Character-based symmetric searchable encryption and its implementation and experiment on mobile devices

doi: 10.1002/sec.876

Special Issue

Character-based Symmetric Searchable Encryption and Its Implementation and Experiment on Mobile Devices

Takanori Suga[1,3], Takashi Nishide[2*], and Kouichi Sakurai[1]

[1]Kyushu University, 744, Motooka, Nishi-ku, Fukuoka, 819-0395, Japan
[2]University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8577, Japan
[3]Currently working at NEC Corporation, Fuchu, Tokyo, Japan

ABSTRACT

Searchable encryption allows us to perform a keyword search over encrypted data. However, we cannot efficiently perform some complex search (e.g., a wildcard search) with traditional searchable encryption schemes since they can deal with only equality matches. Our symmetric searchable encryption can deal with partial matches. This allows us to efficiently perform a wildcard search, a partial match search, and so on. We also examine the feasibility of our scheme by experiments on a smartphone and tablet, and confirm our scheme can be used in these environments. Availability on portable devices will offer high convenience. Copyright © 0000 John Wiley & Sons, Ltd.

KEYWORDS

Bloom filter, partial-matching search, searchable encryption, symmetric encryption, wildcard search

[*]**Correspondence**

University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8577, Japan

Received . . .

1. INTRODUCTION

1.1. Background

The advance in computer and telecommunication technology made cloud computing widespread. In cloud computing, we let the cloud providers store and process our data. To protect our sensitive data even from cloud providers, we can encrypt our sensitive data before sending them to the server. However, encryption generally prevents us from performing efficient searches.

In recent years, searchable encryption was proposed. We can perform a keyword search over encrypted data with searchable encryption. Like traditional encryption schemes, there exist the symmetric ones and asymmetric ones. We focus on symmetric searchable encryption in this work.

We show an example of the process flow of symmetric searchable encryption. In this example, Alice (called data owner) outsources her data to the server, Bob (called data searcher) performs a keyword search, and Alice and Bob share a symmetric secret key in advance.

1. Alice specifies some keywords that represent the contents of the document.
2. Alice encrypts the document.
3. Alice encrypts the keywords specified in Step 1. We call these encrypted keywords “index” in this paper.
4. Alice sends the encrypted document and the index to the server.
5. The server stores the index and the encrypted document in the database.
6. Bob specifies a keyword to perform a keyword search.

7. Bob encrypts the keyword specified in Step 6. We call this encrypted keyword “trapdoor” in this paper.
8. Bob sends the trapdoor to the server.
9. The server searches the database for documents associated with the keyword by using the trapdoor.
10. The server sends the search results (documents) to Bob.

1.2. Related Work

The first practical searchable encryption scheme was proposed by Song et al. in 2000 [1]. After their proposal, many symmetric schemes were proposed [2, 3, 4, 5, 6, 7]. The first asymmetric searchable encryption scheme was proposed by Boneh et al. [8]. After their proposal, many asymmetric schemes were proposed [9, 10]. Their schemes only support an equality search. To enhance the supported types of searches with existing schemes, we must enumerate all possible keywords when we compute a trapdoor. Suppose that as a keyword we specify the creation date of the document like “2013/01/01” when we compute an index, and we want to search documents created in 2013. Then we must enumerate all dates in 2013 like “2013/01/01”, “2013/01/02”, . . ., “2013/12/31” if we can use only eqaulity search.

In recent years, some schemes that aim to enhance the supported types of searches were proposed. Li et al. proposed the first symmetric searchable encryption scheme that supports a fuzzy keyword search [11]. With a fuzzy keyword search, we can find some similar keywords. For example, we can find keyword “colour” with keyword “color”. Sedghi et al. proposed the first asymmetric searchable encryption scheme that supports a wildcard search [12].

Goh proposed a symmetric searchable encryption with Bloom filter [13] and Watanabe et al. proposed a symmetric searchable encryption with Bloom filter for relational database [14]. Goh’s scheme is more efficient than our scheme when we have many keywords in a document and we perform an equality search. However, our scheme is more efficient than his scheme when we perform some complex search like a wildcard search.

1.3. Our Contribution

In this work, we focus on how to create secure indexes for encrypted documents as in most of the existing schemes. Our scheme [15] has the following advantages:

- Our work is the first symmetric searchable encryption that does not require us to enumerate all possible keywords to perform a wildcard search. That is, as we mentioned above, we need to enumerate all possible similar keywords when we want to perform a wildcard search with all previous schemes except Sedghi et al.’s asymmetric searchable encryption [12].
- Our work can decrease the index size of wildcard-based fuzzy keyword search proposed by Li et al. [11] from $\mathcal{O}(\ell^d)$ to $\mathcal{O}(1)$ where a keyword length ℓ and an edit distance d. In their scheme, both the data owner and the data searcher must enumerate possible similar keywords represented by a wildcard keyword. However, in our scheme, only the data searcher needs to enumerate possible similar keywords represented by a wildcard keyword.
- With our scheme, we can achieve a partial-matching keyword search efficiently.

In general, symmetric searchable encryption schemes can be performed more efficiently than asymmetric ones. Furthermore, we show the efficiency of our scheme by performing it on a tablet and even on a smartphone.

2. PRELIMINARIES

2.1. Notations

We use the following notations in this paper.

Symmetric difference. Given two sets A and B, $A \triangle B$ denotes a symmetric difference $A \triangle B = (A - B) \cup (B - A)$.

Random number. Given a set S, $x \xleftarrow{R} S$ means that x is chosen at random from the set S.

The number of elements. Given a set S, $|S|$ denotes the number of elements in S.

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

DOI: 10.1002/sec

Character. Given a string (or array) w and a position n, $w[n]$ denotes n-th character in the string w.

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

2.2. Pseudo-Random Functions

A pseudo-random function is a function computationally indistinguishable from a random function. To be more precise, given a bit length n of an input, a bit length λ of a secret key and a bit length m of an output, we say $f : \{0,1\}^n \times \{0,1\}^\lambda \to \{0,1\}^m$ is a (t, ϵ, q)-pseudo-random function if it satisfies the following properties:

- Given an input $x \in \{0,1\}^n$ and a secret key $sk \in \{0,1\}^\lambda$, $f(x, sk)$ can be computed efficiently.
- No t time algorithm $\mathcal{B}$ with at most q adaptive oracle queries can distinguish between $f(\cdot, sk)$ and a random function $F : \{0,1\}^n \to \{0,1\}^m$ with an advantage more than ϵ.
  That is, $|\Pr[\mathcal{B}^{f(\cdot,sk)} = 0 | sk \xleftarrow{R} \{0,1\}^\lambda] - \Pr[\mathcal{B}^g = 0 | g \xleftarrow{R} \{F : \{0,1\}^n \to \{0,1\}^m\}]| < \epsilon$.

In this paper, we use a keyed hash function like HMAC-SHA256 [16, 17] as a pseudo-random function.

2.3. Symmetric Key Encryption

We use a symmetric key encryption scheme, denoted as $\Pi = \left(\mathrm{Setup}(1^\lambda), \mathrm{Enc}(sk, \cdot), \mathrm{Dec}(sk, \cdot)\right)$. Given a security parameter λ, $\mathrm{Setup}(1^\lambda)$ outputs a secret key. Given a secret key sk and an input, $\mathrm{Enc}(sk, \cdot)$ encrypts the input and $\mathrm{Dec}(sk, \cdot)$ decrypts the input with the secret key sk.

2.4. Bloom Filter

Bloom filter is a space-efficient probabilistic data structure [18]. We can put some elements into this data structure and can test if the Bloom filter contains some elements. Bloom filter has a false-positive. That is, for example, the Bloom filter that has two elements "foo" and "bar" might say it has "baz". However, Bloom filter does not have a false-negative. That is, the Bloom filter that has two elements "foo" and "bar" never says it does not have "foo".

A Bloom filter is an m-bit array initialized with 0's. To add an element x, we compute k hash functions $h_1(x), ..., h_k(x)$, and we make $h_1(x)$-th bit, ..., and $h_k(x)$-th bit be 1. To test if the Bloom filter has an element x, we also compute k hash functions $h_1(x), ..., h_k(x)$, and check if all bits in the array with the positions $h_1(x), ..., h_k(x)$ are 1's. If all bits are 1's, the Bloom filter probably has the element (though this can be an error). Otherwise, the Bloom filter never has the element.

For each hash function h_i, we will use keyed hash function with a secret key.

2.5. Security Model

The security model we use for a symmetric searchable encryption is based on IND-CKA[*] [13]. We define a security model named IND-CPSKA[†] because our scheme generates indexes from one keyword while Z-IDX [13] generates indexes from a collection of keywords.

This security model is defined by the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ as follows.

Setup. $\mathcal{C}$ picks a set S of q pairs of a position and a character, and sends S to $\mathcal{A}$. $\mathcal{A}$ picks some subsets S', that is, keywords picked from S, and sends S' to $\mathcal{C}$. After $\mathcal{C}$ receives S', $\mathcal{C}$ executes KeyGen to obtain a secret key sk and executes BuildIndex to obtain indexes for all subsets in S'. Finally, $\mathcal{C}$ sends all indexes to $\mathcal{A}$. We note that the information about which keyword (a subset) corresponds to which index is not given to $\mathcal{A}$.

Query. $\mathcal{A}$ is allowed to query trapdoors for some keywords (or search expressions) to $\mathcal{C}$. For each trapdoor T_x for a keyword (or a search expression) x, $\mathcal{A}$ can execute SearchIndex to check if an index $\mathcal{I}$ matches the trapdoor T_x. Here we can assume a search expression x does not include $\vee$ (i.e., a logical OR) because $\mathcal{A}$ can obtain trapdoors for $x_1 \vee x_2$ by obtaining a trapdoor for x_1 and a trapdoor x_2 separately.

Challenge. $\mathcal{A}$ picks nonempty two subsets V_0 and V_1 from S' such that $|V_0 - V_1| \neq 0$, $|V_1 - V_0| \neq 0$ and $|V_0| = |V_1|$. Also V_0, V_1 must satisfy that no trapdoor for a set K of pairs of a position and a character was already queried by $\mathcal{A}$ where $|K \cap (V_0 \triangle V_1)| > 0$ [‡]. Also V_0, V_1 must satisfy that

[*] IND-CKA denotes Indistinguishability against Chosen Keyword Attack.

[†] IND-CPSKA denotes Indistinguishability against Chosen Position-Specific Keyword Attack.

[‡] This is the generalized restriction compared with IND-CKA [13].

no index for a set K is given to $\mathcal{A}$ where $|K \cap (V_0 \triangle V_1)| > 0$ [§].

$\mathcal{A}$ sends V_0 and V_1 to $\mathcal{C}$. $\mathcal{C}$ picks $b \in \{0, 1\}$ at random and sends an index for V_b [¶] to $\mathcal{A}$. After $\mathcal{A}$ receives the index for V_b, $\mathcal{A}$ cannot query the trapdoors that do not follow the restriction mentioned above.

Response. $\mathcal{A}$ outputs b' to guess b. The advantage of $\mathcal{A}$ is defined as $Adv_{\mathcal{A}} = |\Pr[b = b'] - 1/2|$. This is an advantage over the probability that $\mathcal{A}$ guesses by tossing a coin.

We say that an adversary $\mathcal{A}$ (t, ϵ, q)-breaks the symmetric searchable encryption scheme if the advantage of $\mathcal{A}$ $Adv_{\mathcal{A}}$ is at least ϵ after $\mathcal{A}$ takes at most t time and queries trapdoors to the challenger $\mathcal{C}$ at most q times. The symmetric searchable encryption scheme $\mathcal{I}$ is (t, ϵ, q)-IND-CPSKA secure if no adversary can (t, ϵ, q)-break $\mathcal{I}$.

3. PROPOSED SCHEME

In our scheme, we use a Bloom filter per keyword to generate an index or a trapdoor. We also use pseudo-random functions to add elements to Bloom filters. The documents can be encrypted with any encryption scheme (out of scope of this paper).

We assume that there is an upper bound u of the keyword length. We note that each keyword is terminated with $null$, which is an end of the keyword string. $null$ allows us to specify an explicit keyword length in performing a search.

We express a keyword search as a DNF logical formula $p = (p_{(1,1)} \wedge ... \wedge p_{(1,m_1)}) \vee ... \vee (p_{(n,1)} \wedge ... \wedge p_{(n,m_n)})$. For example, we express an equality search for (“dog” OR “cat”) as $p = ((w[1] = \text{‘d’}) \wedge (w[2] = \text{‘o’}) \wedge (w[3] = \text{‘g’}) \wedge (w[4] = null)) \vee ((w[1] = \text{‘c’}) \wedge (w[2] = \text{‘a’}) \wedge (w[3] = \text{‘t’}) \wedge (w[4] = null))$. For this search expression (“dog” OR “cat”), we will generate two Bloom filters and as the search results we can obtain encrypted documents associated with keywords “dog” or “cat”.

We show four algorithms of our scheme.

KeyGen(1^λ) This algorithm is a key generator. Given a security parameter λ, this outputs a secret key $sk \xleftarrow{R} \{0, 1\}^\lambda$.

BuildIndex(sk, FID, w) This algorithm is used to generate an index. Given a secret key sk, a file identifier FID and a keyword w, generate an index as follows:

1. Initialize a Bloom filter $\mathcal{I}_{\mathrm{I}}$, that is, initialize a bit array with 0’s.
2. For each $i \in \{1, |w|\}$, add an element $i \parallel w[i]$ (i.e., a pair of a position and a character[‖]) to the Bloom filter $\mathcal{I}_{\mathrm{I}}$ by using keyed hash functions with sk. That is, if the keyword w is “dog” for example, w is viewed as a search expression $p = ((w[1] = \text{‘d’}) \wedge (w[2] = \text{‘o’}) \wedge (w[3] = \text{‘g’}) \wedge (w[4] = null))$.
3. Given an upper bound u of keyword lengths, add $(u - |w|)$ random elements to the Bloom filter $\mathcal{I}_{\mathrm{I}}$ to conceal the keyword length. Given a bit length m of Bloom filter and the number of pseudo-random functions k of the Bloom filter, we can set $(u - |w|) \cdot k$ random bits to 1 in the Bloom filter $\mathcal{I}_{\mathrm{I}}$ instead of adding $(u - |w|)$ random elements.
4. Encrypt a concatenated string FID $\parallel w$ as $\mathcal{I}_{\mathrm{II}} = \mathrm{Enc}_{sk}(\mathrm{FID} \parallel w)$.
5. Output the index $\mathcal{I} = (\mathrm{FID}, \mathcal{I}_{\mathrm{I}}, \mathcal{I}_{\mathrm{II}})$.

Trapdoor(sk, p) This algorithm is used to generate a trapdoor. Given a secret key sk and a search expression $p = (p_{(1,1)} \wedge ... \wedge p_{(1,m_1)}) \vee ... \vee (p_{(n,1)} \wedge ... \wedge p_{(n,m_n)})$, this outputs a trapdoor $T = \{T_1, ..., T_n\}$. We compute T_i for each $i \in [1, n]$ as follows:

1. Initialize a Bloom filter T_i, that is, initialize a bit array with 0’s.
2. For each term $p_{(i,j)}$ $(j \in [1, m_i])$, add a concatenated string $x \parallel c$ (i.e., a pair of a

[§] We need this restriction because an index and a trapdoor have the similar data structure in our scheme.

[¶] The index for V_b will be similar to one of the indexes given in the Setup phase, but we note that the information about which keyword (a subset) corresponds to which index is not given to $\mathcal{A}$ as mentioned before.

[‖] We assume that a $null$ is a special character.

DOI: 10.1002/sec
Prepared using secauth.cls

position and a character[**]) to the Bloom filter T_i where $p_{(i,j)}$ is a term $w[x] = c$.

SearchIndex(T, Idx) This algorithm is used by the server to search for encrypted documents with matching indexes. Given a trapdoor $T = \{T_1, ..., T_n\}$ and a collection of indexes Idx in the database, the server searches for matching indexes as follows. Now let $T_i \in T$ and $\mathcal{I} \in \mathrm{Idx}$. For each pair of $(T_i, \mathcal{I})$, we do the following:

1. Let $\mathcal{I} = (\mathrm{FID}, \mathcal{I}_{\mathrm{I}}, \mathcal{I}_{\mathrm{II}})$.
2. Let J be the set of positions such that if $T_i[j] = 1, j \in J$.
3. For each $j \in J$, check if $\mathcal{I}_{\mathrm{I}}[j] = 1$. If all the bits corresponding to J are 1's in $\mathcal{I}_{\mathrm{I}}$ [††], the server returns, to the data searcher, the encrypted document corresponding to FID and $\mathcal{I}_{\mathrm{II}}$.

We note that the data searcher can use $\mathcal{I}_{\mathrm{II}}$ to know the exact keyword w even if a false-positive happens because of the Bloom filter's property.

Search Examples

If we do the wildcard search such as "2013/??/??", we can use a search expression $p = ((w[1] = \text{'2'}) \wedge (w[2] = \text{'0'}) \wedge (w[3] = \text{'1'}) \wedge (w[4] = \text{'3'}) \wedge (w[5] = \text{'/'}) \wedge (w[8] = \text{'/'}) \wedge (w[11] = null))$ and generate a trapdoor corresponding to this search expression.

If we want to find a keyword that has a prefix "pre", we can use a search expression $p = ((w[1] = \text{'p'}) \wedge (w[2] = \text{'r'}) \wedge (w[3] = \text{'e'}))$ without specifying a $null$.

If we use a DNF search expression such as ("dog" OR "cat"), we generate two Bloom filters corresponding to "dog" and "cat" respectively as shown in the description of Trapdoor(sk, p), and send the two Bloom filters to the server. The server executes SearchIndex() by using the two Bloom filters respectively to find the documents associated with "dog" or "cat" as shown in the description of SearchIndex(T, Idx).

For example, if Document 1 is associated with keyword "dog" and Document 2 is associated with "cat", we will obtain both Document 1 and Document 2 as the search results of the search expression ("dog" OR "cat").

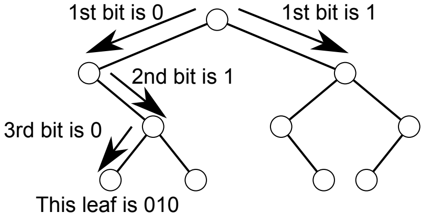

Figure 1. Example of a binary tree of indexes

Index Management on Server Side for Efficient Search

The indexes created by a data owner are sent to the server and the server uses the indexes to search for the encrypted documents with trapdoors. The index management on the server side for executing SearchIndex of our scheme can be implemented with a binary tree search method as well as a linear search method as follows.

Linear search. The first approach is a naive but simple approach, that is, a linear search. Given a Bloom filter T_i in the trapdoor T, the server checks all the indexes exhaustively and the server chooses the matching indexes.

Binary tree search. The second approach is an optimized search, that is, a binary tree search. In this search, we construct a binary tree such that each edge represents a bit (e.g., the left node corresponds to 0 and the right node corresponds to 1) from the Bloom filter $\mathcal{I}_{\mathrm{I}}$ in the index $\mathcal{I}$. We show an example of a binary tree for four Bloom filters $\{010, 011, 101, 110\}$ in Figure 1. Each leaf node has FID and $\mathcal{I}_{\mathrm{II}}$.

When we search this binary tree with a trapdoor 101, we can ignore left nodes of the root node because the first bit must be 1 but their first bits are 0's, thus avoiding the exhaustive search. We need to follow both left and right child nodes when a bit of a trapdoor is 0 (e.g., second bit of 101).

[**] We assume that a $null$ is a special character.

[††] i.e., if the set of 1's included in the Bloom filter T_i is a subset of the set of 1's included in the Bloom filter $\mathcal{I}_{\mathrm{I}}$, we have the keyword match.

4. SECURITY ANALYSIS

4.1. Security Proof

We give the security proof of our scheme based on the security model in Section 2.5.

Theorem 1
Let k be the number of pseudo-random functions for a Bloom filter. Our scheme is $(t, \epsilon, q/k)$-IND-CPSKA secure if a pseudo-random function f for the Bloom filter is a (t, ϵ, q)-pseudo-random function.

Proof
We prove this theorem by basically following the proof for Z-IDX [13] with necessary adaptations.

We prove this theorem with its contrapositive, that is, we assume our scheme is not $(t, \epsilon, q/k)$-IND-CPSKA secure. Then we show that we can construct the algorithm $\mathcal{B}$ that can distinguish between a pseudo-random function and a random function by using $\mathcal{A}$ as a subroutine.

Given an input x, $\mathcal{B}$ can use an oracle $\mathcal{O}$ which outputs $f(x, sk)$ or $g(x)$ to evaluate f or g whenever necessary.

Setup. $\mathcal{B}$ picks a set S of q/k pairs of a position and a character from $\{0,1\}^n$ at random, and sends the set S to $\mathcal{A}$. $\mathcal{A}$ returns a collection S' of polynomially many subsets. For each subset D[‡‡] $\in S'$, $\mathcal{B}$ executes BuildIndex with D and a file identifier FID_D picked at random. $\mathcal{B}$ sends all indexes for S' to $\mathcal{A}$.

Query. $\mathcal{B}$ executes Trapdoor for a keyword (or a search expression) x if $\mathcal{A}$ queries a trapdoor and outputs a trapdoor T_x for the keyword (or the search expression) x. Here we can assume a search expression x does not include $\vee$ (i.e., a logical OR) because $\mathcal{A}$ can obtain trapdoors for $x_1 \vee x_2$ by obtaining a trapdoor for x_1 and a trapdoor x_2 separately.

Challenge. $\mathcal{A}$ picks nonempty subsets V_0 and V_1 from S' such that $|V_0 - V_1| \neq 0$, $|V_1 - V_0| \neq 0$ and $|V_0| = |V_1|$. Also V_0, V_1 must satisfy that no trapdoor for a set K of pairs of a position and a character was already queried by $\mathcal{A}$ where $|K \cap (V_0 \triangle V_1)| > 0$. Also V_0, V_1 must satisfy that no index for a set K is given to $\mathcal{A}$ where $|K \cap (V_0 \triangle V_1)| > 0$.

$\mathcal{A}$ sends V_0 and V_1 to $\mathcal{B}$. $\mathcal{B}$ picks $b \in \{0,1\}$ and a file identifier FID_b at random, executes BuildIndex, and sends the index to $\mathcal{A}$. After $\mathcal{A}$ receives the index for V_b, $\mathcal{A}$ cannot query the trapdoors that do not follow the restriction mentioned above.

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

$\mathcal{B}$ takes at most t time because $\mathcal{A}$ takes at most t time. $\mathcal{B}$ sends at most q queries to $\mathcal{O}$ because there exist only q/k pairs of a position and a characters in S, $\mathcal{A}$ sends at most q/k queries, and $\mathcal{B}$ sends k queries per $\mathcal{A}$'s single query.

The following lemmas show that $\mathcal{B}$ has an advantage greater than ϵ to determine if the oracle $\mathcal{O}$ corresponds to a pseudo-random function f or a random function g.

However, this contradicts the definition of pseudo-random function.

Therefore, the theorem is proven. □

Lemma 1
$|\Pr[\mathcal{B}^{f(\cdot,sk)} = 0 | sk \xleftarrow{R} \{0,1\}^\lambda] - \frac{1}{2}|$ is non-negligible if f is a pseudo-random function.

Proof
$\mathcal{B}$ simulates $\mathcal{C}$ in an IND-CPSKA game perfectly and we assume that $\mathcal{A}$ can break our scheme (i.e., our scheme is not $(t, \epsilon, q/k)$-IND-CPSKA secure). Therefore, the proof of this lemma is immediate. □

Lemma 2
$\Pr[\mathcal{B}^g = 0 | g \xleftarrow{R} \{F : \{0,1\}^n \to \{0,1\}^m\}] = \frac{1}{2}$ if g is a random function.

Proof
All we have to consider are Challenge subsets V_0 and V_1 since other subsets in S' do not leak any information about the Challenge subsets.

Without loss of generality, assume that $V_0 \triangle V_1$ has two pairs x, y of a position and a character such that $x \in V_0$ and $y \in V_1$, and $\mathcal{A}$ guesses b with an advantage σ. That is, given an output $g(z)$, $\mathcal{A}$ distinguishes $z = x$ from $z = y$. If g is a random function chosen at random, σ must be 0. Therefore, we proved this lemma. □

[‡‡] Usually this corresponds to one keyword.

4.2. Limitation

Our scheme has a limitation as the existing schemes.

Our scheme divides a trapdoor into clauses. For example, when we perform a search with a search expression (“dog” OR “cat”) by using a DNF formula, the server obtains two trapdoors from the data searcher. Though the sever cannot obtain the plaintexts of the keywords “dog” and “cat” from the trapdoors, the server can know the distinct search results for “dog” and “cat” respectively.

Not only our scheme but also many of the existing schemes have this limitation (e.g., Goh’s scheme [13], Li et al.’s scheme [11]).

5. PERFORMANCE EVALUATION

Given a bit length m of a Bloom filter, an index consists of FID, an m-bit Bloom filter ($\mathcal{I}_I$) and $\mathcal{I}_{II}$ (which depends on the symmetric encryption scheme). Assume that the search expression can be divided by disjunctions into n_ℓ terms. The size of the trapdoor is $n_\ell m$ bits because the trapdoor consists of n_ℓ (m-bit) Bloom filters. Given the total number of characters in all keywords ℓ_m and the number of pseudo-random functions for the Bloom filer k, the execution time of BuildIndex is $\mathcal{O}(\ell_m)$ since we must compute $\ell_m \cdot k$ (k is constant) pseudo-random functions.

5.1. Implementation as a Native Application

One approach to implement a searchable encryption scheme is to implement it as a native application. We implement our scheme with an equality search on an Intel Core i7 2600K CPU. We used a 256-bit Bloom filter and symmetric key encryption AES [19] and keyed hash function HMAC-SHA256 [16, 17] as pseudo-random functions. Given a secret key sk and an input x, let HMAC-SHA256 be $f(sk, x)$. We can use HMAC-SHA256 as distinct pseudo-random functions $f_i(sk, x) = f(sk, i \,\|\, x)$.

We show the result of this experiment in Figures 2 and 3.

Usually the keyword length will be relatively small (we assume it will be about 10 bytes here for example). The run times of BuildIndex and Trapdoor are less than 1 millisecond for 10-byte keywords. Even if we have a 1000-byte keyword, we need only about 10 milliseconds. Therefore, we believe our scheme is practical in this environment.

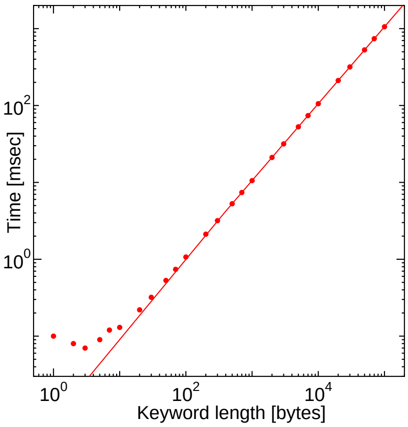

Figure 2. Run time of BuildIndex as a native application.

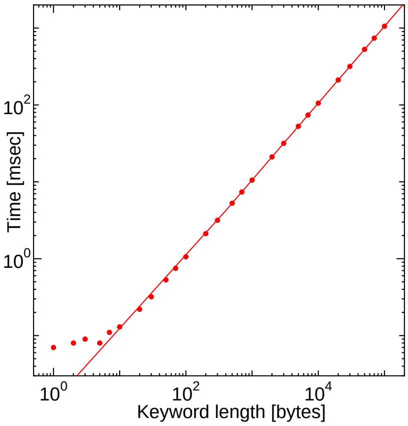

Figure 3. Run time of Trapdoor as a native application.

5.2. Implementation as a Web Application.

Another approach is to implement it as a web application. We also implemented BuildIndex and Trapdoor with JavaScript, and had an experiment in a browser Google Chrome.

We show the result of this experiment in Figures 4 and 5 on the same environment as in Section 5.1.

In these figures, we see these run times are relatively slower than the results for a native application. However, when we have 10-byte keyword, we need only about 2 milliseconds.

We also show the result of this experiment on a tablet (Google Nexus 7: 1.3 GHz NVIDIA Tegra 3 quad core

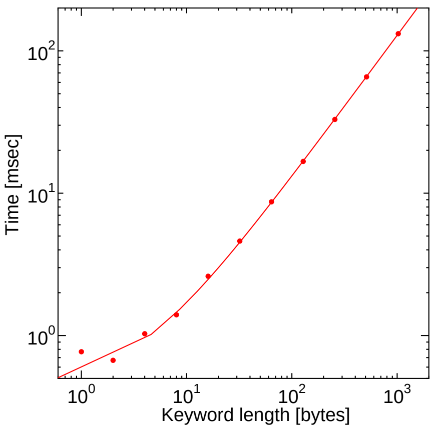

Figure 4. Run time of BuildIndex as a web application.

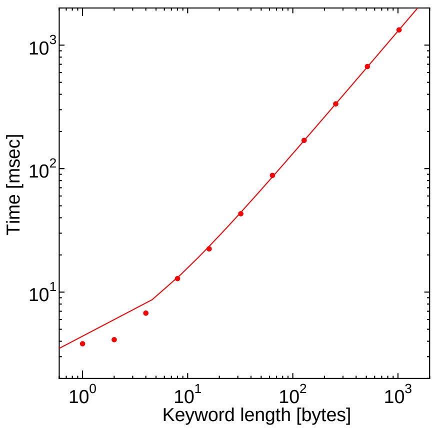

Figure 6. Run time of BuildIndex on a tablet.

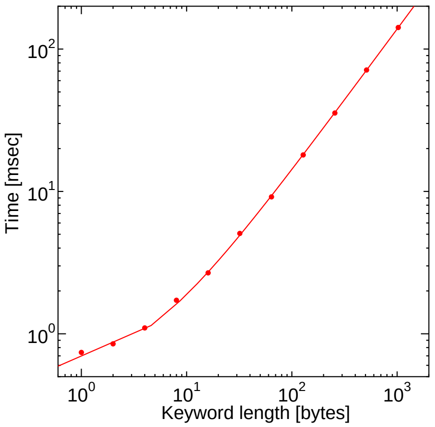

Figure 5. Run time of Trapdoor as a web application.

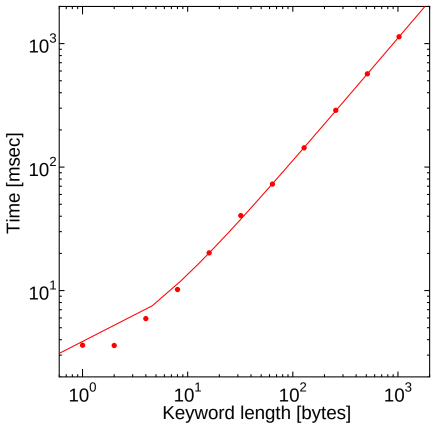

Figure 7. Run time of Trapdoor on a tablet.

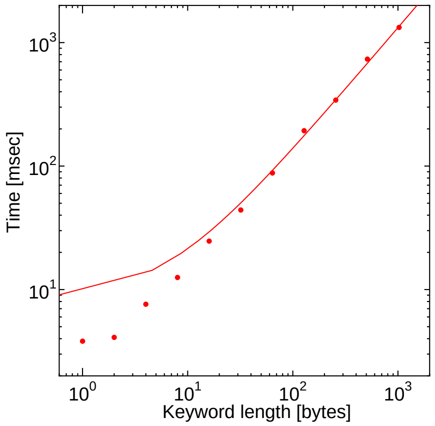

Figure 8. Run time of BuildIndex on a smartphone.

CPU) in Figures 6 and 7, and on a smartphone (SHARP AQUOS PHONE ZETA: 1.5 GHz Qualcomm Snapdragon S4 MSM8960 dual core CPU) in Figures 8 and 9.

We see these run times are further slower than the results for a web application with a PC. However, when we have 10-byte keyword, we need only about 20 milliseconds with a tablet and about 30 milliseconds with a smartphone. Therefore, we believe our scheme can be deployed on a tablet and even a smartphone.

6. CONCLUSION

In this work, we presented a searchable symmetric encryption scheme. Our scheme supports not only an equality search but also other types of searches like a wildcard search based on comparisons per character. We implemented our scheme as a native application and as a web application. We confirmed our scheme can be deployed not only as a native application but also as a web

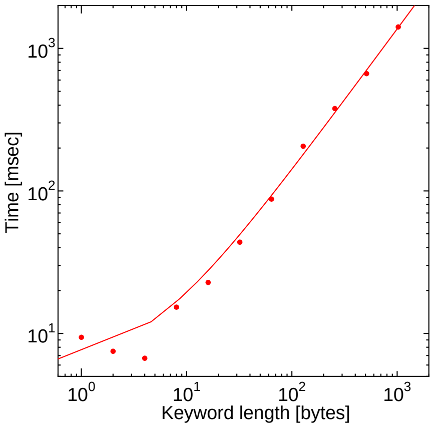

Figure 9. Run time of Trapdoor on a smartphone.

application. Furthermore, we confirmed our scheme can be used even on a tablet and a smartphone.

In our experiment as a web application, we assumed the server does not alter the script file to avoid the encryption. However, the script file might be altered by the malicious cloud provider or an intruder. Therefore, we need some mechanisms to make sure that the script file is not altered and our data will be encrypted. To implement our scheme as a browser add-on might be a countermeasure against this attack. Designing and implementing this mechanism are our future work.

ACKNOWLEDGEMENTS

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 Kurata Grant from The Kurata Memorial Hitachi Science and Technology Foundation.

REFERENCES

1. Song DX, Wagner D, Perrig A. Practical techniques for searches on encrypted data. *IEEE Symposium on Security and Privacy 2000*, 2000; 44–55, doi: 10.1109/SECPRI.2000.848445.
2. Bao F, Deng RH, Ding X, Yang Y. Private query on encrypted data in multi-user settings. *Proceedings of the 4th international conference on Information security practice and experience*, ISPEC'08, Springer-Verlag: Berlin, Heidelberg, 2008; 71–85, doi:10.1007/978-3-540-79104-1_6.
3. Chang YC, Mitzenmacher M. Privacy preserving keyword searches on remote encrypted data. *Applied Cryptography and Network Security*, *Lecture Notes in Computer Science*, vol. 3531, Ioannidis J, Keromytis A, Yung M (eds.). Springer-Verlag: Berlin, Heidelberg, 2005; 391–421.
4. Curtmola R, Garay J, Kamara S, Ostrovsky R. Searchable symmetric encryption: improved definitions and efficient constructions. *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, ACM: New York, NY, USA, 2006; 79–88, doi:10.1145/1180405.1180417.
5. Waters B, Balfanz D, Durfee G, Smetters DK. Building an encrypted and searchable audit log. *In The 11th Annual Network and Distributed System Security Symposium*, 2004.
6. Kamara S, Papamanthou C, Roeder T. Dynamic searchable symmetric encryption. *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, ACM: New York, NY, USA, 2012; 965–976, doi:10.1145/2382196.2382298.
7. Kurosawa K, Ohtaki Y. UC-secure searchable symmetric encryption. *Financial Cryptography and Data Security*, *Lecture Notes in Computer Science*, vol. 7397. Springer: Berlin, Heidelberg, 2012; 285–298, doi:10.1007/978-3-642-32946-3_21.
8. Boneh D, Di Crescenzo G, Ostrovsky R, Persiano G. Public key encryption with keyword search. *Advances in Cryptology - EUROCRYPT 2004*, *Lecture Notes in Computer Science*, vol. 3027, Cachin C, Camenisch J (eds.). Springer: Berlin, Heidelberg, 2004; 506–522, doi:10.1007/978-3-540-24676-3_30.
9. 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* 2008; **21**:350–391, doi:10.1007/11535218_13.

10. Boneh D, Waters B. Conjunctive, subset, and range queries on encrypted data. *Theory of Cryptography*, *Lecture Notes in Computer Science*, vol. 4392, Vadhan S (ed.). Springer Berlin / Heidelberg, 2007; 535–554.
11. Li J, Wang Q, Wang C, Cao N, Ren K, Lou W. Fuzzy keyword search over encrypted data in cloud computing. *INFOCOM, 2010 Proceedings IEEE*, 2010; 1 –5, doi:10.1109/INFCOM.2010.5462196.
12. Sedghi S, van Liesdonk P, Nikova S, Hartel P, Jonker W. Searching keywords with wildcards on encrypted data. *Security and Cryptography for Networks*, *Lecture Notes in Computer Science*, vol. 6280, Garay J, De Prisco R (eds.). Springer: Berlin, Heidelberg, 2010; 138–153, doi:10.1007/978-3-642-15317-4_10.
13. Goh EJ. Secure indexes. Cryptology ePrint Archive, Report 2003/216 2003. `http://eprint.iacr.org/2003/216/`.
14. Watanabe C, Arai Y. Privacy-preserving queries for a DAS model using encrypted bloom filter. *Database Systems for Advanced Applications*, Springer, 2009; 491–495, doi:10.1007/978-3-642-00887-0_43.
15. Suga T, Nishide T, Sakurai K. Secure keyword search using bloom filter with specified character positions. *Provable Security*, *Lecture Notes in Computer Science*, vol. 7496, Takagi T, Wang G, Qin Z, Jiang S, Yu Y (eds.). Springer: Berlin, Heidelberg, 2012; 235–252, doi:10.1007/978-3-642-33272-2_15.
16. Krawczyk H, Bellare M, Canetti R. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational) February 1997.
17. NIST. Announcing the secure hash standard. Federal Information Processing Standards Publication 180-2 2002.
18. Bloom BH. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* July 1970; **13**:422–426, doi:362686.362692.
19. NIST. Announcing the advanced encryption standards (AES). Federal Information Processing Standards Publication 197 2001.