trapdoor with Goh's scheme, Song et al.'s scheme and Li et al.'s scheme in Table 2. In Table 2, "single index" and "single trapdoor" are two methods described in Section 1.5. Note that although we show the orders of the index data size in Table 2 together, these orders have the different meanings because an index of Goh's scheme consists of multiple keywords, and an index of our scheme consists of a single keyword.

We can see the decrease in data size and run time compared with existing schemes in Table 2. In particular, we can decrease the index size in the fuzzy keyword search. Since our scheme is not designed for full-text search, Goh's scheme is more space- and computation-efficient than our scheme when we

Table 2. Efficiency Comparison

Type of search	Name	Data size Index	Data size Trapdoor	Run time Index	Run time Trapdoor
equality search	our scheme	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\ell)$	$\mathcal{O}(\ell)$
	Song et al's scheme [15]	$\mathcal{O}(\ell)$	$\mathcal{O}(\ell)$	$\mathcal{O}(\ell)$	$\mathcal{O}(\ell)$
	Goh's scheme [8]	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
fuzzy keyword search	our scheme	$\mathcal{O}(1)$	$\mathcal{O}(\ell^d)$	$\mathcal{O}(\ell)$	$\mathcal{O}(\ell^{d+1})$
	Song et al's scheme [15]	—	—	—	—
	Li et al's scheme [11]	$\mathcal{O}(\ell^d)$	$\mathcal{O}(\ell^d)$	$\mathcal{O}(\ell^d)$	$\mathcal{O}(\ell^d)$
partial matching search	our scheme (single index)	$\mathcal{O}(1)$	$\mathcal{O}(u)$	$\mathcal{O}(\ell)$	$\mathcal{O}(u\ell)$
	our scheme (single trapdoor)	$\mathcal{O}(\ell)$	$\mathcal{O}(1)$	$\mathcal{O}(\ell^2)$	$\mathcal{O}(\ell)$
	Song et al's scheme [15]	—	—	—	—
	Li et al's scheme [11]	—	—	—	—

(ℓ: keyword length, d: edit distance, u: upper bound of keyword length, n: number of keywords in one file).

perform a full-text search. However, our scheme is more space-efficient than the existing schemes when we perform more complex search like a fuzzy keyword search. c

2 Preliminaries

Notations

We use the following notations in this paper.

Symmetric set difference. We define a symmetric set difference $A \triangle B$ as $A \triangle B = (A - B) \cup (B - A)$.

Random number. $x \overset{R}{\leftarrow} S$ denotes random variable x chosen at random from the set S.

The number of elements. We use $|A|$ to denote the number of elements in A

String concatenation. $a \parallel b$ denotes concatenation of two strings a, b.

Character. $w[n]$ denotes n-th character in string w.

Logical operations. Given two logical values a, b, $a \wedge b$ denotes the logical AND between a and b, and $a \vee b$ denotes the logical OR between a and b.

Pseudo-random Functions

A pseudo-random function is computationally indistinguishable from a random function. To be more specific, we call a function $f\colon \{0,1\}^n \times \{0,1\}^s \to \{0,1\}^m$ which has the following features as (t, ϵ, q)-pseudo-random function.

- Given an input $x \in \{0,1\}^n$ and a key $sk \in \{0,1\}^s$, $f(x, sk)$ can be computed efficiently.
- For any t time oracle algorithm $\mathcal{A}$ with at most q adaptive queries,

$$|\Pr[\mathcal{A}^{f(\cdot,sk)} = 0 | sk \overset{R}{\leftarrow} \{0,1\}^s] - \Pr[\mathcal{A}^g = 0 | g \overset{R}{\leftarrow} \{F\colon \{0,1\}^n \to \{0,1\}^m\}]| < \epsilon.$$

In this paper, we use a keyed hash function like HMAC-SHA256 where the key *sk* is shared among a data owner and data searchers.

Symmetric Key Encryption

We use $\Pi = (Setup(1^\lambda), Enc(sk,\cdot), Dec(sk,\cdot))$ to denote a symmetric key encryption scheme. Given a security parameter λ, $Setup(1^\lambda)$ outputs a secret key. Given a secret key sk, $Enc(sk,\cdot)$ and $Dec(sk,\cdot)$ are an encryption and decryption scheme with sk.

Bloom Filter

Bloom filter [3] is space-efficient probabilistic data structure. Bloom filter has a false positive. It means that Bloom filter may output true even if it does not have the element to be checked. On the other hand, Bloom filter does not have a false negative. It means the element is guaranteed not to be in Bloom filter if Bloom filter outputs false.

Bloom filter is denoted as m-bit array with address from 1 to m. First, this bit array is initialized with 0. To add an element to Bloom filter, we compute k addresses $a_1, a_2, ..., a_k$ with k hash functions $h_1, h_2, ..., h_k$ first. Then, we make the bits corresponding to the addresses be 1. To determine if Bloom filter has the element, we also compute k addresses $a'_1, a'_2, ..., a'_k$. Then, we check if all bits corresponding to the addresses are 1. We can know with certain error rate that the element is in the Bloom filter if and only if all bits are 1. We can also without error know the element is not in Bloom filter if some bits are 0.

Search Expression

In this paper, we express a position-specific keyword search as a DNF logical formula $p = (p_{(1,1)} \wedge ... \wedge p_{(1,m_1)}) \vee ... \vee (p_{(n,1)} \wedge ... \wedge p_{(n,m_n)})$. In this formula, $p_{(i,j)} (i \in [1,n], j \in [1,m_i])$ denotes the logical formula which compares the character in the specified position, and let $w[x] = c$ denote a comparison which checks if x-th character of w is c.

We call this formula p as search expression in this paper.

For example, the search expression p to perform an equality search for either keyword "dog" or "cat" is as follows:

$$\begin{aligned} f = &((w[1] = \text{"d"}) \wedge (w[2] = \text{"o"}) \wedge (w[3] = \text{"g"}) \wedge (w[4] = null)) \\ &\vee((w[1] = \text{"c"}) \wedge (w[2] = \text{"a"}) \wedge (w[3] = \text{"t"}) \wedge (w[4] = null)) \end{aligned}$$

File Identifier

File identifier means the unique name of a file. We can use an absolute path for the file, a universal unique identifier (UUID) [10], and so on. We can use any type of file identifier.

3 Proposed Scheme

We use one Bloom filter to store all characters of one keyword. We also use pseudo-random functions when we add a character to Bloom filter and symmetric key encryption scheme to encrypt data for the search results. We can use any secure symmetric key encryption scheme.

We have to specify an upper bound u of the keyword length before use.

Note that each keyword is terminated with a *null* which denotes a null symbol. *null* denotes the end of the keyword and is used for the query including the end of the keyword.

This scheme has the following four algorithms.

KeyGen(1^λ) Given a security parameter λ, output a secret key $sk \overset{R}{\leftarrow} \{0,1\}^\lambda$.

Trapdoor(sk, p) Given a secret key sk and a search expression p, output a trapdoor for p. Let $p = p_1 \vee ... \vee p_n$, $p_i = (p_{(i,1)} \wedge ... \wedge p_{(i,m_i)})$, where $p_{(i,j)}$ denotes a comparison which checks if $w[x_{(i,j)}] = c_{(i,j)}$. We use $T = \{T_1, ..., T_n\}$ to denote a trapdoor for p. We can compute T_i for $i \in [1, n]$ as follows.

1. Initialize a Bloom filter T_i.
2. For each term $p_{(i,j)}$ for $j \in [1, m_i]$, given $p_{(i,j)}$ denotes a comparison which checks if $w[k] = c$, add a concatenated string $k \parallel c$ to the Bloom filter T_i.

BuildIndex(sk, FID_w, w) Given a secret key sk, a file identifier FID_wCand a keyword w, compute an index $\mathcal{I} = \{\mathcal{I}_I, \mathcal{I}_{II}\}$ for w as follows.

1. Initialize a Bloom filter $\mathcal{I}_\mathrm{I}$.
2. For each character $w[i]$ for $i \in [1, |w|]$, add a concatenated string $i \parallel w[i]$ to $\mathcal{I}_\mathrm{I}$.
3. Let the number of pseudo-random functions used for the Bloom filter $\mathcal{I}_\mathrm{I}$ be k. Pick $(u - |w|) \cdot k$ random values and set the respective bits of the Bloom filter to 1. This operation is equivalent to an insertion of $u - |w|$ random characters, where $u - |w|$ is the difference between the actual length and the upper bound, into the Bloom filter. We have this operation to prevent the number of 1's in $\mathcal{I}_\mathrm{I}$ from revealing the length of the keyword w.
4. Choose a random value $rd \overset{R}{\leftarrow} \{0,1\}^\lambda$. This value is used to randomize $\mathcal{I}_\mathrm{II}$.
5. Encrypt a collection of a file identifier FID_w, a keyword w, and rd with a symmetric key encryption scheme as $\mathcal{I}_\mathrm{II} = \mathrm{Enc}(sk, \mathrm{FID}_w \parallel w \parallel rd)$. This value is used to query the file and to check if the search result is correct. rd randomizes $\mathcal{I}_\mathrm{II}$ to hide the equality of the keyword.
6. Output the index $\mathcal{I} = \{\mathcal{I}_\mathrm{I}, \mathcal{I}_\mathrm{II}\}$.

This algorithm is executed as many times as the number of keywords.

SearchIndex(T, $\mathcal{I}$) Given a trapdoor T for a certain keyword and an index $\mathcal{I}$, search the indexes in which all bits set to 1 in trapdoor are 1 and output the search result $\mathcal{I}_\mathrm{II}$ in the index $\mathcal{I}$.

We describe the information flow of this scheme as follows.

1. Those who share files share a key generated by KeyGen before using this scheme.
2. A data owner encrypts a file with Enc. The data owner specifies keywords and computes indexes of them with BuildIndex.
3. The data owner sends the encrypted file and index.
4. A server stores them in the database.
5. A data searcher computes a trapdoor for a specified keyword with Trapdoor.
6. The data searcher sends it to the server.
7. The server searches with SearchIndex and returns the result.

We describe the concrete examples of the outputs of BuildIndex and Trapdoor in Appendix B that will help understand our scheme intuitively.

Optimization

We can use a linear search to execute SearchIndex. However, we can achieve more efficient search with a binary tree search because we do not have to test all indexes in a binary tree search. See Appendix C for the details.

Determining Suitable Parameters for Bloom Filter

We have to determine the length of Bloom filter m and the number of pseudo-random functions k before use. Given the maximum number of characters n and the acceptable possibility of false-positive f_p, we can determine these parameters as follows [8]:

$$k = -\log_2 f_p, m = \frac{kn}{\ln 2} \tag{1}$$

4 Security Analysis

4.1 Security Model

The security model we use is based on IND-CKA [8]. We define the indistinguishability of keywords. Although Z-IDX [8] creates indexes from the set of keywords, our scheme creates indexes from one keyword. We call this security model IND-CPSKA[1] to make the difference clear.

This security model is defined by the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ as follows. We say that an adversary $\mathcal{A}$ (t, ϵ, q)-breaks our scheme if $\mathrm{Adv}_{\mathcal{A}}$ is at least ϵ after $\mathcal{A}$ takes at most t time and query trapdoors to $\mathcal{C}$ q times. We say that the symmetric searchable encryption $\mathcal{I}$ is (t, ϵ, q)-IND-CPSKA secure if there is no adversary who can (t, ϵ, q)-break $\mathcal{I}$.

[1] IND-CPSKA denotes Indistinguishability under Chosen Position-Specific Keyword Attack.

Setup. $\mathcal{C}$ creates a set S of q pairs of position and character, and gives this to $\mathcal{A}$ D $\mathcal{A}$ chooses some subsets S^*, that is, keywords from S and gives this to $\mathcal{C}$. After receiving S^*, $\mathcal{C}$ runs KeyGen to generate a secret key K_{priv}, and $\mathcal{C}$ computes the indexes for all subsets of S^* with BuildIndex. Finally, $\mathcal{C}$ gives all indexes and related subsets S^* to $\mathcal{A}$ after computing all indexes. We note that the correspondence relation between the indexes and S^* is unknown to $\mathcal{A}$.

Query. $\mathcal{A}$ can query trapdoor T_x for word x to $\mathcal{C}$. For each index $\mathcal{I}$, $\mathcal{A}$ can execute SearchIndex for $T_x, \mathcal{I}$ to tell whether $\mathcal{I}$ matches x.

Challenge. $\mathcal{A}$ picks nonempty two subsets $V_0, V_1 \in S^*$ such that $|V_0 - V_1| \neq 0$, $|V_1 - V_0| \neq 0$ and $|V_0| = |V_1|$. Here, $\mathcal{A}$ must not have queried $\mathcal{C}$ for the trapdoor of any character in $V_0 \triangle V_1$. $\mathcal{A}$ cannot query any trapdoor for a character in $V_0 \triangle V_1$. $\mathcal{A}$ gives V_0 and V_1 to $\mathcal{C}$. $\mathcal{C}$ chooses b from $\{0, 1\}$ at random. $\mathcal{C}$ computes BuildIndex(V_b, K_{priv}) to get an index corresponding to V_b and gives it back to $\mathcal{A}$. After $\mathcal{C}$ gives the challenge (i.e., BuildIndex(V_b, K_{priv})) to $\mathcal{A}$, $\mathcal{A}$ cannot query any trapdoor for any character $x \in V_0 \triangle V_1$ to $\mathcal{C}$.

Response. $\mathcal{A}$ outputs b' to guess b. The advantage $\mathcal{A}$ obtains is defined as $\text{Adv}_{\mathcal{A}} = |Pr[b = b'] - 1/2|$.

4.2 Security Proof

Theorem 1 *Given the number of pseudo-random functions k, our scheme is $(t, \epsilon, q/k)$-IND-CPSKA secure if f is a (t, ϵ, q)-pseudo-random function.*

Proof. We can prove this theorem as the proof in [8].

We prove this theorem using its contrapositive. Suppose our scheme is not $(t, \epsilon, q/k)$-IND-CPSKA secure, that is, there is an algorithm $\mathcal{A}$ which $(t, \epsilon, q/k)$-breaks our scheme. Then we show we can construct the algorithm $\mathcal{B}$ which distinguishes whether f is a pseudo-random function or a random function. Given $x \in \{0, 1\}^n$, $\mathcal{B}$ can use an oracle $\mathcal{O}_f$ which outputs $f(x) \in \{0, 1\}^s$ for unknown function f. $\mathcal{B}$ evaluates f with a query to $\mathcal{O}_f$ whenever computing any four index algorithms.

The algorithm $\mathcal{B}$ makes the simulation for $\mathcal{A}$ as follows.

Setup. $\mathcal{B}$ chooses a set S of q/k pairs of position and character from $\{0, 1\}^n$ at random and sends it to $\mathcal{A}$. $\mathcal{A}$ returns a collection S^* of polynomial numbers of subsets. For each subset D of S^*, $\mathcal{B}$ assigns a file identifier FID$_D$ and gets $\mathcal{I}_{\text{FID}_D}$ by computing BuildIndex for FID$_D$. $\mathcal{B}$ gives all indexes and related subsets S^* to $\mathcal{A}$ after computing all indexes. We note that correspondence relation between the indexes and S^* is unknown to $\mathcal{A}$.

Query. $\mathcal{B}$ computes Trapdoor for x and returns a trapdoor T_x for x.

Challenge. $\mathcal{A}$ picks nonempty subset $V_0, V_1 \in S^*$ such that $|V_0 - V_1| \neq 0$, $|V_1 - V_0| \neq 0$ and $|V_0| = |V_1|$. $\mathcal{A}$ cannot query any trapdoor for a character in $V_0 \triangle V_1$ to $\mathcal{B}$. $\mathcal{A}$ gives V_0 and V_1 to $\mathcal{B}$. $\mathcal{B}$ chooses b from $\{0, 1\}$ and a file identifier V_{id} at random. $\mathcal{B}$ computes BuildIndex to get $\mathcal{I}_{V_b}$ and gives $\mathcal{I}_{V_b}$ to $\mathcal{A}$. The challenge to $\mathcal{A}$ is to guess b. $\mathcal{A}$ cannot query any trapdoor which contains any character $x \in V_0 \triangle V_1$.

Response. $\mathcal{A}$ outputs b' finally. $\mathcal{B}$ outputs 0 if $b = b'$, that is, f is a pseudo-random function. Otherwise, $\mathcal{B}$ outputs 1.

$\mathcal{B}$ takes at most t time because $\mathcal{A}$ takes at most t time. $\mathcal{B}$ sends at most q queries to $\mathcal{O}_f$ because there are only q/k characters, $\mathcal{A}$ creates at most q/k queries, and $\mathcal{B}$ creates k queries for $\mathcal{A}$'s single query.

Finally, according to the following lemmas, $\mathcal{B}$ has advantage greater than ϵ to determines if the unknown function f is a pseudo-random function or f is a random-function because we have the following equation:

$$|\Pr[\mathcal{B}^{f(\cdot,k)} = 0 | k \overset{R}{\leftarrow} \{0,1\}^s] - \Pr[\mathcal{B}^g = 0 | g \overset{R}{\leftarrow} \{F : \{0,1\}^n \rightarrow \{0,1\}^s\}]| \geq \epsilon$$

This contradicts the assumption of pseudo-random functions.

Therefore, our scheme is $(t, \epsilon, q/k)$-IND-CPSKA secure if f is a (t, ϵ, q)-pseudo random function.

Lemma 1. $|\Pr[\mathcal{B}^{f(\cdot,k)} = 0 | k \overset{R}{\leftarrow} \{0,1\}^s] - \frac{1}{2}|| \geq \epsilon$ *if f is a pseudo-random function.*

Lemma 2. $\Pr[\mathcal{B}^g = 0 | g \overset{R}{\leftarrow} \{F : \{0,1\}^n \rightarrow \{0,1\}^s\}] = \frac{1}{2}$ *if g is a random function.*

Proof. Lemma 1 is obvious because $\mathcal{B}$ simulates $\mathcal{C}$ completely in an IND-CPSKA game if f is a pseudo-random function.

We prove Lemma 2. We have to consider only Challenge subsets V_0, V_1 because other subsets in S do not reveal any information about the Challenge subsets.

Without loss of generality, assume that $V_0 \triangle V_1$ has two characters x, y such that $x \in V_0, y \in V_1$ and $\mathcal{A}$ guesses b with advantage δ. Given $f(z)$, it means that $\mathcal{A}$ can determine if $z = x$ or $z = y$ with advantage δ, that is, $\mathcal{A}$ can distinguish the output of a random function f with advantage δ. However, if f is a random function, $\mathcal{A}$ cannot distinguish the output, so we have $\delta = 0$. Therefore, $\mathcal{A}$ can guess b with the probability of at best 1/2. Finally, we proved Lemma 2. $\square$

4.3 Limitation

In this scheme, the trapdoor is divided into the clauses. Therefore, the server can know the search result of each clause. For example, in an equality search for "dog" or "cat", the server cannot know the plaintext "dog" and "cat", but the server can know the search result for "dog" and the search result for "cat" respectively.

This limitation exists in many of the existing works as well as this work. For example, Goh's scheme [8] generates a trapdoor per keyword and the server can know the search result for each keyword. Similarly, Li et al.'s scheme [11] generates a trapdoor set per keyword and the server can know the search result for not only each keyword but also each clause.

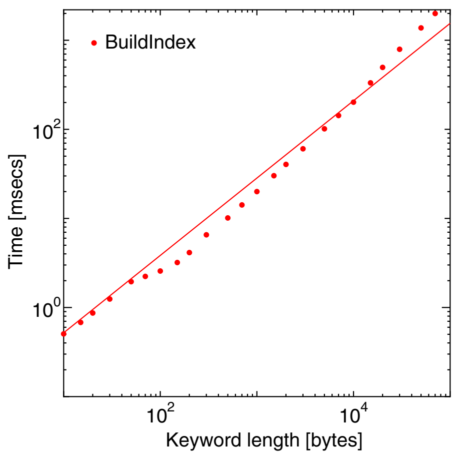

Fig. 2. Run Time for BuildIndex

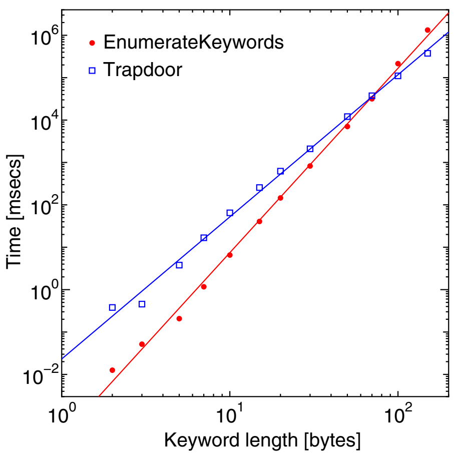

Fig. 3. Run Time for Search

5 Evaluation

Given a bit length of Bloom filter m, an index consists of m-bit Bloom filter and $\mathcal{I}_{\text{II}}$ in $\mathcal{I}$. Given the number of terms divided by disjunctions in the search expression n_ℓ, the size of trapdoor is $n_\ell m$ bits because it has n_ℓ Bloom filters. Given the total number of characters in all keywords ℓ_m, the execution of BuildIndex needs $\mathcal{O}(\ell_m)$ time because we have to compute $\mathcal{O}(\ell_m)$ pseudo-random functions. Similarly, given the number of terms in search expression n_t, the execution of Trapdoor needs $\mathcal{O}(n_t)$ time.

5.1 Implementation

We implemented our scheme with fuzzy keyword search [11] on a 2.8 GHz Intel Core 2 Duo CPU. We used 256-bit Bloom filter, symmetric key encryption scheme AES [12], and keyed hash function HMAC-SHA256 [9,13]. HMAC-SHA256 can be used as distinct pseudo-random functions as $f(sk, x)$, $f_i(sk, x) = f(sk, i \parallel x)$.

The run time for BuildIndex is shown in Figure 2 and the run time for Trapdoor is shown in Figure 3. *EnumerateKeywords* is an algorithm to enumerate keywords proposed by Li et al. We can see in the figure that the larger the keyword length becomes, the more time we need, and our scheme is unsuitable for very long keyword. However, it is not a practical problem because a keyword does not often have a large number of characters.

We implemented *SearchIndex* in the following two ways.

Method 1. This method is based on complete search with a relational database SQLite. The server stores indexes divided into 64 bits. Trapdoor is also divided into 64 bits. Given divided indexes $\text{Idx}_1, ..., \text{Idx}_n$ and trapdoors

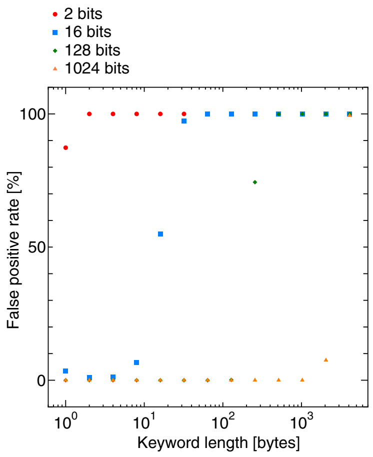

Fig. 4. False-Positive Rate

$\mathrm{Td}_1, ..., \mathrm{Td}_n$, a query for the trapdoor can be constructed as $((\mathrm{Idx}_1\&\mathrm{Td}_1) = \mathrm{Td}_1) \wedge ... \wedge ((\mathrm{Idx}_1\&\mathrm{Td}_n) = \mathrm{Td}_n)$. We divide indexes and trapdoors because the maximum length SQLite can compute AND operation is 64 bits. This is an implementation problem. We do not have to divide indexes and trapdoors if the database supports AND operation of larger bits.

Method 2. This method is based on binary tree search. Each bit of indexes corresponds to a link of the tree. The leaf node has the search result. In this method, we construct a binary tree from the indexes generated with the same key. This search is realized by following the link recursively. We can ignore the link of 0 corresponding to the bit of 0. See Appendix C for the details. Therefore, we do not have to go through the whole tree, and we can achieve more efficient search than the sequential exhaustive search.

A fuzzy keyword search with 6-byte keyword on the database which has 1,000,000 keywords takes 44 seconds on average when we use Method 1. However, it takes only 280 milliseconds on average when we use Method 2.

5.2 Bloom Filter Parameters v.s. False-Positive Rate

We performed an experiment to clarify how the Bloom filter parameters affect the false-positive rate. In this experiment, we used HMAC-SHA256 [9,13], and the number of pseudo-random functions is 3 (fixed).

We performed this experiment as follows. First, we generate a random keyword kw_{idx} and generate an index $\mathrm{Idx}_{kw_{idx}}$ for kw_{idx}. Next, we generate another random keyword kw_{td} ($kw_{idx} \neq kw_{td}$) and generate a trapdoor $\mathrm{Td}_{kw_{td}}$. Finally, we check if $\mathrm{Idx}_{kw_{idx}}$ matches $\mathrm{Td}_{kw_{td}}$. If $\mathrm{Idx}_{kw_{idx}}$ matches $\mathrm{Td}_{kw_{td}}$, this result is a false-positive because $kw_{idx} \neq kw_{td}$.

We show the result of this experiment in Figure 4. In this figure, the x-axis is a keyword length of kw_{idx} and kw_{td}, the y-axis is a false-positive rate, and the graph legends are the bit lengths of the Bloom filter.

This figure shows that an m-bit Bloom filter is effective with small false-positive rates up to m-byte keyword.

6 Conclusion

In this work, we proposed a searchable symmetric encryption scheme which supports a variety of searches by enabling comparison per character as searches for plaintexts. We implemented our scheme and we confirmed that our scheme can be performed on both of client and server in practical time.

Our future work is to find a scheme which creates a single index for the multiple keywords or single trapdoor for multiple terms divided by the disjunctions to decrease data size, run time or information obtained by the server.

Acknowledgments. This work is partially supported by Grant-in-Aid for Young Scientists (B) (23700021), Japan Society for the Promotion of Science (JSPS). This work is also partially supported by the Telecommunications Advancement Foundation (TAF).

References

1. Abdalla, M., Bellare, M., Catalano, D., Kiltz, E., Kohno, T., Lange, T., Malone-Lee, J., Neven, G., Paillier, P., Shi, H.: Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. Journal of Cryptology 21, 350–391 (2008)
2. Bao, F., Deng, R.H., Ding, X., Yang, Y.: Private Query on Encrypted Data in Multi-user Settings. In: Chen, L., Mu, Y., Susilo, W. (eds.) ISPEC 2008. LNCS, vol. 4991, pp. 71–85. Springer, Heidelberg (2008)
3. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 422–426 (1970)
4. Boneh, D., Di Crescenzo, G., Ostrovsky, R., Persiano, G.: Public Key Encryption with Keyword Search. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 506–522. Springer, Heidelberg (2004)
5. Boneh, D., Waters, B.: Conjunctive, Subset, and Range Queries on Encrypted Data. In: Vadhan, S. (ed.) TCC 2007. LNCS, vol. 4392, pp. 535–554. Springer, Heidelberg (2007)
6. Chang, Y.-C., Mitzenmacher, M.: Privacy Preserving Keyword Searches on Remote Encrypted Data. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005)
7. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, pp. 79–88. ACM, New York (2006)
8. Goh, E.J.: Secure indexes. Cryptology ePrint Archive, Report 2003/216 (2003), `http://eprint.iacr.org/2003/216/`

9. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational) (February 1997)
10. Leach, P., Mealling, M., Salz, R.: RFC 4122: A universally unique identifier (UUID) URN namespace (2005)
11. Li, J., Wang, Q., Wang, C., Cao, N., Ren, K., Lou, W.: Fuzzy keyword search over encrypted data in cloud computing. In: 2010 Proceedings of the IEEE INFOCOM, pp. 1–5 (March 2010)
12. NIST: Announcing the advanced encryption standards (AES). Federal Information Processing Standards Publication 197 (2001)
13. NIST: Announcing the secure hash standard. Federal Information Processing Standards Publication 180-2 (2002)
14. Sedghi, S., van Liesdonk, P., Nikova, S., Hartel, P., Jonker, W.: Searching Keywords with Wildcards on Encrypted Data. In: Garay, J.A., De Prisco, R. (eds.) SCN 2010. LNCS, vol. 6280, pp. 138–153. Springer, Heidelberg (2010)
15. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of the IEEE Symposium on Security and Privacy, S&P 2000, pp. 44–55 (2000)
16. Watanabe, C., Arai, Y.: Privacy-Preserving Queries for a DAS Model Using Encrypted Bloom Filter. In: Zhou, X., Yokota, H., Deng, K., Liu, Q. (eds.) DASFAA 2009. LNCS, vol. 5463, pp. 491–495. Springer, Heidelberg (2009)
17. Waters, B.R., Balfanz, D., Durfee, G., Smetters, D.K.: Building an encrypted and searchable audit log. In: The 11th Annual Network and Distributed System Security Symposium (2004)

A Pseudocode

We show the algorithms of our scheme with pseudocodes.

In these algorithms, we use m to denote the bit length of a Bloom filter and k to denote the number of pseudo-random functions.

In Algorithm 2, the search expression se is denoted as
$((x_{(1,1)}, c_{(1,1)}), ..., (x_{(1,m_1)}, c_{(1,m_1)})), ..., ((x_{(n,1)}, c_{(n,1)}), ..., (x_{(n,m_n)}, c_{(n,m_n)}))$
in order to denote it as an array. For example, $((1, d), (2, o), (3, g), (4, null))$ denotes an equality search for "dog" only in this section. $|se|$ denotes the number of elements of the array se.

B Execution Example of Proposed Scheme

We describe the concrete examples of the outputs of BuildIndex and Trapdoor in this section. Suppose that Alice outsources data and Bob searches data. Alice specifies "dog" to denote the content of the file. Bob searches "dog" by an equality search. The bit length of the Bloom filters is 16 bits, the number of pseudo-random functions is 2, and the upper bound of the keyword length is 5. Suppose that they share these parameters.

1. First, Alice executes KeyGen and sends sk to Bob in a secure manner.
2. Alice executes BuildIndex as follows:

Algorithm 1. BuildIndex

```
Require: $w$ is a keyword to build an index.
Ensure: $(BF, c)$ is an index for $w$.
  Initialize a bit array $BF$ with zeros.
  for $i = 1$ to $|w|$ do
    for $j = 1$ to $k$ do
      $p \leftarrow f_j(i \parallel w[i])$.
      $BF[p] \leftarrow 1$.
    end for
    for $j = 1$ to $(u - |w|) \cdot k$ do
      Pick $rd \in [1, |BF|]$ at random.
      $BF[rd] \leftarrow 1$.
    end for
  end for
  Encrypt $FID_w \parallel w \parallel rd$ and assign it to $c$.
  Return $(BF, c)$.
```

Algorithm 2. Trapdoor

```
Require: $se$ is a search expression to generate a trapdoor.
Ensure: $t$ is a trapdoor for $se$.
  $t \leftarrow \{\}$.
  for $i = 1$ to $|se|$ do
    Initialize a bit array $BF_i$ with zeros.
    for $j = 1$ to $|se[i]|$ do
      for $h = 1$ to $k$ do
        $p \leftarrow f_h(se[i][j][1] \parallel se[i][j][2])$.
        $BF_i[p] \leftarrow 1$.
      end for
    end for
    Append $BF_i$ to $t$.
  end for
  Return $t$.
```

Algorithm 3. SearchIndex

```
Require: $td$ is a trapdoor and $idx$ is an index.
Ensure: Output ‘true’ if $idx$ matches $trapdoor$, otherwise return ‘false.’
  for $i = 1$ to $m$ do
    if $td[i] = 1$ and $idx[i] = 0$ then
      Return ‘false.’
    end if
  end for
  Return ‘true.’
```

   (a) Alice creates a 16-bit Bloom filter. The Bloom filter is 0000000000000000 at this point.
   (b) Alice computes two pseudo-random functions for the respective characters of "1 $\|$ d", "2 $\|$ o", "3 $\|$ g"C"4 $\|$ *null*". Suppose that Alice obtains 6, 2 for "1 $\|$ d", 12, 4 for "2 $\|$ o", 5, 13 for "$\|$ g" and 2, 9 for "4 $\|$ *null*". Alice sets the respective bits to 1. The resultant Bloom filter is 0101110010011000 at this point. Alice picks two random values because $(u - |w|) \cdot k = (5 - 4) \cdot 2 = 2$. Suppose that Alice obtains 15 and 4. Alice sets the respective bits to 1. The final Bloom filter is 0101110010011010 at this point. This value is $\mathcal{I}_{\mathrm{I}}$.
   (c) Alice picks random value rd and computes $\mathcal{I}_{\mathrm{II}} = \mathrm{Enc}(sk, \mathrm{FID}_w \parallel w \parallel rd)$.

3. Alice sends $\mathcal{I} = (\mathcal{I}_{\mathrm{I}}, \mathcal{I}_{\mathrm{II}})$ to the server.
4. Bob executes Trapdoor to perform an equality search. The search expression is $(w[1] = \text{"d"}) \wedge (w[2] = \text{"o"}) \wedge (w[3] = \text{"g"}) \wedge (w[4] = null)$. This search expression means that the searched keyword equals to "dog" including the terminal symbol. Bob computes two pseudo-random functions for the respective characters of "1 $\|$ d", "2 $\|$ o", "3 $\|$ g"C"4 $\|$ *null*". These values are the same as those Alice computed. Bob sets the respective bits to 1. The Bloom filter is 0101110010011000 at this point.
5. Bob sends this value to the server.
6. The server searches the indexes in which all bits set to 1 in trapdoor are 1 and returns the search result $\mathcal{I}_{\mathrm{II}}$ in the index $\mathcal{I}$ to Bob. This was generated by Alice in Step 2.
7. Bob decrypts $\mathcal{I}_{\mathrm{II}}$ and confirms whether the original keyword is "dog".
8. Bob queries the file whose identifier is FID_w.
9. The server returns the queried file.

The trapdoor in the example contains the termination character. Therefore, even if Alice registers "doggy", "doggy" does not hit and only "dog" hits because $w[4] \neq null$. Therefore, Bob can perform an exact equality search.

In this example, although we achieve an equality search, we can achieve a wildcard search for "d?g" by computing pseudo-random functions of "1 $\|$ d", "3 $\|$ g", "4 $\|$ *null*" in Step 4. On the other hand, we can achieve a fuzzy keyword search by an enumeration proposed by Li et al. [11]. When we perform a fuzzy keyword search for "dog" with an edit distance 1, the data searcher Bob enumerates "?dog", "?og", "d?g", "do?", "dog?", and Bob executes Trapdoor for each search expression. When we perform a fuzzy keyword search with longer edit distance, we increase the number of wildcard characters like "??dog". In Li et al.'s scheme, the user who computes an index also has to enumerate possible keywords and prepare corresponding indexes. However, our scheme does not require the data owner to enumerate the keywords for a fuzzy keyword search when the data owner computes an index.

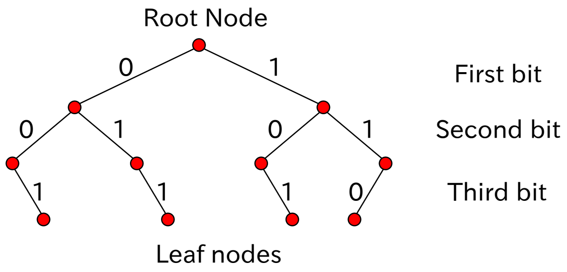

Fig. 5. Binary Search Tree

C Example of Binary Tree Search

We show an example of the binary search tree in Fig. 5. In this example, the tree has four indexes 001, 011, 101 and 110 (i.e., the length of a Bloom filter is three here). The leaf node has the search result.

Suppose that the server receives a trapdoor 101. The server can perform a search as follows:

1. From the root node, follow the link of 1.
2. Follow the link of 0.
3. Follow the link of 1.
4. Obtain the search result from the leaf node corresponding to 101.
5. Go back to the node of Step 2, and follow the link of 1 because the second bit of the trapdoor 101 is 0.
6. Finish the search since there is no more link to follow.