Fine-grained Dynamic Searchable Symmetric Encryption for Conjunction Query

宮野, 資基 / Miyano, Motoki

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
15
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2020-03-24
(URL)
https://doi.org/10.15002/00022733

積集合検索が可能な柔軟な動的検索可能暗号
Fine-grained Dynamic Searchable Symmetric Encryption for Conjunction Query

宮野　資基 *
Motoki Miyano
法政大学大学院　情報科学研究科　情報科学専攻
Email: 18t0015@cis.k.hosei.ac.jp

***Abstract*—Searchable Symmetric Encryption (SSE) has been researched from three perspectives: security, performance, and functionality, and various SSEs have been proposed. Some SSEs are constructed with a data structure called Bloom Filter (BF). However, while BF has a good storage cost, it cannot efficiently perform data update operating and has a problem in functionality. With the spread of the cloud in recent years, it is desirable to have many functions. In this research, we propose two SSEs that have update function employing Counting Bloom Filter (CBF). The first, proposes a single keyword search SSE called D-IDX that can be updated flexibly. The second, improves D-IDX and proposes SSE called Dynamic Cross-Tags（DXT） that can efficiently perform conjunction keyword searches in addition to update operating. Both SSEs use CBF to enable efficient updating of keywords without significantly changing the data stored on the server. Although both SSEs have the same security, DXT can construct a more secure SSE by employing homomorphic encryption. We have implemented proposed SSEs. It shows that it is efficient and practical in update operation and multiple keywords search.**

1. introduction

With the recent development of cloud computing, it has become common to outsource data. However, outsouring data often causes privacy issue, in particular when the data is stored in a server that is not necessarily trusted. A simple solution of such a privacy issue is to encrypt the data before storing it in the server. However, availability of the data is unacceptably reduced when we simply encrypt all the data stored in the server since we cannot perform any operation over encrypted data. For example, the user cannot even search for emails containing certain keyword when emails are encrypted.

Searchable Symmetric Encryption (SSE) is a technology aiming at solving these problems. SSE is a cryptographic primitive that makes it possible to securely retrieve encrypted data associated with certain keyword without decrypting the encrypted data. SSE maintains the data structure called an index where index records the correspondence between documents and keywords included therein. When we want to search documents containing certain keyword, we compute encrypted keyword called a trapdoor and send the trapdoor to the server, which enables the server to search over encrypted documents with the help of index. All SSEs possesses a trade-off between security, performance, and functionality. If SSE is designed in a way that it leaks no information at all, performance and functionality will be reduced, and if SSE is designed to be computationally efficient and rich functionality comparable to an unencrypted database, information leaked via search operation cannot be small.

Since SSE was first proposed[10], computationally efficient schemes with respect to the size of data[13][15][6], and schemes that support document update as well as search have been proposed[11][16] so far. Some of these SSEs [6][1] have been designed with a data structure called Bloom Filter (BF)[7]. However, since BF is not designed to support delete operation, it is not possible to construct BF based SSE that can efficiently update. To resoleve the problem, SSE supporting update operation based on the Counting Bloom Filter (CBF)[8] has been presented where CBF is a variant of BF, to efficiently update the data[12]. However, [12] does not perform the update operating on the server, and does not fully utilize the advantages of cloud computing.

In recent years, high-performance SSE has been attracting attention while maintaining security in order to construct a practical one. Oblivious Cross-Tags (OXT)[2] and Hidden Cross-Tags (HXT)[1] are such state-of-the-art SSEs. OXT is not only a single keyword search but also a protocol that can efficiently conjunction search a product set using multiple keywords. HXT, like OXT, allows multiple keywords, and uses Bloom Filter and lightweight cryptography to create a protocol with small information leakage than OXT. However, these SSEs can only search keywords and cannot efficiently perform update operation.

In this paper, we propose two SSEs called D-IDX and Dynamic Cross-Tags(DXT) that realize efficient update by employing CBF as an index. By generating an index for each document, D-IDX can flexibly update not only the entire document but also words contained in the document. Also, compared to SSE employing CBF[12], proposed SSEs are more suitable for cloud computing since the update process is performed on the server. DXT improves the index of D-IDX by adding a new index which make it possible to efficiently search the conjunction result of multiple keywords. DXT also support efficient update, and possesses richer functionality such as ‘updating + multiple keywords search’. Both SSEs achieved the simulation-based, Non-adaptive security defined by Curtmola[4]. In addition, DXT can be improved to SSE which achieves adaptive security by encrypting the value of CBF using homomorphic encryption. Finally, we implement both SSEs and show the results. In this abstract, DXT is mainly explained, and D-IDX and detailed explanation are described in full paper or [5].

* Supervisor: Prof.Satoshi Obana

2. Preliminaries

2.1. Definition of SSE(Update.ver)

SSE considers a model that consists of a "user" who is a data holder and a searcher, and a "server" that holds data and outputs search results based on the search data, so that the data and the search data do not leak to the server suppose that. The user sends the trapdoor T to the server as a query, and generates an index I that is encrypted data so that the search can be performed. When the server receives the trapdoor, it searches the encrypted data and returns the corresponding data.

If the data to be handled is expressed as n documents as $DB = (db_1, ..., db_n)$, and each db_i is composed of words$(w_1, ..., w_m)$. Let id_{db_i} be the document identifier. $DB(w) = (id_{db_j}, ..., id_{db_k})$ is a set of identifiers of documents including the keyword w.

Since trapdoors are deterministic in symmetric key based searchable encryption, the same trapdoor is generated for the same keyword. Therefore, basically, information on the search pattern of the user is leaked. Further, by performing a search a plurality of times, information of a result pattern indicating the relationship between the results of each query is also leaked. In addition, when building an SSE such as the size of a DB or trapdoor, information that cannot be suppressed from leaking is defined as "unavoidable information leakage".

In dynamic SSE, consider adding or deleting documents or keywords. In both operations, processing is performed by generating a trapdoor according to the update. When adding the document db or the keyword w, given the additional trapdoor T^+ and the index I, outputs the updated I'. Similarly, when deleting the document db or the keyword w, given the additional trapdoor T^- and the index I, outputs the updated I'.

Definition 1. Dynamic SSE is a protocol consisting of the following algorithm SSE = (KeyGen, BuildIndex, SearchTrapdoor, AddTrapdoor, DelTrapdoor, Search, Add, Del) :

1) $K \leftarrow KeyGen(1^s)$
   Algorism for generate the secret key. Given a security parameter s, outputs the secret key K
2) $I \leftarrow BuildIndex(D, K)$
   Algorism for construct the index. Given a document D and the secret key K, outputs the index I.
3) $T(w) \leftarrow SearchTrapdoor(w, K)$
   Algorithm for calculate search data (trap door). Given the secret key K and word w, outputs the trapdoor $T(w)$ for w.
4) $T^+ \leftarrow AddTrapdoor(\{db, w\}, K)$
   Algorithm for calculate add data (trap door). Given the secret key K and word w, outputs the trapdoor T^+ for w.
5) $T^- \leftarrow DelTrapdoor(\{db, w\}, K)$
   Algorithm for calculate delete data (trap door). Given the secret key K and word w, outputs the trapdoor T^- for w.
6) $S(w) \leftarrow Search(T(w), I)$
   Algorism for search word. Given the trapdoor $T(w)$ for word w and the index I for document D, outputs search result $S(w)$
7) $I' \leftarrow Add(I, T^+)$Algorithm for add data (trap door). Given the index I and the trapdoor T^+, outputs the updated index I'
8) $I' \leftarrow Del(I, T^-)$Algorithm for delete data (trap door). Given the index I and the trapdoor T^-, outputs the updated index I'

2.2. Counting Bloom Filter

A Bloom filter is a probabilistic data structure[7], conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. As described later, A Bloom filter has a possibility of false positive, but since the set is represented by a bit string, it is a space efficient data structure. This data structure is defined by a bit array of m bits, k different hash functions and n elements. An empty Bloom filter is a bit array of m bits, all set to 0. To add an element, input it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1. To test whether it is in the set, input it to each of the k hash functions to get k array positions. If any of the bits at these positions is 0, the element is not in the set. In contrast if all are 1, the element is in the set.

However, there is possibility that all of the bit at there position are 1 during the addition of other elements, even though the test element is definitely not in the set, it is judged that it is in the set. This is false positive. On the other hand, if any of the bits at these positions is 0, the element is not in the set. Since the Bloom filter consists of a bit array, elements can not be deleted. Since each element is inserted in an array with k bits, can be deleted if any bit set to 0. But the bit set to 0 can be represented as 1 in other elements, resulting in a other elements are deleted, so false negatives will occur.

Therefore, CBF was proposed in which no false negatives occurred. Counting Bloom filter [8] is a type of Bloom filter and is a data structure that has ability of deleting elements. It differs greatly from Bloom filter is the structure of the array. In the Bloom filter, each element of the array is only 1 bit, but in the counting Bloom filter, each element of the array is expanded to n bits. The insert operation is extended to increment the value of the array positions. The delete operation is also extended to decrement the value of the array positions. the search(test whether it is in the set) operation checks that each of the array positions is non-zero. If all are more than 1, the element is in the set.

3. Construction

We now give the main construction of DXT protocol and then prove its performance and analyze its security

3.1. Protocol

DXT uses (for security parameter λ) (i) a cyclic group $\mathbb{G}$ with prime order p and generator g, for which the DDH assumption holds(ii) a symmetric key encryption scheme Sym with key space $\{0,1\}^\lambda$(iii) a Counting Bloom filter

CBF with length m and k hash functions (iv) PRFs F with range $\{0,1\}^\lambda$ and F_p with range Z_p^*

In DXT, two data sets are used, one is an improved version of the OXT[3] data set, and the other generates a data set employing CBF. DXT can be divided into three parts: setup, search, and update. In this paper, explanation of $Keygen$ algorithm in the definition of SSE is omitted. In the SSE definition, the setup part corresponds to $BuildIndex$, the search part corresponds to $SearchTrapdoor$, $Search$, and the update part corresponds to $AddTrapdoor$, $DelTrapdoor$, Add, Del.

Algorithm 1 Setup

Input: $mks, param.DB$
Output: EDB

function $Setup(mks, param.DB)$
  $mks : (K_s\ for\ PRF\ F\ and\ K_I, K_Z, K_X\ for\ F_p)$
  $param : (H\ for\ hash\ function)$
  $Initialize\ T$
  $Initialize\ CBF \leftarrow 0^m$
  for $w \in W$ **do**
    $K_p || K_e \leftarrow F(K_s, w)$
    $set\ counter\ c \leftarrow 1$
    for $id \in DB(w)$ **do**
      $xid \leftarrow F_p(K_I, id), z_w \leftarrow F_p(K_Z, w||c)$
      $y_c \leftarrow xid \cdot z_w^- 1$
      $l \leftarrow F(K_p, c)$
      $e_c \leftarrow Sym.Enc(K_e, id)$
      $add\ (l, e_c, y_c)\ to\ T$
      $c++$
    end for
  end for
  $DTIDX \leftarrow create(T)$
  for $w \in W$ **do**
    for $id \in DB(w)$ **do**
      $xid \leftarrow F_p(K_I, id)$
      for $j = 1 : k$ **do** // k *is a number of* H
        $h_j \leftarrow H_j(g^{F_p(K_X, w) \cdot xid})$
        $CBF[h_j]++$
      end for
    end for
  end for
  $CFIDX \leftarrow CBF$
  $EDB = (DTIDX, CFIDX)$
  return EDB
end function

The setup is a process of outputting an encrypted data set ($EDB = \{DTIDX, CFIDX\}$) with a secret key,hash function and stored(DB) as inputs. $DTIDX$ is a data set that can identify document IDs according to keywords by storing values $y = F_p(K_I, id) \cdot F_p(K_Z, w||c)^{-1}$ and $l = F(K_p, c)$ depending on document IDs and keywords. Similarly, $CFIDX$ stores values $g^{F_p(K_X,w)\cdot F_p(K_I,id)}$ depending on the document ID and the keyword, but stores values depending on the input to the hash function $H_{j(1\le j\le k)}(g^{F_p(K_X,w)\cdot F_p(K_I,id)})$ and $+1$ of the mapped CBF.

The search part receives multiple keywords and outputs a document ID including all the keywords. Let search query be $q = (w_s; w_2, ..., w_m)$. w_s is set to a keyword having the lowest appearance frequency than other keywords. First, w_s is subjected to a single keyword using

Algorithm 2 Search

Input: $mks, param, W = (w_1, ..., w_n)$
Output: $Result\ R$

function $Search(mks, param, W)$
  $mks : (K_s\ for\ PRF\ F\ and\ K_Z, K_X\ for\ F_p)$
  $param : (H\ for\ hash\ function)$
  $K_p \leftarrow F(K_s, w_1)$
  $send\ K_p\ to\ the\ server$
  $Initialize\ t \leftarrow \{\}$
  for $c = 1\ until\ Get\ returns\ \perp$ **do**
    $l \leftarrow F(K_p, c)$
    $y_c \leftarrow Get(DTIDX, l)$
    $Add\ y_c\ to\ t$
  end for
  $return\ t\ to\ user$
  for $c = 1 : |t|$ **do**
    $Initialize\ CBF_c \leftarrow 0^m$
    $recovers\ y_c\ from\ t$
    $n_{w_1} \leftarrow F_p(K_Z, w_1||c)$
    for $l = 2 : n$ **do**
      $xtoken \leftarrow g^{n_{w_1} \cdot F_p(K_X, w_l)}$
      $xtag = xtoken^{y_c}$
      for $j = 1 : k$ **do**
        $u_j \leftarrow H_j(xtag)$
        $CBF_c[u_j]++$
      end for
    end for
  end for
  $send\ (CBF_1, ..., CBF_t)\ to\ the\ server$
  $Initialize\ R \leftarrow \{\}$
  for $c = 1 : |t|$ **do**
    $recovers\ e_c\ from\ DTIDX's\ result$
    for $i = 0 : m$ **do**
      $V_c = CFIDX[m] - CBF_c[m]$
    end for
    if $V_c\ has\ no\ negative\ value$ **then**
      $add\ e_c\ to\ R$
    end if
  end for
  $return\ R$
end function

$DTIDX$. The value of $l = F(K_p, c)$ stored in $DTIDX$ is searched as a trapdoor, and the value of y and the number of search result are returned to the user. The user generates a CBF corresponding to $q = (w_s; w_2, ..., w_m)$ based on the received data, and performs a search using $CFIDX$. $H_{j(1\le j\le k)}(g^{F_p(K_Z,w_1||c)\cdot F_p(K_X,w_l)\cdot y})(l : 2, ..., m)$ is stored in CBF. The value stored in $CFIDX$ is $H_{j(1\le j\le k)}(g^{F_p(K_X,w)\cdot F_p(K_I,id)})$. Therefore, if the value of y is $F_p(K_I, id) \cdot F_p(K_Z, w||c)^{-1}$, it is $g^{F_p(K_Z,w_1||c)\cdot F_p(K_X,w_l)\cdot y} = g^{F_p(K_X,w)\cdot F_p(K_I,id)}$ and matches the value of $CFIDX$, otherwise it becomes a random value. Since y indicates an ID including w_s, if the values of CBF and CFIDX match, it can be said that the ID is included in the entire q.This makes it possible to search for conjunction.

In the update part, the keyword to be updated is input, and update operation is performed on the stored EDB. In order to avoid new information leakage, using a new key without the key which used in the first generated data set.

Algorithm 3 Update

Input: $mks, w, DTIDX, CFIDX, S, OP$
Output: $DTIDX', CFIDX', S'$

function $Update(mks, w, DTIDX', CFIDX', S)$
  $mks : (K_s^+, K_s^- \ for \ F \ and \ K_I^+, K_Z^+, K_X^+ \ for \ F_p)$
  $recovers \ (DTIDX, CFIDX) \ from \ EDB$
  $S \leftarrow \{\}$
  if $OP = Add$ **then**
    $K_p^+ || K_e^+ \leftarrow F(K_s^+, w)$
    $K_p^- \leftarrow F(K_s^-, w))$
    $c \leftarrow Get(D_{count}, w)$
    if $c = \perp$ **then**
      $c \leftarrow 0$
    end if
    $xid \leftarrow F_p(K_I^+, id), z_w \leftarrow F_p(K_Z^+, w||c)$
    $y \leftarrow xid \cdot z_w^- 1$
    $l \leftarrow F(K_p^+, c)$
    $e \leftarrow Sym.Enc(K_e^+, id)$
    $rm \leftarrow F(K_p^-, id)$
    $send \ (l, e, y, rm) \ to \ server$
    if $rm \in S$ **then**
      $remove \ rm \ from \ S$
    end if
    $add \ (l, e, y) \ to \ DTIDX$
  else if $OP = Delete$ **then**
    $K_p^- \leftarrow F(K_s^-, w)$
    $rm \leftarrow F(K_p^-, id)$
    $send \ rm \ to \ the \ server$
    $add \ rm \ to \ S$
  end if
  $DTIDX', S' \leftarrow DTIDX, S$
  $xid \leftarrow F_p(K_I^+, id)$
  $up \leftarrow g^{F_p(K_X^+, w) \cdot xid}$
  $send \ up \ to \ the \ server$
  for $j = 1 : k$ **do**
    $u_j \leftarrow H_j(up)$
    if $OP = Add$ **then**
      $CFIDX[u_j] + +$
    end if
    if $OP = Delete$ **then**
      $CFIDX[u_j] - -$
    end if
  end for
  $CFIDX' \leftarrow CFIDX$
  $return \ DTIDX', CFIDX', S'$
end function

When adding keywords, create a trapdoor for each and add it to the EDB. In the case of deletion, a data set S for deletion is newly generated, and a keyword is stored in S. The deletion function is realized by checking S and determining whether or not to search when searching. It is necessary to change the search parts by adding this update function, but the detailed algorithms is described in full paper.

4. Security

In this section, we proof DXT security based on the simulation-based, Non-adaptive security defined by Curtmola[4]. Show the attacker the actual protocol $Real$ and the protocol $Idel$ built by Simulator $\mathcal{S}$. $\mathcal{S}$ is constructed based on "unavoidable information leakage" by $Real$ protocol. Currently, if the attacker cannot identify $Real$ or $Idel$, it is determined that the actual protocol does not leak more information than "unavoidable information leakage", and is defined as secure.

In DXT ($Real$), the value is calculated based on the key known only to the user, but the index and trapdoor are actually stored based on the pseudo random function and the DDH assumption. Therefore, if $\mathcal{S}$ generates a random value without the key and generates an index or trapdoor according to the ID or keyword using random value, it cannot be distinguished from $Real$ under a pseudo-random function or DDH assumption. Therefore, DXT achieves Non-adaptive security.

This simulation base has Adaptive security in addition to Non-adaptive security, and assumes a stronger attacker. DXT can achieve adaptive security by employing homomorphic encryption. In Adaptive security, it cannot be achieved if the value of CBF is disclosed to the server. Therefore, by encrypting CBF with homomorphic encryption, achieving security without impairing the conventional search and update functions. Detailed security proofs and construction algorithms using homomorphic encryption are described in full paper.

5. Performance

In this section, we consider computational cost, storage cost, and information leakage while comparing with other protocols.

5.1. Computational cost

In this section, since there are two data sets, we consider time cost of $DTIDX$ and $CFIDX$ separately. $DTIDX$ increases search time when the amount of data stored is large. The stored data increases in the order of the keywords and the number of documents including the keywords $DB(w)$ as can be seen from the 6 and 9 lines of the Setup algorithm. Since keywords increase basically with the number of DBs, it can be said that they increase with the order of DBs. In addition, the search increases with the order $O(DB(w))$ of the number of DBs including the search keywords. For example, keywords included in many DBs such as "THE" take a long time to search, and rare keywords such as "Homomorphic" do not take a long time to search. Since [6],[12] wants to search all documents for one keyword, it is DB's order $O(DB)$. Obviously $DB(w)$ does not exceed the number of DBs, so DXT can be said to be more efficient than [6],[12] in $DTIDX$ single keyword search.

$CFIDX$ clearly becomes the order $O(w)$ of the number of keywords to be searched. The search time also changes depending on the number of results output by $DTIDX$. When the output result of $DTIDX$ is $S(w)$, the exact order of $CFIDX$ is $O(S(w_s) \times m - 1)$ for $q = (w_s; w_2, ..., w_m)$. Therefore, w_s can be efficiently searched by selecting the query with the lowest appearance frequency among the queries. However, since the CBF of $CFIDX$ stores all data as one array, it does not depend on the number of DBs. Since [6],[12] depends on the number of search keywords and the number of DBs,

it can be said that DXT is more efficient than [6],[12] in conjunction search.

5.2. Storage cost

For $DTIDX$, Storage cost is basically the order $O(DB)$ of the number of DBs. $CFIDX$ is similarly $O(DB)$, but the amount of data for one keyword is very small. In OXT, since values depending on the document ID and keyword as corresponding to $CFIDX$, the data amount depends on the prime number p. Considering practicality, p is preferably 1024 bits or more, and therefore requires a data amount of 1024 bits per keyword. However, in $CFIDX$, the amount of data per keyword can be suppressed to the number of hash functions bit by inputting values depending on the document ID or keywords. It is said that the number of hash functions used to generate $CFIDX$ is about 7∼12, so it is greatly reduced from OXT. Compared with BF, according to section 2.2, increased from 1bit to $n$bit, but since n is at most $2 \sim 4$ from [14], it is not a large increase as a whole.

5.3. Information leakage

Since a search is performed using multiple keywords, not only the information of the search result but also the information expressed in the relationship between the keywords is leaked. Consider the following document group:

id	keywords	id	keywords
1	$w_2, w_3, w_4,$	4	w_4, w_5
2	w_1, w_2	5	w_3, w_5
3	w_1, w_2, w_3	6	w_1, w_3

TABLE 1. LEAKAGE COMPARISON FOR QUERY $w_1 \wedge w_2 \wedge w_3$ BETWEEN Z-IDX, OXT, AND DXT

Protocol	Leaked elements
Z-IDX[6]	$\{(id_1, w_2), (id_2, w_2), (id_3, w_2)$ $(id_1, w_3), (id_3, w_3), (id_5, w_3), (id_6, w_3)\}$
OXT[2]	$\{(id_2, w_2), (id_3, w_2), (id_3, w_3), (id_6, w_3)\}$
DXT	$\{(id_3, w_2), (id_3, w_3)\}$

Consider searching for the conjunction of $w_1 \wedge w_2 \wedge w_3$. Let w_1 be less frequently searched than other search keywords. The document ID's containing each of the queried words are $DB(w_1) = \{id_2, id_3, id_6\}, DB(w_2) = \{id_1, id_2, id_3\}, DB(w_3) = \{id_1, id_3, id_5, id_6\}$.

If the search is simply performed in the order of w_1 to w_3, the leaked IDs are $\cup_{j=1}^{3}(DB(w_j)) = \{id_2, id_3, id_6\} \cup \{id_1, id_2, id_3\} \cup \{id_1, id_3, id_5, id_6\}$, and the information of all the searched keywords is leaked as a result pattern. Therefore, OXT searches for w_1 first and keeps taking the intersection with the result, thereby reducing information leakage. In the case of $w_1 \wedge w_2 \wedge w_3$, information will be leaked only for $\cup_{j=2}^{3}(DB(w_1) \cap DB(w_j)) = \{id_2, id_3\} \cup \{id_3, id_6\}$. However, even with this protocol, information of IDs other than id_3 that is the result of $w_1 \wedge w_2 \wedge w_3$ is also leaked.

In DXT, similarly to OXT, w_1 is searched first, but w_2 and w_3 are combined into one of the data, and the intersection with the result of w_1 is obtained. Therefore, the leaked IDs are $\cap_{j=2}^{3} DB(w_j) = \{id_3\}$ and the leaked information is smaller than OXT. Table 1 shows that for Z-IDX and OXT where the number of information leaks is 7 and 4 elements, DXT can be reduced to only 2 elements of information leaks.

However, if an attacker performs a malicious search procedure, information similar to OXT may be leaked. For example, after $w_1 \wedge w_2 \wedge w_3$ are searched, $w_1 \wedge w_2$ are searched, and a difference between values that can be obtained in the middle is obtained, so that the search results of $w_1 \wedge w_3$ can be obtained, and the result $\{id_2, id_3\} \cup \{id_3, id_6\}$ leaks. Therefore, this information leakage is avoided by sending the query at random. This is because even if a difference is obtained by generating a query at random, it is not known what information it has. The random send increases the cost on the user side, but it is considered that the increase amount is small because only the order of transmission is stored.

6. Evaluations

In this section, we compare and evaluate the implementation of DXT and other protocols.

6.1. Implementation

In this implementation, we compare search time with [12]. There are multiple values to be changed for implementation, such as the number of documents, the number of search keywords, and the number of single keyword searches, but as described in the calculation amount, it can be expected to change depending on the number of documents and the number of search keywords. In this implementation, the number of documents (Experiment 1) and the number of search keywords (Experiment 2) are changed and compared. In Experiment 1, the number of documents n is varied from 1 to 10000, and 100 words are searched. At this time, the number of searches output by $DTIDX$ is set so that it does not exceed 50 (when n is 100 or less, it is set to $n/2$). In Experiment 2, the number of search keywords is varied from 1 to 10000 and the number of searches output by $DTIDX$ is set to 50 ($n/2$). The hash function is SHA-1[9], the number of hashes is 7, and the prime number p and the generator g are each 1024 bits. The implementation environment is OS: Windows10, CPU: Intel corei7-6700k, Memory:16GB, Python: 3.6

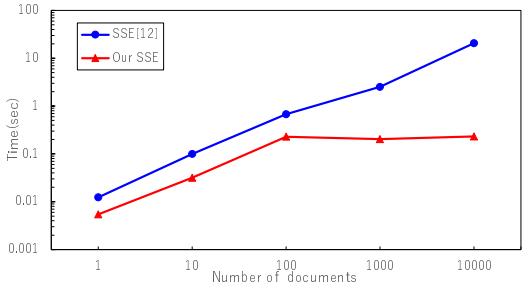

Figure 1. Search time for documents

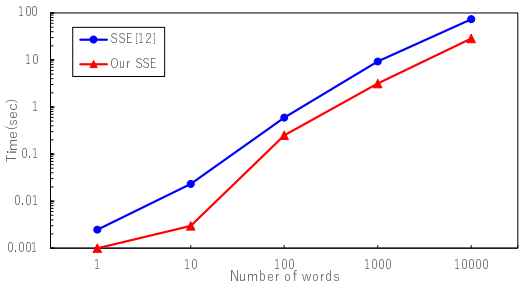

Figure 2. Execute time of keywords

6.2. Evaluation result

Figure 1 shows the results of Experiment 1 and Figure 2 shows the results of Experiment 2. As can be seen from Figure 1, the search time of [12] increases as the number of documents increases, but DXT seems to be constant even when the number of documents increases. Regarding $DTIDX$, if the number of documents increases, the amount of data increases and the search time increases. However, since it is a single keyword to the last, the specific gravity in the entire search time is small. On the other hand, since $CFIDX$ generates only one CBF even if the number of documents increases, only one data set is referred to during retrieval. In addition, since the number of single keyword search results is set so as not to exceed 50, the total search time only calculates 100×50 at most. On the other hand, [12] calculates $100\times n$ because the number of data to be referred to increases as the number of documents increases. Therefore, it is considered that the results shown in Figure 1 that do not depend on DB were obtained.

As shown in Figure 2, the time increases as the number of keywords of both methods increases. However, it can be seen that DXT has about half the search time compared to [12]. Since $DTIDX$ is a single keyword, it is always constant in Experiment 2. As for $CFIDX$, the time increases depending on the number of search keywords. However, since the number of single keyword searches is constant at 50 as in Experiment 1, the calculation amount is $50\times w$. On the other hand, since [12] refers to 100 documents for one keyword, the calculation amount is $100\times w$. Therefore, it is considered that the result shown in Figure 2 was obtained.

Evaluate Figure 1 and Figure 2, [12] increased in time according to the number of DBs and the number of keywords, but in DXT, it was found that it depended on the number of keywords but not on the number of DBs. On the other hand, there is a big difference between 10 and 100 in Figure 1 regardless of the number of DBs. This is thought to come from the difference in the number of single keyword search results. When the number of DBs is 10, the number of search results is 5, and when it is 100, it is 50. In search on $CFIDX$, considered to depend on the result of $DTIDX$, not the number of DBs, because the trapdoor is generated depending on the number of single keyword search results as described in the construction. Also, looking at the real time, searching for 100 words for 10000 DBs is 0.28 seconds, so it is practical.

7. Conclusion

In this paper, we proposed two SSEs that can update and efficiently search for conjunction keywords by employing Counting Bloom Filter. The multiple keywords search that we normally perform can be performed safely and efficiently, and the update can be processed without adding significant changes to the data, making it easy to implement. Further, even if the number of documents increases, a search can be performed at high speed depending on conditions. From the implementation results, it can be said that 100 words can be searched in 0.28 seconds.

This protocol can only search for intersections with conjunction keywords. For this reason, future research will focus on SSE proposals that can handle arbitrary logical expressions using multiple keywords, and SSEs that cause less information leakage.

References

[1] Shangqi Lai,Sikhar Patranabis,Amin Sakzad,Joseph K. Liu,Debdeep Mukhopadhyay,Ron Steinfeld,Shi-Feng Sun,Dongxi Liu,Cong Zuo.Result Pattern Hiding Searchable Encryption for Conjunctive Queries.CCS ' 18, October 1519, 2018,

[2] D. Cash, S. Jarecki, C.S. Jutla, H. Krawczyk, M-C. Rosu, and M. Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In CRYPTO ' 13. 353373.

[3] D. Cash, J. Jaeger, S. Jarecki, C.S. Jutla, H. Krawczyk, M-C. Rosu, and M. Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In NDSS ' 14.

[4] R. Curtmola, J.A. Garay, S. Kamara, and R. Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In ACM CCS ' 06. 7988.

[5] Motoki Miyano, Satoshi Obana:Updatable Searchable Symmetric Encryption with Fine-Grained Delete Functionality.CANDAR Workshops 2018: 438-444

[6] E.-J. Goh. Secure Indexes. Cryptology ePrint Archive, Report 2003/216. http://eprint.iacr.org/2003/216, 2003.

[7] B. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422426, Jul 1970.

[8] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In Proceeding of SIGCOMM ' 98, 1998

[9] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, Proceedings of Crypto 1996, volume 1109 of Lecture Notes in Computer Science, pages 115. Springer-Verlag, Aug 1996.

[10] D.X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. IEEE Symposium on Security and Privacy, pp.44-55, 2000.

[11] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. ACM Conference on Computer and Communications Security, pp.965-976, 2012.

[12] Leyla Tekin, Serap Sahin. Implementation and Evaluation of Improved Secure Index Scheme Using Standard and Counting Bloom filters. INTERNATIONAL JOURNAL OF INFORMATION SECURITY SCIENCE, Vol.6, No.4, pp46-56, 2017

[13] S. Bellovin and W. Cheswick. Privacy-enhanced searches using encrypted bloom filters. Cryp- tology ePrint Archive, Report 2004/022, Feb 2004. http://eprint.iacr.org/2004/022/.

[14] S. Tarkoma, C.E. Rothenberg, E. Lagerspetz. Theory and practice of bloom filters for distributed systems. IEEE Communications Surveys and Tutorials, Vol.14, No.1, pp.131-155, 2012.

[15] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. ACNS, Vol.5, pp.442- 455, 2005.

[16] Kamara, S., Papamanthou, C.: Parallel and Dynamic Searchable Symmetric Encryption. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258274. Springer, Heidelberg 2013.