International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 4, Issue 6, June 2017

An Efficient and Reliable Diverse Keyword Ranked Search Scheme Over Ciphered Cloud Data

[1] Ms. Shruthi, [2] Mehul Bhatt, [3] Pushpa J, [4] Sobin Baby, [5] Akhil K. Manoj
[1]Asst. Prof.,[2][3][4]UG Student
[1][2][3][4] Dept. of CSE, RR Institute of Technology, Bangalore, Karnataka

Abstract— **Due to the increasing popularity of cloud computing, more and more data owners are motivated to outsource their data to cloud servers for great convenience and reduced cost in data management. However, sensitive data should be encrypted before outsourcing for privacy requirements, which obsoletes data utilization like keyword-based document retrieval. In this paper, we present a secure multi-keyword ranked search scheme over encrypted cloud data, which simultaneously supports dynamic update operations like deletion and insertion of documents. Specifically, the vector space model and the widely-used TFxIDF model are combined in the index construction and query generation. We construct a special tree-based index structure and propose a "Greedy Depth-first Search" algorithm to provide efficient multi-keyword ranked search. The secure kNN algorithm is utilized to encrypt the index and query vectors, and meanwhile ensure accurate relevance score calculation between encrypted index and query vectors. In order to resist statistical attacks, phantom terms are added to the index vector for blinding search results. Due to the use of our special tree-based index structure, the proposed scheme can achieve sub-linear search time and deal with the deletion and insertion of documents flexibly.**

Index Terms— **Encryption, Diverse-keyword ranked search, Dynamic update, cloud computing.**

1 INTRODUCTION

CLOUD computing has been considered as a newmmodel of enterprise IT infrastructure, which can organize huge resource of computing, storage and applications, and enable users to enjoy ubiquitous, convenient and on-demand network access to a shared pool of configurable computing resources with great efficiency and minimal economic overhead. Attracted by these appealing features, both individuals and enterprises are motivated to outsource their data to the cloud, instead of purchasing software and hardware to manage the data themselves. Despite of the various advantages of cloud services, outsourcing sensitive information (such as e-mails, per-sonal health records, company finance data, government documents, etc.) to remote servers brings privacy con-cerns. The cloud service providers (CSPs) that keep the data for users may access users' sensitive information without authorization. A general approach to protect the data confidentiality is to encrypt the data before outsourcing . However, this will cause a huge cost in terms of data usability. For example, the existing tech-niques on keyword-based information retrieval, which are widely used on the plaintext data, cannot be directly applied on the encrypted data. Downloading all the data from the cloud and decrypt locally is obviously impractical. In order to address the above problem, researcher-s have designed some general-purpose solutions with fully homomorphic encryption or oblivious RAMs. However, these methods are not practical due to their high computational overhead for both the cloud sever and user. On the contrary, more practical special-purpose solutions, such as searchable encryption (SE) schemes have made specific contributions in terms of efficiency, functionality and security. Searchable encryption schemes enable the client to store the encrypted data to the cloud and execute keyword search over ciphertext domain. So far, abundant works have been proposed under different threat models to achieve various search functionality, such as single keyword search, similarity search, multi-keyword boolean search, ranked search, multi-keyword ranked search, etc. Among them, multi-keyword ranked search achieves more and more attention for its practical applicability. Recently, some dynamic schemes have been proposed to support inserting and deleting operations on document collection. These are significant works as it is highly possible that the data owners need to update their data on the cloud server. But few of the dynamic schemes support efficient multi-keyword ranked search.

II. EXISTING SYSTEM

Searchable encryption schemes enable the clients to store the encrypted data to the cloud and execute keyword search over cipher text domain. Due to different cryptography primitives, searchable encryption schemes can be constructed using public key based cryptography or symmetric key based cryptography .Song et al. proposed the first symmetric searchable encryption (SSE) scheme, and the search time of their scheme is linear to the size of the data collection. Goh proposed formal security definitions for SSE and designed a scheme based on Bloom filter. The search time of Goh's scheme is O (n), where n is the cardinality of the document collection. Curtmola et al. proposed two schemes (SSE-1 and SSE-2) which achieve the optimal search time. Their SSE-1 scheme is secure against chosen-keyword attacks (CKA1) and SSE-2 is secure against adaptive chosen-keyword attacks (CKA2). These early works are single keyword boolean search schemes, which are very simple in terms of functionality. Afterward, abundant works have been proposed under different threat models to achieve various search functionality, such as single keyword search, similarity search multi-keyword boolean search ranked search and multi-keyword ranked search Multi-keyword boolean search allows the users to input multiple query keywords to request suitable documents. Among these works, conjunctive keyword search schemes only return the documents that contain all of the query keywords. Disjunctive keyword search schemes return all of the documents that contain a subset of the query keywords. Predicate search schemes are proposed to support both conjunctive and disjunctive search. All these multi keyword search schemes retrieve search results based on the existence of keywords, which cannot provide acceptable result ranking functionality. Ranked search can enable quick search of the most relevant data. Sending back only the top-k most relevant documents can effectively decrease network traffic. Some early works have realized the ranked search using order-preserving techniques, but they are designed only for single keyword search. Cao et al.realized the first privacy-preserving multi-keyword ranked search scheme, in which documents and queries are represented as vectors of dictionary size. With the "coordinate matching", the documents are ranked according to the number of matched query keywords. However, Cao et al.'s scheme does not consider the importance of the different keywords, and thus is not accurate enough. In addition, the search efficiency of the scheme is linear with the cardinality of document collection. Sun et al. presented a secure multi-keyword search scheme that supports similarity-based ranking. The authors constructed a searchable index tree based on vector space model and adopted cosine measure together with TF×IDF to provide ranking results. Sun et al.'s search algorithm achieves better-

DISADVANTAGES OF EXISTING SYSTEM

- Lower search efficiency
- No Keyword privacy

III. PROBLEM FORMULATION

3.1 Notations and Preliminaries

• *W* – The dictionary, namely, the set of keywords, denoted as $W = \{w1; w2; :::; w_m\}$.

• *m* – The total number of keywords in *W*.

• *Wq* – The subset of *W*, representing the keywords in the query.

• F – The plaintext document collection, denoted as a collection of n documents $F = \{f1; f2; :::; fn\}$. Each document f in the collection can be considered as a sequence of keywords.

• *n* – The total number of documents in *F*.

• *C* – The encrypted document collection stored in the cloud server, denoted as $C = \{c1; c2; :::; c_n\}$.

• *T* – The unencrypted form of index tree for the whole document collection *F*.

• *I* – The searchable encrypted tree index generated from *T* .

• *Q* – The query vector for keyword set W_q.

• *TD* – The encrypted form of *Q*, which is named as trapdoor for the search request.

• D_u – The index vector stored in tree node u whose dimension equals to the cardinality of the dictionary

• *W*. Note that the node u can be either a leaf node or an internal node of the tree.

• I_u – The encrypted form of *Du*.

Vector Space Model and Relevance Score Function. Vector space model along with TF×IDF rule is widely used in plaintext information retrieval, which efficiently supports ranked multi-keyword search [34]. Here, the term frequency (TF) is the number of times a given term (keyword) appears within a document, and the inverse document frequency (IDF) is obtained through dividing the cardinality of document collection by the number of documents containing the keyword. In the vector space model, each document is denoted by a vector, whose elements are the normalized TF values of keywords in this document. Each query is also denoted as a vector Q, whose elements are the normalized IDF values of query keywords in the document collection. Naturally, the lengths of both the TF vector and the IDF vector are equal to the total number of keywords, and the dot product of the TF vector Du and the IDF vector Q can be calculated to quantify the relevance between the query and corresponding document. Following are the notations used in our relevance evaluation function:

- $Nf_{;wi}$ – The number of keyword *wi* in document *f*.
- *N* – The total number of documents.
- N_{wi} – The number of documents that contain key-word wi.
- *TF′f;wi* – The TF value of *wi* in document f.
- *IDF′wi* – The IDF value of wi in document collection.
- *TFu;wi* – The normalized *TF* value of keyword *wi* stored in index vector Du.
- *IDFwi* – The normalized IDF value of keyword wi in document collection.

The relevance evaluation function is defined as:

$$\mathrm{RScore}(Du; Q) = Du \cdot Q = \sum_{wi\ Wq} TFu;wi \times IDFwi : (1)$$

If *u* is an internal node of the tree, *TFu;wi* is calculated from index vectors in the child nodes of *u*. If the *u* is a leaf node, *TFu;wi* is calculated as:TF

$$TF'_{u;w_i} = \sqrt{\frac{TF'_{f;w_i}\ {}_{i})^2}{\sum}}\ ; \qquad (2)$$

where *TF* ′ *f;wi* = 1 + ln *Nf;wi* . And in the search vector Q, *IDFwi* is calculated as:

$$IDF_{Wi} = \frac{IDF'_{W_i}}{\sqrt{\sum_{wi\ W_q} (IDF'_{W_i})^2}}\ ; \qquad (3)$$

where $IDF'_{Wi} = \ln(1 + N/N_{Wi})$.

Keyword Balanced Binary Tree. The balanced binary tree is widely used to deal with optimization problems [35], [36]. The keyword balanced binary (KBB) tree in our scheme is a dynamic data structure whose node stores a vector D. The elements of vector D are the normalized TF values. Sometimes, we refer the vector D in the node u to Du for simplicity. Formally, the node u in our KBB tree is defined as follows:

$$u = ID; D; Pl; Pr; FID ; \qquad (4)$$

where ID denotes the identity of node u, Pl and Pr are respectively the pointers to the left and right child of node u. If the node u is a leaf node of the tree, FID stores the identity of a document, and D denotes a vector consisting of the normalized TF values of the keywords to the document. If the node u is an internal node, FID is set to null, and D denotes a vector consisting of the TF values which is calculated as follows:

$$D[i] = max\{u{:}Pl \to D[i];\ u{:}Pr \to D[i]\};\ i = 1; :::; m: (5)$$

The detailed construction process of the tree-based index is illustrated in Section 4, which is denoted as Build IndexTree(F).

3.2 The System and Threat Models

The system model in this paper involves three different entities: data owner, data user and cloud server, as illustrated in Fig. 1.

Data owner has a collection of documents *F = {f1; f2; :::; fn}*that he wants to outsource to the cloud server in encrypted form while still keeping the capa-bility to search on them for effective utilization. In our scheme, the data owner firstly builds a secure searchable tree index I from document collection F, and then generates an encrypted document collection C for F. Afterwards, the data owner outsources the encrypted collection C and the secure index I to the cloud server, and securely distributes the key information of trapdoor generation (including keyword IDF values) and document decryption to the authorized data users. Besides, the data owner is responsible for the update operation of his documents stored in the cloud server. While updating, the data owner generates the update information locally and sends it to the server.

Data users are authorized ones to access the documents of data owner. With t query keywords, the authorized user can generate a trapdoor TD according to search control mechanisms to fetch k encrypted documents from cloud server. Then, the data user can decrypt the documents with the shared secret key.

Cloud server stores the encrypted document collection C and the encrypted searchable tree index I for data owner. Upon receiving the trapdoor TD from the data user, the cloud server executes search over the index tree I, and finally returns the corresponding collection of top-k ranked encrypted documents. Besides, upon receiving the update information from the data owner, the server needs to update the index I and document collection C according to the
received information.
The cloud server in the proposed scheme is considered as "honest-but-curious", which is employed by lots of works on secure cloud data search [25], [26], [27]. Specifically, the cloud server honestly and correctly executes encrypted search index tree request top-k ranked encrypted documents Semi-trusted result cloud server search control (trapdoors) access control (data decryption keys)

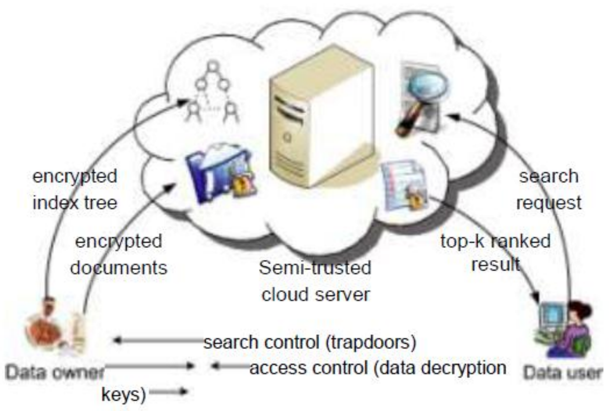

Fig. 1. The architecture of ranked search over encrypted cloud data

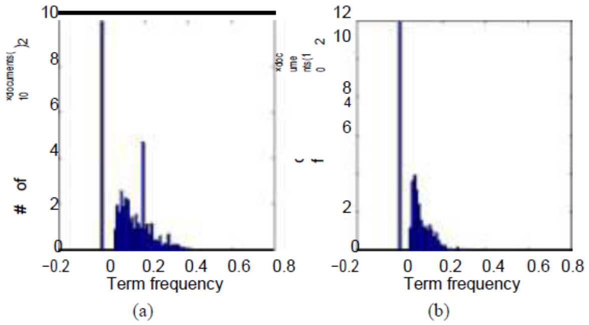

Fig. 2. Distribution of term frequency (TF) for (a) keyword "subnet", and (b) keyword "host".

instructions in the designated protocol. Meanwhile, it is curious to infer and analyze received data, which helps it acquire additional information. Depending on what information the cloud server knows, we adopt the two threat models proposed by Cao et al. [26].

Known Cipher text Model. In this model, the cloud server only knows the encrypted document collection C, the searchable index tree I, and the search trapdoor TD submitted by the authorized user. That is to say, the cloud server can conduct cipher text-only attack (COA) [37] in this model.

Known Background Model. Compared with known Cipher text model, the cloud server in this stronger model is equipped with more knowledge, such as the term frequency (TF) statistics of the document collection. This statistical information records how many documents are there for each term frequency of a specific keyword in the whole document collection, as shown in Fig. 2, which could be used as the keyword identity. Equipped with such statistical information, the cloud server can conduct TF statistical attack to deduce or even identify certain keywords through analyzing histogram and value range of the corresponding frequency distributions [24], [25], [27].

3.3 Design Goals

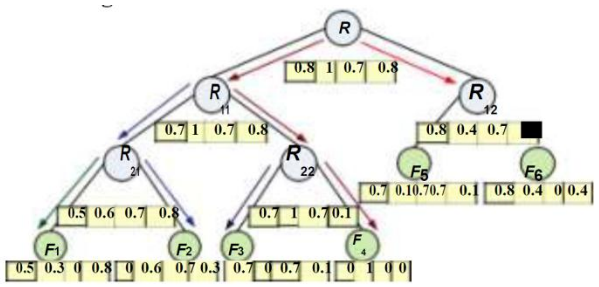

Fig. 3. An example of the tree-based index with the document collection F = ffiji = 1; :::; 6g and cardinality of the dictionary m = 4. In the construction process of the tree index, we first generate leaf nodes from the documents. Then, the internal tree nodes are generated based on the leaf nodes. This figure also shows an example of search process, in which the query vector Q is equal to (0; 0:92; 0; 0:38). In this example, we set the parameter k = 3 with the meaning that three documents will be returned to the user. According to the search algorithm, the search starts with the root node, and reaches the first leaf node f4 through r11 and r22. The relevance score of f4 to the query is 0:92. After that, the leaf nodes f3 and f2 are successively reached with the relevance scores 0:038 and 0:67. Next, the leaf node f1 is reached with score 0:58 and replace f3 in RList. Finally, the algorithm will try to search subtree rooted by r12, and find that there are no reasonable results in this subtree because the relevance score of r12 is 0:52, which is smaller than the smallest relevance score in RList

To enable secure, efficient, accurate and dynamic multi-keyword ranked search over outsourced encrypted cloud data under the above models, our system has the following design goals.

Dynamic: The proposed scheme is designed to pro-vide not only multi-keyword query and accurate result ranking, but also dynamic update on document collections.

Search Efficiency: The scheme aims to achieve sub linear search efficiency by exploring a special tree-based index and an efficient search algorithm.

Privacy-preserving: The scheme is designed to pre-vent the cloud server from learning additional information about the document collection, the index tree, and the query. The specific privacy requirements are summarized as follows,

1) Index Confidentiality and Query Confidentiality: The underlying plaintext information, including keywords in the index and query, TF values of key-words stored in the index, and IDF values of query keywords, should be protected from cloud server;

2) Trapdoor Unlinkability: The cloud server should not be able to determine whether two encrypted queries (trapdoors) are generated from the same search request;

3) Keyword Privacy: The cloud server could not Identify the specific keyword in query, index or document collection by analyzing the statistical information like term frequency. Note that our proposed scheme is not designed to protect access pattern, i.e., the sequence of returned documents.

IV. THE PROPOSED SCHEMES

In this section, we firstly describe the unencrypted dynamic multi-keyword ranked search (UDMRS) scheme which is constructed on the basis of vector space model and KBB tree. Based on the UDMRS scheme, two secure search schemes (BDMRS and EDMRS schemes) are con-structed against two threat models, respectively.

4.1 Index Construction of UDMRS Scheme

In Section 3, we have briefly introduced the KBB index tree structure, which assists us in introducing the index construction. In the process of index construction, we first generate a tree node for each document in the collection. These nodes are the leaf nodes of the index tree. Then, the internal tree nodes are generated based on these leaf nodes. The formal construction process of the index is presented in Algorithm 1. An example of our index tree is shown in Fig. 3. Note that the index tree T built here is a plaintext.

Following are some notations for Algorithm 1. Be-sides, the data structure of the tree node is defined as *ID; D; Pl; Pr; FID* , where the unique identity *ID* for each tree node is generated through the function GenID().

• *CurrentN odeSet* – The set of current processing nodes which have no parents. If the number of nodes is even, the cardinality of the set is denoted as $2h(h \in Z +)$, else the cardinality is denoted as $(2h + 1)$.

• *T empN odeSet* – The set of the newly generated nodes. In the index, if $Du[i] = 0$ for an internal node *u*, there is at least one path from the node *u* to some leaf, which indicates a document containing the keyword *wi*. In addition, $Du[i]$ always stores the biggest normalized TF value of *wi* among its child nodes. Thus, the possible largest relevance score of its children can be easily estimated.

4.2 Search Process of UDMRS Scheme

The search process of the UDMRS scheme is a recursive procedure upon the tree, named as "Greedy Depth-first Search (GDFS)" algorithm. We construct a result list denoted as RList, whose element is defined as RScore; FID . Here, the RScore is the relevance score of the document fFID to the query, which is calculated according to Formula (1). The RList stores the k accessed documents with the largest relevance scores to the query. The elements of the list are ranked in descending order according to the RScore, and will be updated timely during the search process. Following are some other notations, and the GDFS algorithm is described in Al-gorithm 2.

• RScore(*Du; Q*) – The function to calculate the relevance score for query vector *Q* and index vector *Du* stored in node *u*, which is defined in Formula (1).

• *k th score* – The smallest relevance score in current *RList*, which is initialized as 0.

• *hchild* – The child node of a tree node with higher relevance score.

• *lchild* – The child node of a tree node with lower relevance score.

Since the possible largest relevance score of documents rooted by the node u can be predicted, only a part of the nodes in the tree are accessed during the search process. Fig. 3

shows an example of search process with the document collection $F = \{fi|i = 1; :::; 6\}$, cardinality of the dictionary $m = 4$, and query vector • $Q = (0; 0{:}92; 0; 0{:}38)$

Algorithm 1 BuildIndexTree(F)

Input: the document collection $F = \{f1; f2; :::; fn\}$ with the identifiers $FID = \{FID|FID = 1; 2; :::; n\}$.
Output: the index tree *T*

```
for each document fFID in F do
    Construct a leaf node u for fFID, with $u{:}ID$ = GenID(), $u{:}Pl = u{:}Pr = null$, $u{:}FID = FID$, and $D[i] = TF_{fFID;wi}$ for $i = 1; :::; m$;—
    Insert u to CurrentN odeSet;
end for
while the number of nodes in CurrentN odeSet is larger than 1 do
    if the number of nodes in CurrentN odeSet is even, i.e. 2h then
        for each pair of nodes u and u in CurrentN odeSet do
            Generate a parent node u for u and u , with $u{:}ID$ = GenID(), $u{:}Pl = u$, $u{:}Pr = u$, $u{:}FID = 0$ and $D[i] = max\{u :D[i]; u :D[i]\}$ for each $i = 1; :::; m$;
            Insert u to T empN odeSet;
        end for
    else
        for each pair of nodes $u$ and $u$ of the former $(2h - 2)$ nodes in $CurrentN odeSet$ do
            Generate a parent node $u$ for $u$ and $u$ ;
            Insert $u$ to $T empN odeSet$;
        end for
        Create a parent node $u1$ for the $(2h - 1)$-th and $2h$-th node, and then create a parent node $u$ for $u1$ and the $(2h + 1)$-th node;
        Insert $u$ to $T empN odeSet$;
    end if
    Replace $CurrentN odeSet$ with $T empN odeSet$ and then clear $T empN odeSet$;
end while
return the only node left in $CurrentN odeSet$, namely,
the root of index tree $T$ ;
```

Algorithm 2 GDFS(IndexTreeNode *u*)

```
if the node $u$ is not a leaf node then
    if RScore($Du; Q$) > $k$ th score then
        GDFS($u:hchild$);
        GDFS($u:lchild$);
    else
        return
    end if
else
    if RScore($Du; Q$) > $k$ th score then
        Delete the element with the smallest relevance score from $RList$;
        Insert a new element RScore($Du; Q$); $u:FID$ and sort all the elements of $RList$;
    end if
    return
end if
```

V. PERFORMANCE ANALYSIS

We implement the proposed scheme using C++ language in Windows 7 operation system and test its efficiency on a real-world document collection: the Request for Comments (RFC) [39]. The test includes 1) the search precision on different privacy level, and 2) the efficiency of index construction, trapdoor generation, search, and update. Most of the experimental results are obtained with an Intel Core(TM) Duo Processor (2.93 GHz), except that the efficiency of search is tested on a server with two Intel(R) Xeon(R) CPU E5-2620 Processors (2.0 GHz), which has 12 processor cores and supports 24 parallel threads.

5.1 Precision and Privacy

The search precision of scheme is affected by the dummy keywords in EDMRS scheme. Here, the ’precision’ is defined as that in [26]: Pk = k =k, where k is the number of real top-k documents in the retrieved k documents. If a smaller standard deviation is set for the random

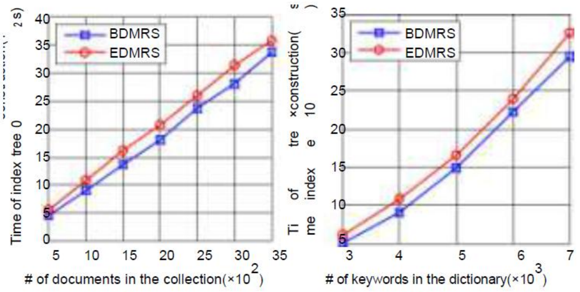

Fig. 5. Time cost for index tree construction: (a) for the different sizes of document collection with the fixed dictionary, m = 4000, and (b) for the different sizes of dictionary with the fixed document collection, n = 1000.

TABLE 3
Storage consumption of index tree.

Size of dictionary	1000	2000	3000	4000	5000
BDMRS (MB)	73	146	220	293	367
EDMRS (MB)	95	168	241	315	388

Σ

variable "v, the EDMRS scheme is supposed to obtain higher precision, and vice versa. The results are shown in Fig. 4(a).

In the EDMRS scheme, phantom terms are added to the index vector to obscure the relevance score calculation, so that the cloud server cannot identify keywords by analyzing the TF distributions of special keywords. Here, we quantify the obscureness of the relevance score by “rank privacy”, which is defined as:

$$P_k = \sum \left| r_i' - r_i \right| = k^2; \qquad (8)$$

where *ri* is the rank number of document in the retrieved top-*k* documents, and *ri* is its real rank number in the whole ranked results. The larger rank privacy denotes the higher security of the scheme, which is illustrated in Fig. 4(b).

In the proposed scheme, data users can accomplish different requirements on search precision and privacy by adjusting the standard deviation , which can be treated as a balance parameter.

We compare our schemes with a recent work proposed by Sun et al. [27], which achieves high search efficiency. Note that our BDMRS scheme retrieves the search results through exact calculation of document vector and query vector. Thus, top-k search precision of the BDMRS scheme is 100%. But as a similarity-based

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 4, Issue 6, June 2017

multi-keyword ranked search scheme, the basic scheme in [27] suffers from precision loss due to the clustering of sub-vectors during index construction. The precision test of [27]'s basic scheme is presented in Table 2. In each test, 5 keywords are randomly chosen as input, and the precision of returned top 100 results is observed. The test is repeated 16 times, and the average precision is 91%.

5.2 Efficiency

5.2.1 Index Tree Construction

The process of index tree construction for document collection *F* includes two main steps: 1) building an unencrypted KBB tree based on the document collection *F*, and 2) encrypting the index tree with splitting operation and two multiplications of a ($m \times m$) matrix. The index structure is constructed following a post order traversal of the tree based on the document collection *F*, and *O*(*n*) nodes are generated during the traversal. For each node, generation of an index vector takes *O*(*m*) time, vector splitting process takes *O*(*m*) time, and two multiplications of a ($m \times m$) matrix takes *O*(*m* **2**) time. As a whole, the time complexity for index tree construction is *O*(*nm* **2**). Apparently, the time cost for building index tree mainly depends on the cardinality of document collection *F* and the number of keywords in dictionary *W*. Fig. 5 shows that the time cost of index tree construction is almost linear with the size of document collection, and is proportional to the number of keywords in the dictionary. Due to the dimension extension, the index tree construction of EDMRS scheme is slightly more time-consuming than that of BDMRS scheme. Although the index tree construction consumes relatively much time at the data owner side, it is noteworthy that this is a one-time operation. On the other hand, since the underlying balanced binary tree has space complexity *O*(*n*) and every node stores two *m*-dimensional vectors, the space complexity of the index tree is *O*(*nm*). As listed in Table 3, when the document collection is fixed (n = 1000), the storage consumption of the index tree is determined by the size of the dictionary.

5.2.2 Trapdoor Generation

The generation of a trapdoor incurs a vector splitting operation and two multiplications of a ($m \times m$) matrix, thus the time complexity is $O(m^2)$, as shown in Fig. 6(a). Typical search requests usually consist of just a few keywords. Fig. 6(b) shows that the number of query keywords has little influence on the overhead of trapdoor generation when the dictionary size is fixed. Due to the dimension extension, the time cost of EDMRS scheme is a little higher than that of BDMRS scheme.

5.2.3 Search Efficiency

During the search process, if the relevance score at node *u* is larger than the minimum relevance score in result list *RList*, the cloud server examines the children of the node; else it returns. Thus, lots of nodes are not accessed during a real search. We denote the number of leaf nodes that contain one or more keywords in the query as . Generally, is larger than the number of required documents *k*, but far less than the cardinality of the document collection *n*. As a balanced binary tree, the height of the index is maintained to be log *n*, and the complexity of relevance score calculation is *O*(*m*). Thus,

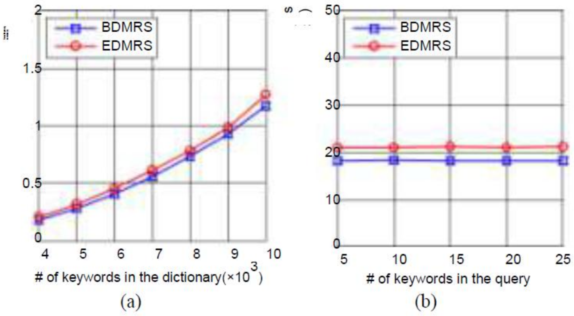

Fig. 6. Time cost for trapdoor generation: (a) for different sizes of dictionary with the fixed number of query keywords, t = 10, and (b) for different numbers of query keywords with the fixed dictionary, m = 4000.

the time complexity of search is *O*(*m* log *n*). Note that the real search time is less than *m* log *n*. It is because 1) many leaf nodes that contain the queried keywords are not visited according to our search algorithm, and 2) the accessing paths of some different leaf nodes share the mutual traversed parts. In addition, the parallel execution of search process can increase the efficiency a lot. We test the search efficiency of the proposed scheme on a server which supports 24 parallel threads. The search performance is tested respectively by starting 1, 4, 8 and 16 threads. We compare the search efficiency of our scheme with that of Sun *et al.* [27]. In the implementation of Sun's code, we divide 4000 keywords into 50 levels. Thus, each level contains 80 keywords. According to [27], the higher level the query keywords reside, the higher the search efficiency is. In our experiment, we choose ten keywords from the 1st level (the highest level, the optimal case) for search efficiency comparison. Fig. 7 shows that if the query keywords are chosen from the 1st level, our scheme obtains almost the same efficiency as [27] when we start 4 threads. Fig. 7 also shows that the search efficiency of our scheme increases a lot when we increase the number of threads from 1 to 4. However, when we continue to increase the threads, the search efficiency is not increased remarkably. Our search algorithm can be executed in parallel to improve the search efficiency. But all the start-ed threads will share one result list *RList* in mutually exclusive manner. When we start too many threads, the threads will spend a lot of time for waiting to read and write the *RList*.

An intuitive method to handle this problem is to construct multiple result lists. However, in our scheme, it will not help to improve the search efficiency a lot. It is because that we need to find *k* results for each result list and time complexity for retrieving each result list is *O*(*m* log *n*=*l*). In this case, the multiple threads will not save much time, and selecting *k* results from the multiple result list will further increase the time consumption. In the Fig. 8, we show the time consumption when we start multiple threads with multiple result lists. The experimental results prove that our scheme will obtain better search efficiency when we start multiple threads with only one result list.

5.2.4 Update Efficiency

In order to update a leaf node, the data owner needs to update log n nodes. Since it involves an encryption operation for index vector at each node, which takes O(m 2) time, the time complexity of update operation is thus O(m 2 log n). We illustrate the time cost for the deletion of a document. Fig. 9(a) shows that when the size of dictionary is fixed, the deletion of a document takes nearly logarithmic time with the size of document collection. And Fig. 9(b) shows that the update time is proportional to the size of dictionary when the document collection is fixed. In addition, the space complexity of each node is O(m). Thus, space complexity of the communication package of updating a document is O(m log n).

6 CONCLUSION AND FUTURE WORK

In this paper, a secure, efficient and dynamic search scheme is proposed, which supports not only the accurate multi-keyword ranked search but also the dynamic deletion and insertion of documents. We construct a special keyword balanced binary tree as the index, and propose a “Greedy Depth-first Search” algorithm to obtain better efficiency than linear search. In addition, the parallel search process can be carried out to further reduce the time cost. The security of the scheme is protected against two threat models by using the secure kNN algorithm. Experimental results demonstrate the efficiency of our proposed scheme. There are still many challenge problems in symmetric SE schemes. In the proposed scheme, the data owner is responsible for generating updating information and sending them to the cloud server. Thus, the data owner needs to store the unencrypted index tree and the information that are necessary to recalculate the IDF values. Such an active data owner may not be very suitable for the cloud computing model. It could be a meaningful but difficult future work to design a dynamic searchable encryption scheme whose updating operation can be completed by cloud server only, meanwhile reserving the ability to support multi-keyword ranked search. In addition, as the most of works about searchable encryption, our scheme mainly considers the challenge from the cloud server. Actually, there are many secure challenges in a multi-user scheme. Firstly, all the users usually keep the same secure key for trapdoor generation in a symmetric SE scheme. In this case, the revocation of the user is big challenge. If it is needed to revoke a user in this scheme, we need to rebuild the index and distribute the new secure keys to all the authorized users. Secondly, symmetric SE schemes usually assume that all the data users are trustworthy. It is not practical and a dishonest data user will lead to many secure problems. For exam-ple, a dishonest data user may search the documents and distribute the decrypted documents to the unauthorized ones. Even more, a dishonest data user may distribute his/her secure keys to the unauthorized ones. In the future works, we will try to improve the SE scheme to handle these challenge problems.

REFERENCES

[1] K. Ren, C. Wang, Q. Wang et al., “Security challenges for the public cloud,” IEEE Internet Computing, vol. 16, no. 1, pp. 69–73, 2012.

[2] S. Kamara and K. Lauter, “Cryptographic cloud storage,” in Financial Cryptography and Data Security. Springer, 2010, pp. 136– 149.

[3] C. Gentry, “A fully homomorphic encryption scheme,” Ph.D. dissertation, Stanford University, 2009.

[4] O. Goldreich and R. Ostrovsky, “Software protection and simula-tion on oblivious rams,” Journal of the ACM (JACM), vol. 43, no. 3, pp. 431–473, 1996.

[5] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, “Public key encryption with keyword search,” in Advances in Cryptology-Eurocrypt 2004. Springer, 2004, pp. 506–522.

[6] D. Boneh, E. Kushilevitz, R. Ostrovsky, and W. E. Skeith III, “Public key encryption that allows pir queries,” in Advances in Cryptology-CRYPTO 2007. Springer, 2007, pp. 50–67.

[7] D. X. Song, D. Wagner, and A. Perrig, “Practical techniques for searches on encrypted data,” in Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on. IEEE, 2000, pp. 44– 55.

[8] E.-J. Goh et al., “Secure indexes.” IACR Cryptology ePrint Archive, vol. 2003, p. 216, 2003.

[9] Y.-C. Chang and M. Mitzenmacher, “Privacy preserving keyword searches on remote encrypted data,” in Proceedings of the Third

in-ternational conference on Applied Cryptography and Network Security. Springer-Verlag, 2005, pp. 442–455.

[10] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, “Searchable symmetric encryption: improved definitions and efficient con-structions,” in Proceedings of the 13th ACM conference on Computer and communications security. ACM, 2006, pp. 79–8