Wuhan University
Journal of Natural Sciences

2013, Vol.18 No.5, 393-401

Article ID 1007-1202(2013)05-0393-09
DOI 10.1007/s11859-013-0947-3

A Novel Fuzzy Keyword Retrieval Scheme over Encrypted Cloud Data

□ **TIAN Ke, ZHANG Weiming, LI Ke, WU Junming, YU Nenghai†**

School of Information Science and Technology, University of Science and Technology of China, Hefei 230027, Anhui, China

Abstract: In this paper, we focus on the fuzzy keyword search problem over the encrypted cloud data in the cloud computing and propose a novel Two-Step-Bloom-Secure-Filter (TSBSF) scheme based on Bloom filter to realize the efficiency and flexibility of data use. The proposed scheme not only reduces the space complexity significantly but also supports the data update with low time complexity and guarantees the search accuracy. Experimental results on real world data have certified the validity and practicality of this novel method.

Key words: fuzzy search; privacy preserving; cloud computing
CLC number: TP 309.2

Received date: 2013-04-30
Foundation item: Supported by the National Natural Science Foundation of China (61170234) (60803155), the Strategic Priority Research Program of the Chinese Academy of Sciences (XDA06030601), and the Funding of Science and Technology on Information Assurance Laboratory (KJ-13-02)
Biography: TIAN Ke, male, Ph.D. candidate, research direction: network communications and information security. E-mail: tianke@mail.ustc.edu.cn, ketian@vt.edu
† To whom correspondence should be addressed. E-mail: ynh@ustc.edu.cn

0 Introduction

In recent years, the rapid growth of the data especially the appearance of the Big Data impels the development of cloud computing [1]. At present, more and more important data will be outsourced from the local storage to the cloud server. With the popularity of cloud storage, many corporations and companies are still afraid of outsourcing their data to the cloud server for the reason that their data may be leaked and then abused by cloud sever. In order to protect data privacy, the best solution is encryption; in other words, the data owner will encrypt his/her data before they outsource them to the remote cloud server. After encryption, we have to face a new challenge-how to make best use of these encrypted data to achieve efficient application, such as information retrieval for the reason that searching is always the prior demand for the users. All the files are encrypted in the cloud server, and the decryption keys are only shared by the data owner and users. Without the keys, the cloud server will never know the exact content of files. Traditional symmetric encryption needs the users to download all the data, decrypt all the data, and search keyword as plaintext search. This method not only costs user plenty of time but also is rather impractical with the huge increase of data. In order to solve the problem, searchable encryption (SE) [2] has been proposed. SE is a very useful technology to help users directly do operations on the encrypted data. However, deploying this technology to large-scale cloud data will cost plenty of time and make the search inefficient. To solve the problem, building a secure index [3] is suggested.

Furthermore, fuzzy keyword is a hot point in the plaintext information retrieval because users cannot of-

fend the spell mistakes, which is also a creative work in the encrypted database, and many researchers[4-12] have done active study in this area. Li *et al* [6] and Wang *et al* [7] have proposed wildcard-based fuzzy set construction to make a list of fuzzy keyword set from a keyword, which is most popularly used in this area. They also suggest a solution using trie-tree for the search index built, but it has limitations: the first is that the space cost of building an index is very large, and the second is that its update functions are infeasible. Even though several work has been thought up to reduce the cost, such as Chuah *et al* [8] using bedtree, Liu *et al* [9] using secret sharing, but they could not guarantee the high efficiency. Recently, Zhou *et al* [10] proposed a different method to make fuzzy set by introducing *k*-gram. However, this method will also face the efficiency problem.

In this paper, we mainly concentrate on the fuzzy keyword search in the encrypted database in cloud computing. We propose our newly solution, which is called Two-Step-Bloom-Secure-Filter (TSBSF) based on the technique Bloom filter [13], to build the secure index supporting the fuzzy keyword search. Our solution will reduce much more space complexity and guarantee highly efficiency, and it also supports dynamic update and is provable secure. The main contributions of this paper can be summarized as follows:

1) We propose a novel TSBSF scheme to build the secure fuzzy keyword index and fulfill a complete mechanism design for fuzzy keyword search with the satisfactory demand of security.

2) Our scheme is highly efficient and reduces the space complexity significantly according to the experiments.

3) Our scheme also supports the feasible update data and can be easily applied to multi-fuzzy keyword search.

The rest of the paper is organized as follows: In Section 1, we will describe the system model and preliminaries. In Section 2, we will describe our own scheme and take an analysis on it. We present the experiment results in Section 3 and draw a conclusion in Section 4.

1 Problem Formulation

In this section, we describe our searching system model and the privacy requirements. Then, we will describe some preliminaries related to our work.

1.1 System Model

Different from traditional information retrieval, the encrypted database retrieval always has three entities: the data owner, the remote cloud server, and the users. The data owner can be the individual or corporations who own a collection of n document plain files $D=\{D_1,D_2,\cdots,D_n\}$. To protect the privacy, the owner encrypts the files into $\varepsilon=\{E_1,E_2,\cdots,E_n\}$ and outsources them to the remote cloud server with the secure index I to support the fuzzy keyword search. I is built from the keyword dictionary set $W=\{W_1,W_2,\cdots,W_m\}$, and m is total number of the different keyword. I is donated as $I=\{I_1,I_2,\cdots,I_m\}$.

To search over the encrypted files ε, the data user will first make a query Q consisting of keyword the user wants to search and turn Q into trapdoor T using the encrypted key. The cloud server will receive T and search it in the index I and then find the match keyword ID set $\Delta=\{\mathrm{ID}_1,\mathrm{ID}_2,\cdots,\mathrm{ID}_k\}$. With Δ, the server can easily find the corresponding files ID and then return the result R, which consists of the resulting encrypted files; finally, the user decrypts R and gets the plaintext files. Since all the procedures are isolated from cloud servers, the cloud server will not know the keyword the user search nor the information about return files.

Updating the index means that owner add a new keyword W_r to dictionary W or a new entry I_r to index I, which is a worthy problem because updating may modify the whole index and increase the time and space consume obviously.

The whole framework can be divided into six algorithms:

1) KenGen

The data owner generates index encryption keys $K=\{K_1,K_2,\cdots,K_t\}$ for Hash functions and traditional symmetric keys K' for encrypting and decrypting files. The keys are shared with the owner and users.

2) FuzzySetMake

For every different keyword in the dictionary, the data owner builds corresponding fuzzy keyword set by adopting construct functions.

3) BuildIndex

The data owner builds a privacy-protecting index I, supporting fuzzy keyword search from the database D based on K. After building the index, the owner can send both I and encrypted ε to cloud server.

4) TrapGen

The data user can generate the trapdoor T according to the query Q including the interesting terms.

5) Search

The cloud server compares the T with each entry in I and returns the match result R to the user.

6) Update

The data owner adds new entries into the dictionary W and index I, and reserves the changes in the cloud storage.

We will discuss more detail in Section 2 and focus on the BuildIndex, Search, and Update because others can be done in traditional ways. The whole framework is presented in Fig.1.

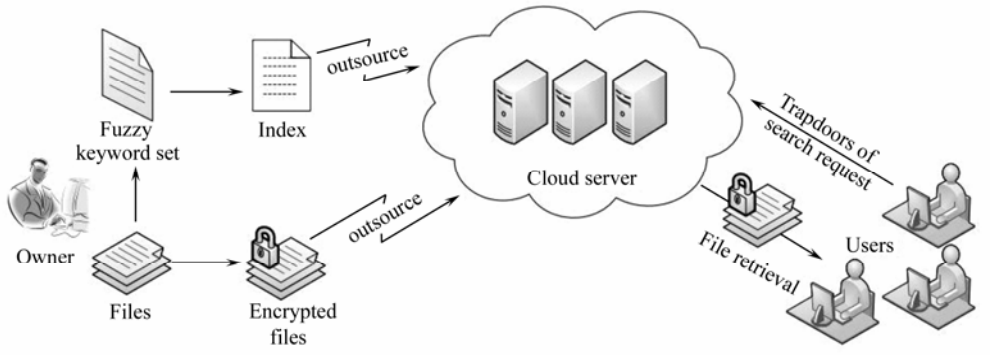

Fig. 1 System model of fuzzy keyword search

1.2 Privacy Requirements

The cloud server is been regarded as "semi-honest", also called "honest-but-curious". Specifically, a cloud server will not remove encrypted data files or index from the storage. It will also correctly follow the designated protocol specification. However, it is curious to analyze data (including data, trapdoor, and index) in the storage and flow messages in order to learn additional information. This threat model is defined as known background model. Curtmola *et al* [2] proves that the model that we adopt can meet the nonadapt semantic security, and stored data privacy can be protected by traditional symmetric encryption.

1.3 Preliminaries

Edit Distance Given two strings S_i and S_j, the edit distance[14] between S_i and S_j is defined as minimum number of primitive operations(including character deletion, insertion, and substitution) needed to transform from S_i to S_j, denoted by edit(S_i, S_j).

Trie-Tree A trie-tree [15], also described as digital tree or prefix tree, is an ordered tree data structure to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string.

Bloom Filter Bloom filter[13] is a data structure with high space efficiency used to answer set member queries. It is initially set to 0 with an array of b bits length. Generally, there are r independent Hash functions (like MD5, SHA-1, etc.). h_t :{0,1}*→[1,b], t = 1,⋯, r, which means that with one function, each element is mapped to a position in b array bits. Here is a simple map shows how Bloom filter works (Fig.2): in this example, array b=9, r=4, the term "flower" has been mapped to four positions P={2,3,6,9}. Bloom filter has a possibility of false positives, because the positions of an element may have been set by one or more other elements. Even though Bloom filter is a useful technology, however, directly applying it to the fuzzy keyword search will cause serious error rate and cannot guarantee the efficiency. We will regard the original Bloom filter as baseline and show the significantly improvement by applying our TSBSF scheme in the next section.

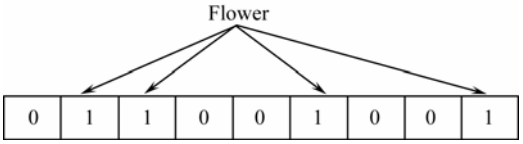

Fig. 2 A simple example of Bloom filter

2 Proposed Scheme

Before describing our own scheme, we get a short review of scheme of Wang *et al*[7], in which they build the corresponding fuzzy set from each keyword, and then map each term in the set to be l=160 bits stream using Hash functions. Then, each θ bit part is regarded as a character in the trie-tree. After dividing the string into "l / θ" parts, eventually, Wang *et al* [7] builds a single trie-tree including all the fuzzy keyword set. As mentioned in the

introduction, this scheme has both space and efficiency limitations. We will fulfill our mechanism in the following section, which is ordered as different functions we need.

2.1 Fuzzy Keyword Set Construction

Building a practical fuzzy keyword set is our first challenge to achieve fuzzy keyword retrieval. We define the $S_{w,d}$ as the fuzzy keyword set of w, where any $w' \in S_{w,d}$, edit(w, w') $\leqslant d$, and denote the number of members including in $S_{w,d}$ by $|S_{w,d}|$. Wang *et al* [7] have proposed a wildcard-based fuzzy set construction for a keyword of length l in edit distance d. If the edit distance d=1, this algorithm can reduce space complexity from $(2l+1)\times 26+1$ to $2l+1$. For example, when w_i = cate, $S_{w_i,1}$ ={*cate, cate, *ate, c*te, c*ate, ca*te, ca*e, ca*te, cat*}. Furthermore, Wang *et al* [7] have proved that the intersection of the similarity sets $S_{w_i,d}$ and $S_{w,d}$ for keyword w_i, and searching input w is not empty if and only if edit(w_i,w)$\leqslant d$. Since this technology is practical, we will also adopt this wildcard-based construction. That is, for an input w and edit distance $d>1$, we calculate all the fuzzy keyword set on the condition $d'<d$ like $S_{w,d}=S_{w,d-1}\bigcup S_{w,d-2}\bigcup\cdots\bigcup S_{w,1}$. If edit distance d=1, we can directly insert * into the keyword w in order. Following is the detail about this algorithm (Algorithm 1). By using this algorithm, we can get the efficient fuzzy keyword set without affecting its searching correctness.

Algorithm 1 Fuzzy keyword set construction

Input:
Keyword w_i;
Edit distance d;
Output:
Fuzzy keyword set $S_{w,d}$

```
if d>1 then CreatFuzzySet(w_i, d-1)
end if
if d=0 then set S_{w,d}=w_i
else
  for k←1 to |S_{w,d-1}| do
    for j←1 to |S_{w,d-1}[k]|+1 do
      if j is odd then
        set variable as S_{w,d-1}[k]
        insert * at position ⌊(j+1)/2⌋
      else
        set variable as S_{w,d-1}[k]
        replace ⌊j/2⌋-th position with *
      end if
      if variable is not in S_{w,d-1} then
        S_{w,d-1}=S_{w,d-1} ∪ {variable}
      end if
    end for
  end for
end if
```

2.2 Making Efficient Secure Index

In this section, we will talk about our novel TSBSF scheme in detail. A symbol-based tri-traverse searching scheme based on the data structure of trie-tree has been proposed before. However, it will confront a big challenge that the method will consume space largely and cannot support update perfectly. Thus, space complexity must be taken into consideration, and supporting efficient update will also be an important factor to attract users using the cloud service. With the selectable parameters, our scheme can solve these problems perfectly.

2.2.1 TSBSF scheme

The reason why we called the proposed scheme the TSBSF is that, compared with original Bloom Filter, we have recorded not only the position information p but also the gray information g. The definition of gray information comes from the image gray value to display the weight quality in one position. We use both position information and gray information to identify the corresponding keyword.

For each $w\in S_{w,d}$, the first step is to calculate the position information $h_p:\{0,1\}^*\rightarrow[1,b]$ using function POS; This step is similar to the original Bloom Filter, and POS is a function with which we can select the position, where w' will be mapped to. That is, with POS function, we get the random position number with the input seeds including our Hash result and the keys K. The second step is to calculate the gray information $h_g:\{0,1\}^*\rightarrow[1,h]$ with the pseudo-random function GRAY with seeds K_i and $\{0,1\}^*$, where h is the maximum number of gray information.

The detail about TSBSF is shown in Algorithm 2.

Algorithm 2 TSBSF

Input:
One fuzzy keyword set $S_{w,d}$
Encrypt key set K
Output:
Position information set P
Gray information set G

```
P=∅, G=∅
for i←1 to |S_{w,d}| do
  operation on w_i ∈ S_{w,d}
  for j←1 to |K| do
    {0,1}* ←Hash(w_i)
    p_0←POS({0,1}*,K_j)
    g_0←GRAY({0,1}*,K_j)
    insert p_0 into P
    insert g_0 into G
  end for
end for
```

Because gray information is also a random number influenced by the seeds we input, it is nearly impossible for two different terms be mapped to the same position with same gray information. In Algorithm 2, each term w_i ($w_i \in S_{w,d}$) will be mapped to $|K|$ positions with change of the secret key K_i, and each position is associated with its corresponding gray information. Thus, each term will be unique in the view of statistics, because two terms have very low rate (almost close to 0) to turned into same vector in both position information and gray information areas in our situation. In a word, given a fuzzy keyword set $S_{w,d}$, we calculate a vector of position information P with POS function and a corresponding vector of gray information Q at GRAY function.

The experiment will improve the accuracy in next section "choosing the parameters". Our advantage lies in the fact that we use two random variables to control each term at the same time, and this method will help us distinguish two different terms much more correctly.

2.2.2 Building index

We will build a secure index by using TSBSF mentioned before. However, a question we need to answer is that since the different term may be mapped to the same position by using Bloom filter, we will also have the same problem even though we include the gray information. For example, $i, j \in [1, |P|]$, $p_i = p_j$, but $q_i \neq q_j$, what should we do to fill them into the array? The original Bloom filter will be set position to 1; since we have introduced gray information to distinguish the different keyword, we set the position to $I_k[p_i] = \max(q_i, q_j)$, where I_k is one entry of the index I and is a vector of b length. Moreover, this modification may have bad effect on our search accuracy, but we will introduce a fault tolerant rate ε to defend this modification and guarantee the accuracy. To further protect the privacy, we can choose random distortion into the index, which means that we can embed some random numbers into random positions.

More discussion is shown in the following section. Our built index algorithm can be described as follows: given a keyword collection W, encrypt key set K, and edit distance d; to each $w_i \in W$, we first calculate fuzzy set $S_{w,d}$ with wildcard-based construction. Second, we calculate the position information vector P and gray information vector G with the TSBSF algorithm. Then, we build a single Bloom filter vector I_i containing information of P and G mentioned above. In the end, we build the entire index I by repeating the operations for each keyword w_i and gather them I_i together. The detailed building fuzzy index algorithm is in Algorithm 3.

Algorithm 3 Building fuzzy secure index

Input:
Keyword collection W
Encrypt key set K
Edit distance d

Output:
Secure fuzzy keyword index I

```
1   I=∅
2   for i←1 to |W| do
3   setup I_i
4   S_{w,d} ←CreatFuzzySet(w_i) using Algorithm 1
5   {P, G}←TSBSF (S_{w,d}, K)  using Algorithm 2
6   for j←1 to |W| do
7     I_i [p_j]=max( I_i [p_j], g_i)
8   end for
9   insert I_i into I
10  end for
```

2.3 Searching Algorithm

When we input query Q, we need TrapGen algorithm to make a trapdoor T from Q. Because TrapGen algorithm is similar to that in building one entry of index, we regard TrapGen as one step to make a search from the index I. In the search algorithm, the user will first generate the trapdoor T from a query set Q. The cloud server will receive T and search it in the secure index I. T and $I_i \in I$ are vectors of same b length, and for each $I_i[j] \subseteq I_i$, we have $I_i[j] \subseteq [1,h], j \in [1,b]$. Comparing T with each I_i, we calculate the similarity rate by using the measurement called fault tolerant rate ε_0 which we will talk in detail in the next section. If this similarity rate is within our acceptable range, we then add the word ID into our result set R', else we regard this word beyond our interest. After processing the result of matching keyword set R', the server will return the encrypted files collection $R = \{E_1, E_2, \cdots\}$ by finding those files containing similarity keyword listed in R'. The user can decrypt R with the sharing keys and get the plaintext files. The main algorithm is shown in Algorithm 4.

2.4 Choosing the Parameters

Since updating data is isolated from the basic system framework, before introducing the update algorithm, we will display how to choose the best parameters to make building index and searching efficiency. We will discuss it into two parts: the search parameter fault tolerant rate ε and the building parameter embedding rate λ. The experiment results show that these parameters can have huge influence on our scheme.

Algorithm 4 Search the index

Input:
Search query Q
Encrypt key set K
Secure fuzzy keyeords index I
Output:
The matching files set R

```
R=∅, R′=∅
T=TrapGen(Q)
for i←1 to |I| do
count=0,tolerate=0
 for j←1 to b do
  if T[j]≥0 and I_i[j]≥T[j] then
break
  end if
  if T[j]≥0 then
   if I_i[j]=T[j] then
    count++
   end if
  else
   count++, tolerate++
  end if
 end for
 if tolerate/count<ε_0 and count>|K| then
  insert word ID_i into R′
 end if
end for
get R from R′
```

2.4.1 Fault tolerant rate

Fault tolerant rate is defined as follows, where N(count) in Eq.(1) is the number of count, and N(tolerate) is the number of tolerate in Algorithm 4.

$$\varepsilon = \frac{N(\text{tolerate})}{N(\text{count})} \tag{1}$$

To help understand the definition better, we take an analysis based on a given example. Considering that the edit distance is d, our search query is term w' and turn it into trapdoor T'_w, when we use the trapdoor T'_w to search the $I_i \subseteq I$. If edit $(w', w_i) < d$, then the fuzzy keyword set $S_{w',d} \bigcap S_{w_i,d} \neq \varnothing$. Moreover, we assume that $w^t \in S_{w',d} \bigcap S_{w_i,d}$. According to Algorithm 2, the w^t will be mapped to $|K|$ number of positions, each with related gray information. Thus, $|T|$ and I_i will share more than K same positions with related same gray information in each position in the view of statistics. This is why we choose the criterion that count $>|K|$.

However, looking back at Algorithm 3, we may encounter a problem called fault tolerant rate, which is first mentioned in the part of "build the index". For $j \in [0, b]$, it was compared with $T[j]$ and $I_i[j]$, where I_i is the entry index by the input of keyword w_i, as shown in Algorithm 3. If $T[j] < I_i[j]$ at the position j, we cannot make sure whether this position has the same gray information. There are two possible happenings: one is that they do have the same gray information at the beginning, but the other word $w_x \in S_{w_i,d}$ has been mapped to the position j, and its larger gray information covers the original one, which causes $T[j] < I_i[j]$; the second is that they are different terms and just been mapped into the same positions with different gray information. In order to solve this problem, we introduce N(count) to present the total number sharing the same position and N(tolerate) to present the number of mistakes.

The precision and recall [16] rate are two criterion parameters to evaluate the accuracy of the searching. We evaluate our scheme accuracy by comparing our search results with plaintext results to calculate the precision rate and recall rate. It is easy to conclude that the larger ε_0, the more likely we will include the false keyword into our result, and the recall will increase and precision will decrease; the smaller the ε_0, the most likely that we delete some matching keyword, and the recall will decrease, and precision will increase. We have done an experiment to make the best choice of ε_0, as shown in Fig. 3. High precision will guarantee the accuracy, and high recall will guarantee the completeness. We test the 500 keyword set and random queries, the results show that when ε_0=0.6, we get the best performance, which is our expectation.

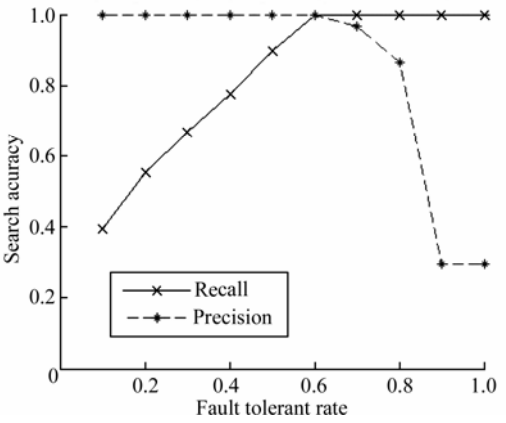

Fig. 3 Search accuracy with different fault tolerant rate

2.4.2 Embedding rate

In this part, we will mainly focus on the space and time complexity. Our goal is to reduce the complexity as much as possible without the loss of efficiency. We define the embedding rate λ as

$$\lambda_d = \frac{|S_{\bar{w},d}| \times |K|}{L} \tag{2}$$

Where, $S_{\overline{w},d}$ means the average size of the fuzzy set in terms of d edit distance, K is the number of the encrypt keys for index security, which is the same as the number of POS functions mentioned in the "searching algorithm", and L is the length of vector array that is equal to b in our algorithm above.

Since the edit distance d=1 is most practical because most people would not type wrong words with $d>1$ in the view of statistics. Moreover, our experiment is mainly in the condition d=1; therefore, we can simplify Eq. (2) into

$$\lambda_{d=1}=\frac{(2\times\overline{l}+1)\times|K|}{L} \tag{3}$$

Where, $\overline{l}$ is average length of the keyword set. Considering that $\overline{l}$ is dependent on the file collections and there are two variable quantities, we define

$$\varphi=\frac{1}{\frac{|K|}{L}}=\frac{L}{|K|} \tag{4}$$

Considering that $|K|$ is the number of POS functions, a large $|K|$ will increase the time complexity, but a small $|K|$ can affect the accuracy; in the experiment, we choose $|K|$=4 as the best option. However, we more focus on the space complexity. Because with the growing computing capacity, the server will deal with data much more quickly, and huge stored data will be a disturbing problem. We will describe how to choose the best φ in detail, if we reduce φ, and many different keyword may be mapped to the same position and result in high fault tolerant rate as mentioned above. Furthermore, we use the origin Bloom filter as a baseline to show the improvement of our scheme. The experiment is displayed in Figs. 4 and 5. As shown, when we choose $\varphi\approx100$, we get the best result, which makes precision rate and recall rate up to 100%. In this situation, $\lambda_{d=1}\approx0.173$, which means that in every 100 positions, nearly 17 different positions have been mapped. Although the experiments work perfectly, this still needs mathematical proof, and we will do an analysis in our future work. We expect to find a formula to describe the deep relationship between λ and ε.

2.5 Support Update Algorithm

One of our contributions in this paper is supporting the update files. Generally speaking, the index built from mass data will consume huge space; this is why we need to store it into cloud. It is unrealistic to download the index, rebuild it, and upload to the cloud server. Thus, it is very essential to support efficient update data. Our update algorithm can be described as follows: given a keyword w_i, first, we make sure whether this keyword has already existed. If this keyword is a new one, we just need to set up a new I_{new} using the input w_i with the TSBSF algorithm. After we build I_{new}, we only need to attach this entry of index to the original one without any change in the original index. The same with that is when we want to delete some keyword, we just need to remove the corresponding entry of index from the original one. Comparing this with trie-tree, if there is a new character added into the original tree, the whole index has to be repeat calculated to adjust to this change. In other words, our update algorithm is much more feasible. Given a new keyword, the data owner does not need to modify the whole secure index; all he has to do is to make new entries and combine them with the former ones. It is significantly useful in the cloud storage. Furthermore, the time complexity is $O(1)$ with one operation, which has the low time cost. The complete algorithm is shown in Algorithm 5.

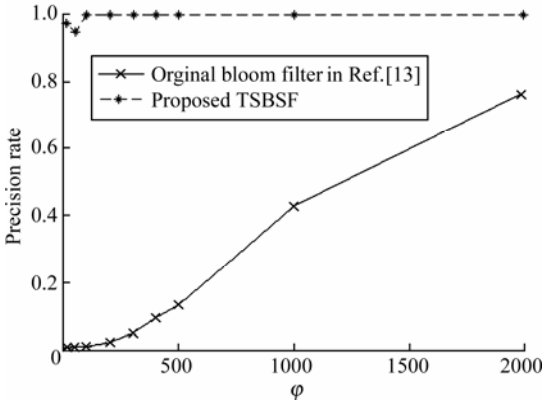

Fig. 4 Precision rate with different φ

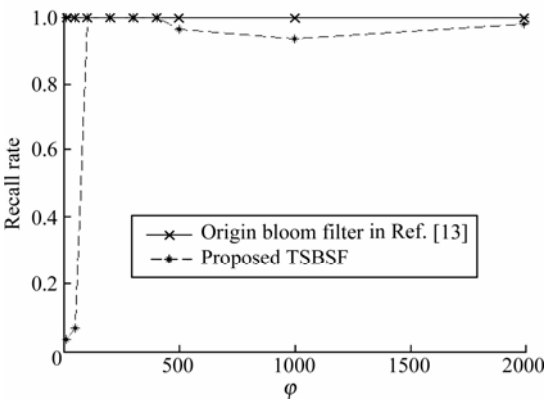

Fig. 5 Recall rate with different φ

2.6 Support Multi-Fuzzy Keyword Search

The proposed scheme can also be easily applied to multi-fuzzy keyword search. Considering a simple example that we want search for the multi-fuzzy keyword $q_2=\{w_1,w_2\}$ at the same time in the condition that edit

Algorithm 5 Update the index

Input:
New keyword set W_{new}
Keyword set W
Secure fuzzy keyeords index I
Encrypt key set K

Output:
Updated secure index I_{new}

```
for i←1 to |W_new| do
if w_new,i ∈ W then
    goto next keyword in W_new
else
    set up I_new
    build I_new with w_new using Algorithm 3
    insert I_new into I
end if
update I into I_new
end for
```

distance d. First we can use wildcard-based construction to make each fuzzy set $S_{w_1,d}$ and $S_{w_2,d}$. Because d is the total edit distance of Q, we choose $\forall w' \in S_{w_1,d}$ and $w'' \in S_{w_2,d}$, st. $\text{edit}(w', w_1) + \text{edit}(w'', w_2) < d$, Thus we can generate two new $Q_{w'}$ and $Q_{w''}$. With the Search algorithm, we get the result $R = R' \bigcap R''$. This method can be easily expanded to search $n\,(n > 2)$ multi-fuzzy keyword.

3 Results and Comparison

In this section, we will compare TSBSF scheme with Wang *et al*'s trie-tree algorithm. For the sake of fairness, we will compare these algorithms on the same level of the accuracy. Taking an analysis of all the possible existed algorithms, to our best knowledge, their algorithm is the most represented one, others will confront the efficiency or accuracy problems. We have already done several experiments to choose the suitable parameters in above section. With these selectable parameters, Figures 4 and 5 have displayed high efficiency of precision rate and recall rate. On the condition of guaranteeing the high accuracy, we will then focus on the space and time complexity especially the space compared with Wang *et al*'s trie-tree algorithm [7]. To sum up, we demonstrate a thorough experimental evaluation on the TREC data [17], which consists of 7 594 documents and 16 864 distinct terms. The experiment is implemented by C++ language and conducted on a computer with i-3 CORE 2.13 GHz processor and Windows 7 home basic system. Furthermore, we choose d=1, $|K|$=4, h=256, φ=100, and ε=0.6. Moreover, we use SHA-1 as our Hash function, which was suggested by Wang *et al*[7]. According to Wang *et al*'s algorithm [7], we choose l=160 bits, θ=8.

As shown in Fig. 6, our space consumed is much less than the trie-tree, and our space consumed increases much slower than the trie-tree with the increase in the keyword. The time cost comparison is listed in Table 1, which shows that even though our search time is longer than that for trie-tree, considering the situation that when we input a query Q including a keyword w_1, we will make a trapdoor T and need just one search in the index I. However, trie-tree needs turn each $w' \in S_{w,d}$ into a distinct trapdoor T and search it in the index I, so it will need to search a total of $|S_{w,d}|$ times and may leak more information. In this situation, our scheme performs better than Wang *et al*'s scheme [7].

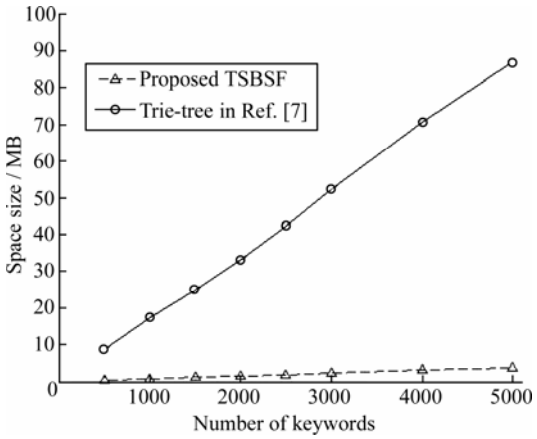

Fig. 6 Space complexity comparison between Wang *et al*'s scheme and proposed TSBSF scheme

Table 1 Time complexity comparison between Wang *et al*'s scheme and proposed TSBSF scheme

Scheme	BuildIndex	TrapGen	Search	Update
TSBSF	$O(\|W\|)$	$O(1)$	$O(\|W\|)$	$O(1)$
Trie-tree	$O(\|W\|)$	$O(1)$	$O(1)$	—

4 Conclusion

In this paper, we fulfill a system model to realize a fuzzy keyword search in the encrypted database in cloud computing. To meet the challenge that guarantee the efficient fuzzy search without privacy leaking, we proposed a novel TSBSF scheme to achieve this goal. Experiment shows that the novel scheme achieves advantages both in efficiency and storage space.

References

[1] Armbrust M, Fox A, Griffith R, *et al*. A view of cloud comput-

ting [J]. *Communications of the ACM*, 2010, **53**(4): 50-58.

[2] Curtmola R, Garay J, Kamara S, *et al*. Searchable symmetric encryption: improved definitions and efficient constructions [C]//*Proc* 13*th ACM Conference on Computer and communications security*. New York: ACM Press, 2006: 79-88.

[3] Shmueli E, Waisenberg R, Elovici Y, *et al*. Designing secure indexes for encrypted databases [C]//*Proc* 19*th Annual IFIP WG* 11.3 *Working Conference on Data and Applications Security*. Berlin, Heidelberg: Springer-Verlag, 2005: 54- 68.

[4] Yang C, Zhang W, Xu J, *et al*. A fast privacy-preserving multi-keyword search scheme on cloud data [C]//*Proc* 2012 *International Conference on Cloud and Service Computing* (*CSC*). Shanghai: IEEE Press, 2012: 104-110.

[5] Xu J, Zhang W, Yang C, *et al*. Two-Step-Ranking secure multi-keyword search over encrypted cloud data [C]//*Proc* 2012 *International Conference on Cloud and Service Computing* (*CSC*). Shanghai: IEEE Press, 2012: 124-130.

[6] Li J, Wang Q, Wang C, *et al*. Fuzzy keyword search over encrypted data in cloud computing [C]//*Proc* 30*th IEEE International Conference on Computer Communications* (*INFOCOM*). San Diego: IEEE Press, 2010: 1-5.

[7] Wang C, Ren K, Yu S, *et al*. Achieving usable and privacy-assured similarity search over outsourced cloud data [C]//*Proc* 32*th IEEE International Conference on Computer Communications* (*INFOCOM*). Orlando: IEEE Press, 2012: 451-459.

[8] Chuah M, Hu W. Privacy-aware bedtree based solution for fuzzy multi-keyword search over encrypted data [C]//*Proc* 31*st International Conference on Distributed Computing Systems Workshops* (*ICDCSW*). Minneapolis: IEEE Press, 2011: 273-281.

[9] Liu C, Zhu L, Li L, *et al*. Fuzzy keyword search on encrypted cloud storage data with small index [C]//*Proc* 2011 *International Conference on Cloud Computing and Intelligence Systems* (*CCIS*). Beijing: IEEE Press, 2011: 269-273.

[10] Zhou W, Liu L, Jing H, *et al*. K-gram based fuzzy key-word search over encrypted cloud computing [J]. *Journal of Software Engineering and Applications*, 2013, **6**(1): 29- 32.

[11] Wang J, Ma H, Tang Q, *et al*. A new efficient verifiable fuzzy keyword search scheme [J]. *Journal of Wireless Mobile Networks*, *Ubiquitous Computing and Dependable Applications*, 2012, **3**(4): 61-71.

[12] Ballard L, Kamara S, Monrose F. Achieving efficient conjunctive keyword searches over encrypted data [C]//*Proc* 7*th International Conference on Information and Communications Security* (*ICICS*). Berlin, Heidelberg: Springer-Verlag, 2005: 414-426.

[13] Bloom B H. Space/time trade-offs in Hash coding with allowable errors [J]. *Communications of the ACM*, 1970, **13**(7): 422-426.

[14] Levenshtein V. Binary codes capable of correcting spurious insertions and deletions of ones [J]. *Problems of Information Transmission*, 1965, **1**(1): 8-17.

[15] De La Briandais R. File searching using variable length keys [C]//*Proc Western Joint Computer Conference*. New York: ACM Press, 1959: 295-298.

[16] Morita M, Shinoda Y. Information filtering based on user behavior analysis and best match text retrieval [C]//*Proc* 17*th International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York: Springer-Verlag, 1994: 272-281.

[17] Bailey P, Craswell N, Soboroff I, *et al*. The CSIRO enterprise search test collection [C]//*ACM Special Interest Group on Information Retrieval*. New York: ACM Press, 2007, **41**(2): 42-45.

□