Hindawi
Security and Communication Networks
Volume 2020, Article ID 7298518, 16 pages
https://doi.org/10.1155/2020/7298518

Research Article

Efficient Searchable Symmetric Encryption Supporting Dynamic Multikeyword Ranked Search

Yu Zhang,[1] **Yin Li**,[2] **and Yifan Wang**[3]

[1]*School of Computer and Information Technology, Xinyang Normal University, Xinyang 464000, China*
[2]*School of Cyberspace Security, Dongguan University of Technology, Dongguan, China*
[3]*Wayne State University, 42 W. Warren Ave., Detroit, MI 48202, USA*

Correspondence should be addressed to Yu Zhang; willow1223@126.com

Received 9 January 2020; Revised 10 June 2020; Accepted 24 June 2020; Published 16 July 2020

Academic Editor: Stelvio Cimato

Searchable symmetric encryption that supports dynamic multikeyword ranked search (SSE-DMKRS) has been intensively studied during recent years. Such a scheme allows data users to dynamically update documents and retrieve the most wanted documents efficiently. Previous schemes suffer from high computational costs since the time and space complexities of these schemes are linear with the size of the dictionary generated from the dataset. In this paper, by utilizing a shallow neural network model called "Word2vec" together with a balanced binary tree structure, we propose a highly efficient SSE-DMKRS scheme. The "Word2vec" tool can effectively convert the documents and queries into a group of vectors whose dimensions are much smaller than the size of the dictionary. As a result, we can significantly reduce the related space and time cost. Moreover, with the use of the tree-based index, our scheme can achieve a sublinear search time and support dynamic operations like insertion and deletion. Both theoretical and experimental analyses demonstrate that the efficiency of our scheme surpasses any other schemes of the same kind, so that it has a wide application prospect in the real world.

1. Introduction

Nowadays, with the development of the network and virtualization technology, cloud computing technology has been developed rapidly. Through the cloud service, enterprises and individuals can obtain better computing and storage services at a lower cost. Since cloud servers are not entirely trusted, utilizing cloud services while maintaining data privacy is an essential concern. A straightforward way to address this issue is encrypting the data before outsourcing it to the cloud servers. However, this approach fails to meet the requirement of data retrieval since traditional encryption will scramble the original data, making the data inconvenient to utilize. In this scenario, the users have to download all the ciphertext data and decrypt them locally, which will bring huge transmission, storage, and computation overhead, which is not applicable in cloud environment.

Searchable encryption (SE) can support keyword search without decrypting the data, and thus it is very suitable to achieve the keyword search over ciphertext. Based on the SE scheme, data owners and authorized users share a secret key. Data owners can encrypt the sensitive data and upload them to the cloud server. If data users want to search the encrypted data, they can generate an encrypted trapdoor by using the query and the secret key. When the cloud server receives the trapdoor, it tests the trapdoor against the encrypted data without decrypting these data and returns the data related to the query to the users. The first searchable symmetric encryption (SSE) scheme was proposed by Song et al. [1]. This scheme can not only encrypt data, but also provide a search mechanism over the encrypted data. With the improvement of security and efficiency of SSE schemes, it has attracted the community attention. During recent years, researchers focused on how to construct solutions with complex query functions, such as multikeyword search [2–7], similarity search [8, 9], and ranked search [10–15]. In particular, ranked search schemes can sort the query results according to the relevant degree between the documents and queries

and only return the most related (top-k) documents. Thus, ranked schemes can significantly reduce the computation and storage costs.

The primary ranked search schemes were proposed in [10, 11], which only support a single-keyword search. The early SSE scheme supporting multikeyword ranked search was given by Cao et al. [12]. The score evaluation method used in their scheme is the inner product between the query and document vectors. In their scheme, since each document has its own vector representation, the search time is linear with the number of documents in the dataset, which will have a very high storage overhead for a big data environment. Then, Sun et al. [13] gave a similar scheme with a better-than-linear search efficiency by using a tree-based index [16, 17]. They adopt the technique of term frequency-inverse document frequency (TF-IDF) to evaluate the score between the index and queries. To further improve the search efficiency and support dynamic update, Xia et al. proposed an efficient SSE scheme supporting dynamic multikeyword ranked search [14]. In their scheme, they construct a tree-based structure and propose a parallel search algorithm to accelerate the search process. Moreover, they also provide a dynamic update method to cope with the deletion and insertion of documents flexibly. Recently, by utilizing the Bloom filter [18], Guo et al. constructed an efficient SSE scheme supporting dynamic multikeyword ranked search [15] to further improve the efficiency of keywords search and index construction. Owing to the Bloom filter, the internal nodes in the index tree are not needed to be encrypted, and the dimension of the vectors in the internal nodes is also reduced. As a result, this scheme can achieve a better performance than the previous similar schemes.

Another kind of SE is called searchable public key encryption (SPE), which is established on the public key system. In SSE, the key for encrypting data is the same as the key for generating search trapdoor. By contrast, in SPE, the public key for encrypting data is open to public, while the secret key for generating search trapdoor is only given to the authorized data receivers. The very first SPE scheme supporting keyword search was introduced by Boneh et al., and it is so-called public key with keyword search (PEKS) [19]. However, their work only supports a single-keyword search. In order to support more expressive query, many SPE schemes [20–23] were proposed to realize advanced search, for example, conjunctive, disjunctive, and Boolean keyword search. By using a special hidden structure, Xu et al. proposed two SPE schemes supporting single-keyword search [24, 25] whose search performance is very close to that of a practical SSE scheme. By converting an attribute-based encryption scheme, Han et al. proposed an SPE scheme which can control user's search permission according to an access control policy [26]. After this, Kai et al. proposed an SPE scheme achieving both Boolean keyword search and fine-grained search permission [27]. Sepehri et al. proposed a scalable proxy-based protocol for privacy-preserving queries, which allows authorized users to perform queries over data encrypted with different keys [28]. Later, by utilizing an El-Gamal elliptic curve encryption system, Sepehri et al. gave a similar scheme with better efficiency [29]. In order to improve search accuracy, Zhang et al. proposed an SPE scheme supporting semantic keywords search by adopting a method called "Word2vec" [30]. For the sake of brevity, we summarize some SPE and SSE schemes in Table 1, which describes the difference between our scheme and previous schemes.

1.1. Motivation. The previous ranked search schemes in symmetric key setting are secure and somewhat efficient. However, the index building, trapdoor generation, and search time are all related to the size of the dictionary generated from the dataset, which is not suitable for the big data environment. According to the statistical information given in [20], we found that the vocabulary size in a dataset is commonly linear with $O(10^6)$. Therefore, it is necessary to construct a more efficient ranked search scheme. Motivated by this, in this paper, we aim to construct a novel SSE scheme supporting dynamic multikeyword ranked search (SSE-DMKRS) with high efficiency.

1.2. Contributions. The main contributions are summarized as follows:

(1) Based on "Word2vec" [31] technique, we propose a novel method which can change the documents and queries into vector representations. The dimension of the vector representation obtained by our method is nearly 10% of that in the previous SSE-DMKRS schemes [14, 15].

(2) We propose an efficient index building algorithm which can create a balanced binary tree to index all the documents. The obtained index tree can achieve a sublinear search time and support dynamic update operations.

(3) Through applying the secure k-nearest neighbour (KNN) scheme [32] to encrypt the index tree and the query, we propose an efficient SSE-DMKRS scheme.

In addition, we implement our scheme on a widely used data collection. The experiment results show that our scheme extremely reduces the time cost of index building, trapdoor generation, keywords search, and update without losing too much accuracy; e.g., the time cost of index building in our scheme is nearly 10% of that in the previous schemes. Meanwhile, the storage cost of encrypted index is also reduced greatly; e.g., the storage cost of the index in our scheme is nearly one percent of that in the previous schemes. In conclusion, compared to the previous SSE-DMKRS schemes [14, 15], our scheme is very suitable for the mobile cloud environment in which the client device has limited computation and storage resources.

1.3. Organization. This paper is organized as follows. In Section 2, we give a formal definition of the system model and threat model in our scheme and also introduce the tools we adopt in our scheme, which contains "Word2vec" and the vector space model. In Section 3, we present the

Table 1: Comparison between previous SE schemes and ours.

Type	Ref.	Query condition	Additional special abilities
SSE	[6]	Conjunctive keyword search	—
	[8]	Multikeyword fuzzy search	—
	[11]	Single-keyword ranked search	—
	[12]	Multikeyword ranked search	—
	[14]	Multikeyword ranked search	Dynamic update
	Ours	Multikeyword ranked search	Semantic search and dynamic update
SPE	[22]	Conjunctive and disjunctive keyword search	—
	[23]	Boolean keyword search	—
	[25]	Single-keyword search	Fast search
	[27]	Boolean keyword search	Access control
	[29]	Multikeyword search	Data sharing
	[30]	Multikeyword search	Semantic search

construction of the search index tree and the SSE-DMKRS scheme. Besides, a detailed security analysis and update operations of our scheme are also given. Theoretical and experimental analyses are given in Section 4. Section 5 gives the conclusion.

2. Preliminaries

In this section, we first give the framework of the system model and introduce the threat model adopted in our scheme. Then, we introduce some tools adopted in our schemes, including a famous term representation method in the field of natural language processing, e.g., "Word2vec," and the vector space model. Finally, we present the design goal of our scheme. In addition, the main notations used in this paper are summarized in Table 2.

2.1. System Model. The system model contains three different roles: data owner, data user, and cloud server. The data owner outsources a group of documents $F = \{f_1, f_2, \ldots, f_n\}$ to the cloud in ciphertext form $C = \{c_1, c_2, \ldots, c_n\}$. Moreover, the data owner also generates an encrypted searchable index for keywords search operation. For each query of an arbitrary keyword set Q, the data user computes a search trapdoor T_Q of the query Q and sends it to the cloud server. Upon receiving T_Q from the data user, the cloud server searches against the encrypted index and returns the candidate encrypted documents. After this, the data user decrypts the candidate documents and obtains the plaintext.

As illustrated in Figure 1, the architecture of the system model is formally described as follows:

(1) *Data Owner (DO).* DO holds a group of documents $F = \{f_1, f_2, \ldots, f_n\}$ and generates a secure searchable index I from F and an encrypted document collection C for F. Then, DO uploads I and C to the cloud server and distributes the secret key to the authorized data users. Furthermore, DO needs to update the index and documents stored in the cloud server.

(2) *Data User (DU).* Authorized DU can launch keywords query over the encrypted data by utilizing a trapdoor which is generated by using the secret key fetched from DO. Moreover, DU can decrypt the encrypted documents by utilizing the secret key.

(3) *Cloud Server (CS).* CS stores the encrypted index I and documents C from DO. When CS receives the trapdoor for query Q from DU, CS executes keywords query over the index and returns the top-k most relevant encrypted documents associated with the query Q. Upon receiving the update information from DO, CS also performs update operation over the encrypted data. In addition, we assume that CS is "honest-but-curious," which is employed by many searchable encryption schemes [12, 14, 15]. This means that CS honestly and correctly executes the algorithms in our scheme. However, CS curiously infers and analyses the received data to obtain extra privacy information.

2.2. Threat Model. Throughout the paper, we mainly utilize two threat models proposed by Cao et al. [12]:

(1) *Known Ciphertext Model.* CS only knows the information of the encrypted index, ciphertext, and trapdoor. That is to say, CS can execute cipher-only attacks in this model.

(2) *Known Background Model.* CS knows more information than the known ciphertext model, such as the statistical information inferred from the documents. By taking advantage of these pieces of statistical information, e.g., term frequency (TF) and inverse document frequency (IDF), CS can conduct statistical attack to verify whether certain keywords are in the query [33].

2.3. Design Goals. As mentioned before, we aim to build a secure and efficient SSE-DMKRS scheme. The design goal of our scheme is described as follows:

(1) *Efficiency.* The scheme aims to realize a sublinear search efficiency, and the time and space costs of index building and trapdoor generation are much less than those of the current schemes.

Table 2: Notations.

F	A document set $\{f_1, f_2, \ldots, f_n\}$.
n	The number of documents in F.
C	The encrypted form of F, denoted by $\{c_1, c_2, \ldots, c_n\}$.
W_i	The keyword set $\{w_{i1}, w_{i2}, \ldots, w_{it_i}\}$ for the document f_i.
t_i	The number of keywords in W_i, and $i \in [1, n]$.
w_{ij}	The jth keywords in W_i, and $i \in [1, n]$, $j \in [1, t_i]$.
$\overrightarrow{W_i}$	The vector representation for W_i.
$\overrightarrow{w_{ij}}$	The vector representation for w_{ij}.
u	A node in the index tree.
$\overrightarrow{u}$	The vector representation for the node u.
$\overrightarrow{u_{\min}}$, $\overrightarrow{u_{\max}}$	Vector representations are obtained by splitting $\overrightarrow{u}$.
I_u	The encrypted index for the node u.
I_T	The encrypted index tree of F.
Q	The keyword set, $\{q_1, q_2, \ldots, q_t\}$, for query.
q_j	A keyword in Q, $j \in [1, t]$.
$\overrightarrow{q}$	The vector representation for query Q.
$\overrightarrow{q_{\min}}$, $\overrightarrow{q_{\max}}$	Vector representations are obtained by splitting $\overrightarrow{q}$.
$\overrightarrow{q_j}$	The vector representation for q_j.
T_Q	The trapdoor of Q.
D	A dictionary, $\{w_1, w_2, \ldots, w_m\}$, containing all keywords in F.
M_{11}, M_{12}, M_{21}, M_{22}	Matrices for encryption (encryption key).
M_{11}^{-1}, M_{12}^{-1}, M_{21}^{-1}, M_{22}^{-1}	Matrices for decryption (decryption key).
N	The number of semantic keywords associated with each dictionary's keyword.
m	The number of keywords in D.
d	The dimension of vector generated by using "Word2vec."
k	The number of files returned to the user.

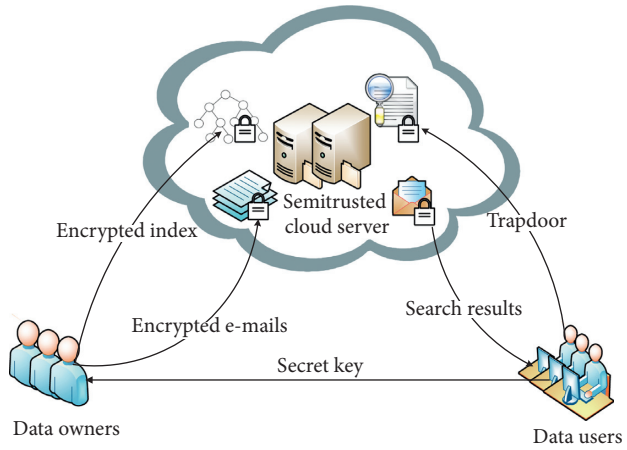

Figure 1: System model of the keywords search over encrypted data.

(2) *Privacy Preserving*. Similar with previous schemes [12, 14, 15], our scheme needs to prevent CS from learning extra privacy information, which is inferred from the documents, secure index, and queries. More precisely, the privacy requirement is listed as follows:

- *Index and Trapdoor Privacy*. The plaintext information concealed in the index and the trapdoor cannot be leaked to CS. This information involves the keywords and the corresponding vector representation of each keyword.
- *Trapdoor Unlinkability*. CS cannot determine whether two trapdoors are built from the same query.
- *Keyword Privacy*. CS cannot identify whether a specific keyword is in the trapdoor or index by analysing the search results and the statistical information of the documents.

(3) *Dynamic*. The scheme can efficiently support dynamic operations like documents insertion and deletion. Note that the efficiency of update operations in our scheme is better than the previous SSE-DMKRS schemes.

2.4. Word2Vec. "Word2Vec" model is a shallow, two-layer neural network, which is used to convert words into a group of vector representations [31]. Under this model, each word in the document set is mapped to a vector, which can be used to calculate the similarity between words. For instance, Figure 2 shows that, through training a simple corpus, three words "dog," "fox," and "orange" are mapped to three vector representations, respectively. By utilizing these vectors, the similarity among these three words can be calculated. We can find that the similarity between "dog" and "fox" is more than that between "dog" and "orange" since "dog" and "fox" are animals. Thus, we can utilize "Word2Vec" to convert the keywords in a corpus into a group of vector representations and then apply these vectors to perform ranked search.

2.5. Advice on Equations. Vector space model is a very popular method used in the field of information retrieval, usually along with the TF-IDF rule to realize the top-k search, where TF is term frequency and IDF is the inverse document frequency [34]. By utilizing the TF-IDF rule, the

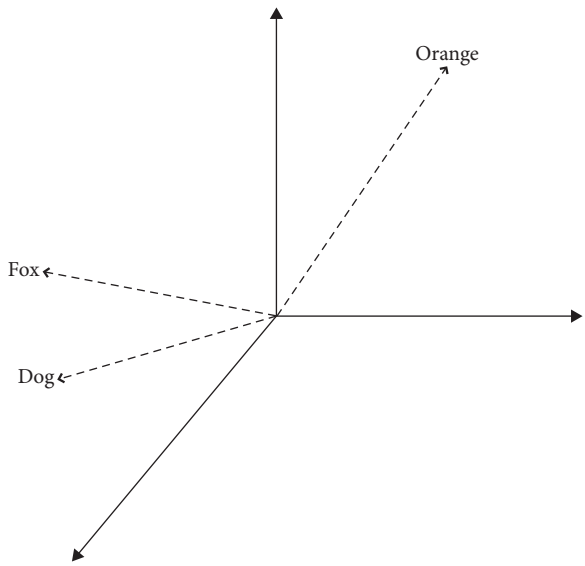

Figure 2: A vector space representation of words shows that "dog" is closer to "fox" since they share more common attributes than "dog" and "orange."

documents and queries can be represented as a group of vectors. These vectors can be adopted in the top-k search over the ciphertext [12, 14, 15]. However, the dimension of these obtained vectors is linear with the number of words in the dataset, which is not efficient if the dataset has a lot of words. To address this issue, we will apply "Word2Vec" to present a novel keywords conversion method, which is described as follows:

(1) Through applying "Word2Vec" to a corpus, we create a dictionary in which each keyword is associated with a vector representation.

(2) For the keyword set $W_i = \{w_{i1}, w_{i2}, \ldots, w_{it_i}\}$ of the document f_i, we obtain a vector $\overrightarrow{x_i} = \overrightarrow{w_{i1}} + \overrightarrow{w_{i2}} + \cdots + \overrightarrow{w_{it_i}}$ by looking up the dictionary, where $\overrightarrow{w_{ij}}$ is a vector representation for w_{ij} and $i \in [1, n]$, $j \in [1, t_i]$. After this, we set $\overrightarrow{W_i} = \overrightarrow{x_i}/\|\overrightarrow{x_i}\|$ as the vector representation of W_i.

(3) For the query keyword set $Q = \{q_1, q_2, \ldots, q_t\}$, we utilize the dictionary to construct a vector $\overrightarrow{v} = \overrightarrow{q_1} + \overrightarrow{q_2} + \cdots + \overrightarrow{q_t}$. Then, we set $\overrightarrow{Q} = \overrightarrow{v}/\|\overrightarrow{v}\|$ as the vector representation of Q.

Note that the dimensions of W_i and Q are very small, e.g., 200, which is significantly smaller than the number of words in the dataset. Thus, the proposed method is better than the previous method based on the TF-IDF rule. In addition, together with the vector space model mentioned above, we use the cosine measure to evaluate the relevance between the document and the query. The relevant evaluation function is defined in the next section.

3. Proposed Scheme

In this section, we first give the algorithms of the index tree building and the search algorithm on this tree. Then, we give the concrete construction of our scheme and the dynamic update operations of our scheme. Finally, we give a detailed analysis of the security of our scheme.

3.1. Search Index Balanced Binary Tree. In this section, we adopt a balanced binary tree to create the search index, which will be used in our main scheme. Inspired by the construction process in [14], the tree building and the search process for our scheme are described as follows.

3.1.1. Tree Building Process. Formally, the data structure of the tree node u is defined as $u \leq \mathrm{ID}, \overrightarrow{u_{\min}}, \overrightarrow{u_{\max}}, P_l, P_r, \mathrm{FID} >$, where ID is the identity of the node u, $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$ are the vector representations of the node u, P_l and P_r are pointers which point u's left and right children, respectively, and FID stores the identity of a document if u is a leaf node. Note that, compared with the previous index trees [12, 14, 15], the node in our tree has two vectors while it has only one vector in previous trees. The main reason is that the node vector in our tree has a negative number while the node vector in previous trees only contains positive number. For clarity, we give a simple example. Let $\overrightarrow{a} = \{0.1, 0.2, -0.3\}$ and $\overrightarrow{b} = \{-0.5, 0, 0.3\}$ be two vectors of leaf nodes A and B, respectively. For the previous index trees, the vector of the parent node C of these two leaf nodes is $\overrightarrow{c} = \{0.1, 0.2, 0.3\}$ in which the value of each dimension is the larger value of $\overrightarrow{a}$ and $\overrightarrow{b}$. For a query vector $\overrightarrow{v} = \{-1, 0, -1\}$, the scores of the nodes A, B, and C are 0.2, 0.2, and −0.4, respectively. It is very important to note that the score of the parent node is less than the scores of its children, which causes the fact that these two leaf nodes will be ignored in the tree search process even if they should be considered.

In our index tree, let the dimensions of $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$ be both d. The methods for constructing $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$ are denoted by M_1 and M_2, respectively, and given as follows:

(1) M_1: if the node u is a leaf node which is corresponding a file f, we create a vector $\overrightarrow{u}$ for f by adopting the keywords conversion method mentioned in Section 2.5. Then, we set $\overrightarrow{u_{\min}} = \overrightarrow{u}$ and $\overrightarrow{u_{\max}} = \overrightarrow{u}$

(2) M_2: if the node u is an internal node, the $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$ are based on its children vectors. Let $P_l \cdot \overrightarrow{u_{\min}}$ and $P_l \cdot \overrightarrow{u_{\max}}$ be the two vectors of u's left child, and let $P_r \cdot \overrightarrow{u_{\min}}$ and $P_r \cdot \overrightarrow{u_{\max}}$ be the two vectors of u's right child.

Suppose that Min () and Max () are the functions of the minimum and maximum, respectively; $\overrightarrow{u_{\min}}$ is built as follows:

$$\overrightarrow{u_{\min}}[i] = \mathrm{Min}\left(P_l \cdot \overrightarrow{u_{\min}}[i], P_r \cdot \overrightarrow{u_{\min}}[i]\right), \quad i \in [1, d]. \quad (1)$$

And, $\overrightarrow{u_{\max}}$ is built as follows:

$$\overrightarrow{u_{\max}}[i] = \mathrm{Max}\left(P_l \cdot \overrightarrow{u_{\max}}[i], P_r \cdot \overrightarrow{u_{\max}}[i]\right), \quad i \in [1, d]. \quad (2)$$

We find that $\overrightarrow{u_{\max}}$ is built by utilizing the larger number of $P_l \cdot \overrightarrow{u_{\max}}$ and $P_r \cdot \overrightarrow{u_{\max}}$, and $\overrightarrow{u_{\min}}$ is created by using the smaller number of $P_l \cdot \overrightarrow{u_{\min}}$ and $P_r \cdot \overrightarrow{u_{\min}}$.

An illustration of the above methods is given in Figure 3. From Figure 3, let the node u be a leaf node, and let W be the keyword set of the file that u stores. By using the keyword conversion method, W is converted to be a vector $\overrightarrow{u} = \{-0.2, 0.2, 0.5, -0.7, 0.8\}$. Then, we set $\overrightarrow{u_{\min}} = \overrightarrow{u_{\max}} = \overrightarrow{u}$ a. If the node u is an internal node, and the vectors of its children are $P_l \cdot \overrightarrow{u_{\min}}$, $P_l \cdot \overrightarrow{u_{\max}}$, $P_r \cdot \overrightarrow{u_{\min}}$, and $P_r \cdot \overrightarrow{u_{\max}}$ and the vectors of the internal node are $\overrightarrow{u_{\min}} = \{-0.2, -0.2, -0.3, -0.7, -0.5\}$ and $\overrightarrow{u_{\max}} = \{0.3, 0.7, 0.3, -0.1, 0.3\}$.

Based on the methods M_1 and M_2, inspired by the tree building algorithm introduced in [14], our tree building algorithm is given in Algorithm 1. An example of the proposed index tree is given in Example 1 and Figure 4. In Algorithm 1, we use function GenID () to generate the unique identity ID for each node, and apply GenFID () to generate the unique file ID for each leaf node. CurrentNodeSet contains a group of nodes having no parent node, which are needed to be processed. |CurrentNodeSet| is the number of nodes in CurrentNodeSet. If |CurrentNodeSet| is even, we assume that $|\text{CurrentNodeSet}| = 2h$; otherwise, we assume that $|\text{CurrentNodeSet}| = 2h + 1$, where h is a positive number. TempNodeSet is a set containing the newly generated nodes. Moreover, for each node u, if u is a leaf node, we use method M_1 to generate $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$; otherwise, $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$ are created by using M_2.

3.1.2. Search Process. For a query vector $\overrightarrow{q}$ of query Q, we spilt $\overrightarrow{q}$ into two vectors $\overrightarrow{q_{\min}}$ and $\overrightarrow{q_{\max}}$. For each dimension $i \in [1, d]$, if $\overrightarrow{q}[i] < 0$, $\overrightarrow{q_{\min}}[i] = \overrightarrow{q}[i]$ and $\overrightarrow{q_{\max}}[i] = 0$; otherwise, $\overrightarrow{q_{\min}}[i] = 0$ and $\overrightarrow{q_{\max}}[i] = \overrightarrow{q}[i]$. Obviously, $\overrightarrow{q_{\min}}$ holds all the negative part of $\overrightarrow{q}$, while $\overrightarrow{q_{\max}}$ holds the positive part. For clarity, we denote this splitting method for query Q by M_3. The illustration of this method is given in Figure 5. If the query vector $\overrightarrow{q} = \{0.1, -0.2, 0.3, -0.4, 0.5\}$, then $\overrightarrow{q_{\min}} = \{0, -0.2, 0, -0.4, 0\}$ and $\overrightarrow{q_{\max}} = \{0.1, 0, 0.3, 0, 0.5\}$.

For a query Q and a node u, the score is calculated as

$$\text{Score}(u, Q) = \overrightarrow{u_{\min}} \cdot \overrightarrow{q_{\min}} + \overrightarrow{u_{\max}} \cdot \overrightarrow{q_{\max}}. \tag{3}$$

We can utilize the above equation to evaluate which documents are the most related to the query. Moreover, we can verify that the score of the parent node is larger than its children's score. This property can significantly reduce the number of nodes which will be checked in the search process.

The search process is given in Algorithm 2. In Algorithm 2, we use RList to store the top-k files which have the k-largest relevance scores to the query. The RList is initialized to be an empty list, and it is updated when finding a relevance file. The kth score is defined as the smallest relevance score in the current RList, which is initialized to be a very small integer. By using the kth score, we can accelerate the search process by ignoring some paths with low scores. In Example 1 and Figure 4, an illustration of the search process is given, where $F = \{f_1, f_2, \ldots, f_6\}$, query vectors are $\overrightarrow{q_{\min}} = \{0, -0.3, 0\}$ and $\overrightarrow{q_{\max}} = \{0.1, 0, 0.2\}$, and d (vector dimension) is 3.

3.1.3. Example 1. An example of an index tree and a search process on this tree is illustrated in Figure 4. In Figure 4, we show an index tree with $F = \{f_1, f_2, \ldots, f_6\}$ in which the dimension of the vector for each node is 3. For each node u in the tree, the upper vector and lower vector are corresponding to $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$, respectively. In the tree building process, we first generate the leaf nodes from F and then create the internal nodes based on these leaf nodes.

Moreover, Figure 4 also gives an illustration of the search process. In Figure 4, we set $\overrightarrow{q} = \{0.1, -0.3, 0.2\}$ and split it into $\overrightarrow{q_{\min}} = \{0, -0.3, 0\}$ and $\overrightarrow{q_{\max}} = \{0.1, 0, 0.2\}$. We suppose that top-3 files will be returned to the data user. According to Algorithm 2, the search process begins with the root node r and calculates the score between the query Q and the two child nodes r_{11} and r_{12} of r by using equation (3). The calculation process is presented as follows:

$$\begin{aligned}\text{Score}(r_{11}, Q) &= \{-0.4, -0.2, -0.3\} \cdot \{0.0, -0.3, 0.0\} \\ &\quad + \{0.3, 0.3, 0.8\} \cdot \{0.1, 0.0, 0.2\} = 0.25, \\ \text{Score}(r_{12}, Q) &= \{-0.5, 0.7, -0.2\} \cdot \{0.0, -0.3, 0.0\} \\ &\quad + \{0.5, 0.9, 0.8\} \cdot \{0.1, 0.0, 0.2\} = 0.\end{aligned} \tag{4}$$

Because the score between r_{11} and Q is higher than that between r_{12} and Q, Algorithm 2 will traverse the subtree with r_{11} as the root node and compute the score between the query Q and two child nodes of r_{11}. Since the score between r_{21} and Q is higher than that between r_{22} and Q, Algorithm 2 will traverse the subtree with r_{21} as the root node and add the leaf nodes f_1, f_2 to the RList. After this, the subtree with r_{22} as the root node will be traversed, and the leaf nodes f_3 and f_4 are reached. Since the number of files in RList is less than 3, f_3 is added to RList directly. For the file f_4, since the number of files in RList equals 3 now, Algorithm 2 will compare the score between f_4 and Q to the minimum score in the RList. Because the score between f_4 and Q is smaller than the minimum score in the RList, f_4 is not added to the RList. At present, the subtree with r_{11} as the root node has been traversed. Algorithm 2 will traverse the subtree with r_{12} as the root node. As the score between r_{12} and Q is smaller than the minimum score in the RList, which means that the score of all child nodes of r_{12} is smaller than the minimum score in the RList (this property is described in Section 3.1.2), f_5 and f_6 will not be checked. Therefore, Algorithm 2 outputs $\text{RList} = \{f_1, f_2, f_3\}$.

3.2. Construction of SSE-DMKRS. In this section, through combining the secure KNN algorithm [32] and the index tree building algorithm, we propose a concrete SSE-DMKRS scheme. The SSE-DMKRS scheme consists of five algorithms. The algorithms *KeyGen*, *DictionaryBuild*, and *IndexBuild* are executed by the data owners, while the algorithms *TrapdoorGen* and *Search* are performed by the data users and the cloud server, respectively:

(i) KeyGen (λ): given a security parameter λ, this algorithm first randomly chooses four $d \times d$ invertible matrices M_{11}, M_{12}, M_{21}, and M_{22}, where d is the

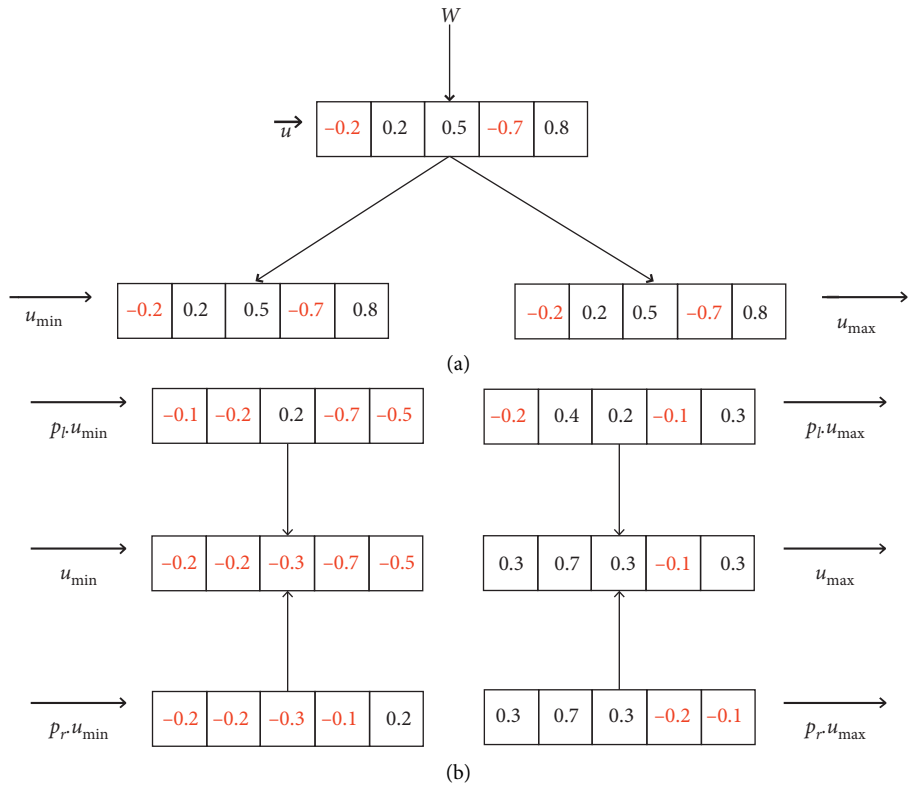

FIGURE 3: An example of the vectors generation of the node u. (a) Method M_1: u is a leaf node, W is a keyword set for the file which u stores, and $\overrightarrow{u}$ is a vector generated by adopting the keyword conversion method mentioned in Section 2.5. (b) Method M_2: u is an internal node, and $P_l \cdot \overrightarrow{u_{\min}}$, $P_l \cdot \overrightarrow{u_{\max}}$, $P_r \cdot \overrightarrow{u_{\min}}$ and $P_r \cdot \overrightarrow{u_{\max}}$ are the vectors of its children.

Input: the document collection $F = \{f_1, f_2, \ldots, f_n\}$, a semantic dictionary D generated by applying "Word2Vec" to F.
Output: the index tree T.

(1) **for** each $i \in [1, n]$ **do**:
(2) Construct a leaf node u for f_i, with $u.ID = GenID\ ()$, $u.P_l = u.P_r = NULL$, $u.FID = GenFID\ (f_i)$, and generate $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$ according to the method M_1;
(3) Insert u to *CurrentNodeSet*;
(4) **end for**
(5) **while** $|CurrentNodeSet| \geq 1$ **do**:
(6) **if** $|CurrentNodeSet|$ is even, i.e. $2h$ **then:**
(7) **for** each pair of nodes u' and u'' in *CurrentNodeSet* **do:**
(8) Create a parent node u for u' and u'', with $u.ID = GenID\ ()$, $u.P_l = u'$, $u.\ P_r = u''$, $u.FID = NULL$, and set $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$ according to the method M_2;
(9) Insert u to *TempNodeSet*;
(10) **end for**
(11) **else** \\Suppose that $|CurrentNodeSet| = 2h + 1$
(12) **for** each pair of nodes u' and u'' of the former $2h - 2$ nodes in *CurrentNodeSet* **do**:
(13) Create a parent node u for u' and u'';
(14) Insert u to *TempNodeSet*;
(15) **end for**
(16) Create a parent node u_1 for the $(2h - 1)$-th and $(2h)$-th nodes, and then generate a parent node u for the $(2h + 1)$-th node and u_1;
(17) Insert u to *TempNodeSet*;
(18) **end if**
(19) Set *CurrentNodeSet* = *TempNodeSet* and clear *TempNodeSet*;
(20) **end while**
(21) **return** *CurrentNodeSet*;
(22) \\Note that the *CurrentNodeSet* only contains one node which is the root of the index tree T.

ALGORITHM 1: BuildIndexTree (FileSet F, Dictionary D).

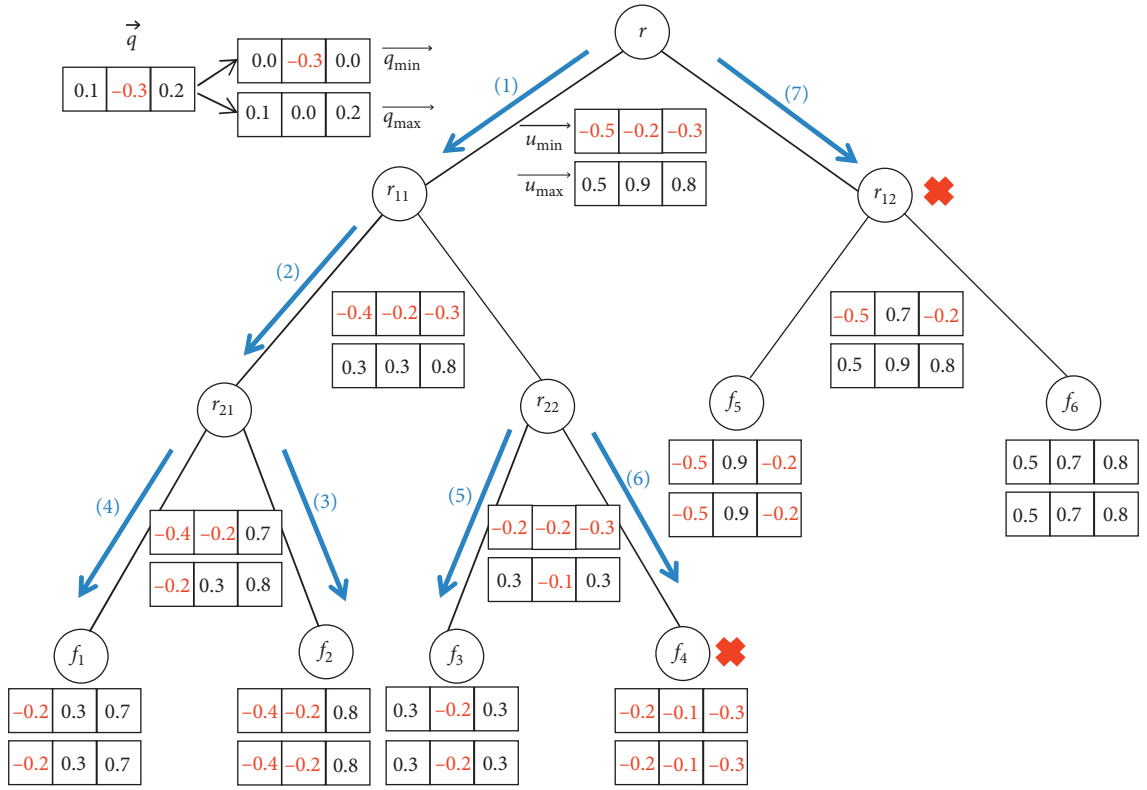

Figure 4: An example of Algorithm 1 and Algorithm 2 (example 1).

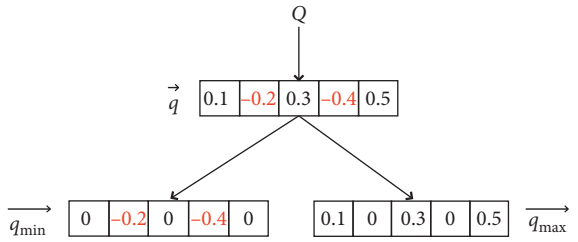

Figure 5: An example of the vector generation of the query Q. Method M_3: $\vec{q}$ is a vector generated by adopting the keyword conversion method mentioned in Section 2.5. $\overrightarrow{q_{\min}}$ holds all the negative part of $\vec{q}$, while $\overrightarrow{q_{\max}}$ holds the positive part.

dimension of $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$. Then, it randomly generates a d-bit vector S. Finally, it outputs the secret key $\mathrm{sk} = \{S, M_{11}, M_{12}, M_{21}, M_{22}\}$.

(ii) DictionaryBuild (F): given the document set $F = \{f_1, f_2, \ldots, f_n\}$, the algorithm runs "Word2vec" to generate the dictionary D of F. In the dictionary D, each keyword is associated with a vector representation. Besides, each keyword is also corresponding with a set of semantically related keywords.

(iii) IndexBuild (sk, F, D): given the document set F and the dictionary D for F, the algorithm first creates the index tree T by using the algorithm *BuildIndexTree* (F, D) (Algorithm 1). Then, for each node u in the tree T, the algorithm generates two random vector pairs $\{V'_{u_{\min}}, V''_{u_{\min}}\}$ and $\{V''_{u_{\max}}, V''_{u_{\max}}\}$ for the vectors of $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$, respectively. More precisely, if $S[i] = 0$, it sets $V'_{u_{\min}}[i] = V''_{u_{\min}}[i] = \overrightarrow{u_{\min}}[i]$ and $V'_{u_{\max}}[i] = V''_{u_{\max}}[i] = \overrightarrow{u_{\max}}[i]$; if $S[i] = 1$, $V'_{u_{\min}}, V''_{u_{\min}}, V'_{u_{\max}}, V''_{u_{\max}}$ are set as four random values under the constraints $V'_{u_{\min}}[i] + V''_{u_{\min}}[i] = \overrightarrow{u_{\min}}[i]$ and $V'_{u_{\max}}[i] + V''_{u_{\max}}[i] = \overrightarrow{u_{\max}}[i]$. This process is expressed as the following equation:

$$\left.\begin{cases} V'_{u_{\min}}[i] = V''_{u_{\min}}[i] = \overrightarrow{u_{\min}}[i], & \text{if } S[i] = 0; \\ V'_{u_{\max}}[i] = V''_{u_{\max}}[i] = \overrightarrow{u_{\max}}[i], & \text{if } S[i] = 0; \\ V'_{u_{\min}}[i] + V''_{u_{\min}}[i] = \overrightarrow{u_{\min}}[i], & \text{if } S[i] = 1; \\ V'_{u_{\max}}[i] + V''_{u_{\max}}[i] = \overrightarrow{u_{\max}}[i], & \text{if } S[i] = 1; \end{cases}\right\} i \in [1, d]. \tag{5}$$

Finally, for each node u, it computes $I_u = \{M_{11}^T V'_{u_{\min}}, M_{12}^T V''_{u_{\min}}, M_{21}^T V'_{u_{\max}}, M_{22}^T V''_{u_{\max}}\}$. Through replacing the plaintext vectors $\overrightarrow{u_{\min}}$ and $\overrightarrow{u_{\max}}$ with the encrypted index I_u, an encrypted index tree I_T is created.

(iv) TrapdoorGen (sk, Q): given a query keyword set Q, the algorithm first extends Q to a new semantic keyword set Q'. The process is as follows:

(a) It generates a new keyword set Q', which is initialized to an empty set.

Input: A vector $\overrightarrow{q}$ of query Q, a semantic dictionary D generated by applying "Word2Vec" to F, a root node u of *IndexTree* and *RList*.
Output: *RList*.
(1) Split $\overrightarrow{q}$ into $\overrightarrow{q_{\min}}$ and $\overrightarrow{q_{\max}}$ according to the method M_3;
(2) **if** u is an internal node **then:**
(3) **if** *Score (u, Q)* > *k*-th score **then:**
(4) *SearchIndexTree* ($\overrightarrow{q}$, *D*, *u.P*$_l$, *RList)*;
(5) *SearchIndexTree* ($\overrightarrow{q}$, *D*, *u.P*$_r$, *RList)*;
(6) **else:**
(7) **return**
(8) **end if**
(9) **else**
(10) **if** *Score (u, Q)* > *k*-th score **then:** \\Update *RList.*
(11) Delete the element holding the smallest relevance score in *RList*;
(12) Insert a new element <*Score (u, Q), u.FID*> in the *Rlist*, and sort the elements in *RList*;
(13) **end if**
(14) **return**
(15) **end if**

Algorithm 2: SearchIndexTree (QueryVector $\overrightarrow{q}$, Dictionary D TreeNode u RList).

(b) Note that each keyword in the dictionary is associated with a group of keywords semantically related to this keyword. For each keyword q in Q, it randomly chooses k' semantic keywords based on the dictionary and inserts these keywords into Q', where k' is chosen dynamically and $k \in [1, N]$.

Then, based on Q', the *TrapdoorGen* algorithm generates a pair of vectors $\overrightarrow{q_{\min}}$ and $\overrightarrow{q_{\max}}$ by adopting the method M_3. After this, it generates two random vector pairs $\{Q_{q_{\min}}', Q_{q_{\min}}''\}$ and $\{Q_{q_{\max}}', Q_{q_{\max}}''\}$ for the vectors of $\overrightarrow{q_{\min}}$ and $\overrightarrow{q_{\max}}$, respectively. This process is similar to the process in the *IndexBuild* algorithm and can be expressed as the following equations:

$$\left.\begin{cases} Q'_{q_{\min}}[i] = Q''_{q_{\min}}[i] = \overrightarrow{q_{\min}}[i], & \text{if } S[i] = 0; \\ Q'_{q_{\max}}[i] = Q''_{q_{\max}}[i] = \overrightarrow{q_{\max}}[i], & \text{if } S[i] = 0; \\ Q'_{q_{\min}}[i] + Q''_{q_{\min}}[i] = \overrightarrow{q_{\min}}[i], & \text{if } S[i] = 1; \\ Q'_{q_{\max}}[i] + Q''_{q_{\max}}[i] = \overrightarrow{q_{\max}}[i], & \text{if } S[i] = 1; \end{cases}\right\} i \in [1, d]. \tag{6}$$

Finally, this algorithm generates $T_Q = \{M_{11}^{-1}Q'_{q_{\min}}, M_{12}^{-1}Q''_{q_{\min}}, M_{21}^{-1}Q'_{q_{\max}}, M_{22}^{-1}Q''_{q_{\max}}\}$as the trapdoor for Q.

(v) Search (sk, T_Q, I_T): for each node u in I_T, the algorithm computes

$$\begin{aligned} I_u \cdot T_Q &= \left(M_{11}^T V'_{u_{\min}} \cdot M_{11}^{-1} Q'_{q_{\min}}\right) + \left(M_{12}^T V''_{u_{\min}} \cdot M_{12}^{-1} Q''_{q_{\min}}\right) \\ &\quad + \left(M_{21}^T V'_{u_{\max}} \cdot M_{21}^{-1} Q'_{q_{\max}}\right) + \left(M_{22}^T V''_{u_{\max}} \cdot M_{22}^{-1} Q''_{q_{\max}}\right) \\ &= \left(V'_{u_{\min}} \cdot Q'_{q_{\min}}\right) + \left(V''_{u_{\min}} \cdot Q''_{q_{\min}}\right) + \left(V'_{u_{\max}} \cdot Q'_{q_{\max}}\right) \\ &\quad + \left(V''_{u_{\max}} \cdot Q''_{q_{\max}}\right) \\ &= \overrightarrow{u_{\min}} \cdot \overrightarrow{q_{\min}} + \overrightarrow{u_{\max}} \cdot \overrightarrow{q_{\max}} = \text{Score}(u, Q). \end{aligned} \tag{7}$$

According to equation (3), the relevance score calculated from the encrypted vector I_u and the trapdoor T_Q equals the value of Score (u, Q). By using this property, the algorithm can utilize the *SearchIndexTree* algorithm (Algorithm 2) to perform ranked search.

3.3. Dynamic Update Operations. Besides search operation, the proposed scheme also supports some dynamic operations, e.g., documents insertion and deletion, satisfying the requirement of real-world application. Because the proposed scheme is built over a balanced binary tree, the update operations are realized by modifying the nodes in the tree. Inspired by the update method introduced in [14, 15], the update algorithm is presented as follows:

(i) UpdateInfoGen (sk, T_s, f_i, Utype): this algorithm is executed by the data owners and generates the update information $\{I_s, c_i\}$ to the cloud server, where T_s is a set containing all the update nodes, I_s is an encrypted form of T_s, f_i is the target document, c_i is an encrypted form of f_i, and Utype is the update type. In order to reduce the communication cost, the data owners will store the unencrypted index tree on its own device. For the Utype ∈ {Ins, Del}, the algorithm works as follows:

(a) If Utype = "Del," it means that the algorithm will delete a document f_i from the tree. The algorithm first finds the leaf node associated with the document f_i and deletes it. In addition, internal nodes associated with this leaf node are also added to T_s. Specifically, if the deletion operation will break the balance of the index tree, the algorithm can set the target leaf node as a fake node instead of removing it. After this, the algorithm encrypts T_s to generate I_s. Finally, the algorithm sends I_s to the cloud server and sets c_i as null.

(b) If Utype = "Ins," it means that the algorithm will insert a document f_i to the tree. The algorithm first creates a leaf node for f_i according to the method M_1 introduced in Section 3.1 and inserts this leaf node to T_s. Then, based on the method M_2, the algorithm updates the vectors of the internal nodes which are placed on the path from root to the new leaf node and inserts these internal nodes to T_s. Here, the algorithm prefers to replace the fake leaf node with the new leaf node rather than insert a new leaf node. Finally, the algorithm encrypts T_s and f_i to generate I_s and c_i, respectively, and sends them to the cloud server.

(ii) Update (I_T, C, I_s, c_i, Utype): this algorithm is executed by the cloud server to update the index tree I_T with encrypted nodes set I_s. After this, if Utype = "Del," then the algorithm removes c_i from C. Otherwise, the algorithm inserts c_i to C.

Note that after a period of insertion and deletion operations, the number of keywords in the dictionary should be changed. Because the dimensions of the index and trapdoor vectors in the previous schemes are linear with the number of keywords in the dictionary, these schemes have to rebuild the search index tree. By contrast, our scheme will not be affected by this problem. For the proposed scheme, the dimensions of the vectors in the index and trapdoor are determined by the tool of "Word2vec" and set by the users. For example, if we set the dimension of the vector as 200, the dimension of each keyword's vector is 200, and thus the dimensions of the vectors of $\overrightarrow{u_{\min}}$, $\overrightarrow{u_{\max}}$, $\overrightarrow{q_{\min}}$, and $\overrightarrow{q_{\max}}$ are all 200. According to the above analysis, our scheme is more suitable for the update operations than the previous schemes.

3.4. Security Analysis. In this section, we analyse the security of the proposed SSE-DMKRS scheme according to the privacy requirement introduced in Section 2.3:

(1) *Index and Trapdoor Privacy*. In the proposed scheme, each node u in the index tree and the query Q in the trapdoor are encrypted by using the secure KNN algorithm introduced in [32]. Thus, the attackers cannot obtain the original vectors in the tree nodes and the query, which means that the index and trapdoor privacy are well protected.

(2) *Trapdoor Unlinkability*. In the trapdoor generation phase, the query vector will be split randomly. Moreover, the same keyword set Q will be extended to be multiple different semantic keyword sets Q'. So, the same query Q will be encrypted to be different trapdoors, which means that the goal of the trapdoor unlinkability is achieved.

(3) *Keyword Privacy*. Since the index and the trapdoor are protected by the secure KNN algorithm, the adversary cannot infer the plaintext information from the index and the trapdoor under the known ciphertext model. Considering that the known background model is common in real-world applications, we will analyse the security of the proposed scheme under the known background model. For the *TrapdoorGen* algorithm, the original query keyword set Q is extended to a new set Q'. Specifically, for each keyword q in Q, randomly choosing a number k', the algorithm chooses k' semantic keywords related to q by utilizing the dictionary and inserts these keywords into the Q'. Suppose that each keyword is associated with N semantic keywords in the dictionary, each keyword can generate 2^N different keyword sets since each semantic keyword can be chosen or not. For example, if a keyword q is associated with three semantic keywords $\{q_1, q_2, q_3\}$, then q can generate 2^3 keyword sets $\{q\}$, $\{q, q_1\}$, $\{q, q_2\}$, $\{q, q_3\}$, $\{q, q_1, q_2\}$, $\{q, q_1, q_3\}$, $\{q, q_2, q_3\}$, and $\{q, q_1, q_2, q_3\}$. Since the query Q usually contains more than one keyword, Q will generate more than 2^N different semantic keyword sets. According to this method, the final similarity score is obfuscated by these random semantic keyword sets. As the analysis in [14, 15], our scheme can protect the keyword privacy under the known background model.

4. Proposed Scheme

In this section, we analyse the proposed SSE-DMKRS scheme theoretically and experimentally. A detailed experiment is given to demonstrate that our scheme can efficiently perform dynamic ranked keywords search over the encrypted data. Our experiment is run on Intel® Core™ i7 CPU at a 2.90 GHz processor and 16 GB memory size and is based on a real-world e-mail dataset called Enron e-mail dataset [35]. We mainly analyse the performance of our scheme in two aspects: (1) the efficiency of the proposed scheme including index building, trapdoor generation, search, and update; (2) the relationship between the search precision and the privacy level. Moreover, in order to show the advantages of our scheme, we also compare our scheme to two previous schemes related to our scheme. For simplicity, we denote these two schemes introduced in [14, 15] by X15 and G19.

4.1. Efficiency

4.1.1. Index Building. The process of index building mainly consists of two steps: (1) creating an unencrypted index tree by utilizing Algorithm 1; (2) encrypting each node in the tree by using the secure KNN scheme. In the tree building step, Algorithm 1 generates $O(n)$ nodes based on the document set F. Because each node has two vectors $\overrightarrow{u_{\min}}$, $\overrightarrow{u_{\max}}$ whose dimensions are both d, the vector splitting process needs $O(d)$ time and the matrix multiplication operations take $O(d \times d)$ time in the encryption step. According to these two steps, the whole time complexity of index building is $O(nd^2)$, which means that the time cost for index building mainly depends on the number of documents in F and the dimension of each node's vector.

Since the dimensions of each node's vector in X15 and G19 are both linear with the number of keywords in the dictionary (m), the time costs for index building in X15 and G19 are both $O(nm^2)$. Due to $d \ll m$, we can argue that the time cost for index building in our scheme is much less than that in X15 and G19. In addition, for the scheme G15, the internal nodes are constructed by the tool called bloom filter, and thus the dimension of each internal node's vector is linear with b. Since b is usually smaller than m, the index building time in G19 is less than that in X15.

Figure 6(a) shows that the time cost for index building in our scheme is much less than that in X15 and G19. More precisely, when $n = 1000$, $m = 20000$, $d = 1000$, and $b = 10000$, the time consumption for index building in X15 and G19 is nearly 100~200 times that in our scheme, respectively. As m increases, the advantages of our scheme will become even more significant.

In addition, because the index tree has $O(n)$ nodes and each node holds two d-dimensional vectors, the space complexity of the index tree is $O(nd)$. By contrast, the space complexities of the index tree in X15 and G19 are both $O(nm)$. From Table 3, even if we set $n = 1000$, $m = 20000$, $d = 1000$, and $b = 10000$, the storage cost of the index tree in our scheme is still much less than that in X15 and G19.

4.1.2. Trapdoor Generation. In our scheme, the query is converted to be two vectors $\overrightarrow{q_{\min}}$ and $\overrightarrow{q_{\max}}$, whose dimensions are both d. The trapdoor generation process is to multiply these two vectors by the $d \times d$ matrices in the key. So, the time complexity of trapdoor generation in our scheme is $O(d^2)$. By contrast, since the dimensions of query vectors in X15 and G19 are both m, the time complexities of trapdoor generation are both $O(m^2)$. Thus, the time cost of trapdoor generation of our scheme is much less than that in X15 and G19. Particularly, from Figure 6(b), when $n = 1000$, $m = 20000$, and $d = 1000$, the time cost for trapdoor generation in our scheme is 1.5 ms, while that in G19 and X15 is 287 ms and 290 ms, respectively.

4.1.3. Search. In the search process, if the relevance score of an internal node u and the query Q is less than the minimum relevance score of the current top-k documents, the subtree which uses node u as the root node will not be accessed. Thus, not all of the nodes in the tree will be accessed during the search process. We suppose that there are θ leaf nodes that contain at least one keyword in the query Q. Since the height of the tree is $O(\log n)$ and the time complexity of the relevance score calculation is $O(d)$, the time complexity of the search process is $O(\theta d \cdot \log n)$. For the scheme X15, because the time complexity of relevance score calculation is $O(m)$, the time complexity of the search process is $O(\theta m \cdot \log n)$ in X15. For the scheme G19, because each internal node contains a Bloom filter whose size is b and each leaf node involves a vector whose size is m, the time complexity of search process in G19 is $O(\theta(m + b \cdot \log n))$. From Figure 6(c), when $n = 1000$, $m = 20000$, $d = 1000$, and $b = 10000$, the search time cost in our scheme is 36 ms, while that in G19 and X15 is 135 ms and 214 ms, respectively.

4.1.4. Update. When the data owners want to insert or delete a document, they will not only insert or delete a leaf node, but also update $O(\log n)$ internal nodes. Since the encryption time for each node is $O(d^2)$, the time complexity of an update operation is $O(\log n \cdot d^2)$. For X15 scheme, because the encryption time for each node is $O(m^2)$, the time complexity of an update operation is $O(\log n \cdot m^2)$. For G19 scheme, because the internal nodes are based on the Bloom filter which is not encrypted, the time cost for updating the internal nodes can be ignored. Thus, the time complexity of update in G19 is $O(m^2)$ since only the leaf node is encrypted. From Figure 6(d), when $n = 1000$, $m = 20000$, $d = 1000$, and $b = 10000$, the time cost for updating one document in our scheme is 16 ms, while that in X15 and G19 is 1020 ms and 107 ms, respectively.

4.2. Precision and Privacy. The search precision of our scheme is affected by a group of semantic keywords related to the original index and query keywords. We measure our scheme by adopting a metric called "precision" defined in [12]. The metric of precision is defined as follows:

$$P_k = \frac{k'}{k}, \tag{8}$$

where k' is the number of real top-k documents in the retrieved k documents.

In addition, the semantic keywords in the index and query keyword set will disturb the relevance score calculation in the search process, which makes it harder for adversaries to identify keywords in the index and trapdoor through the statistical information about the dataset. To measure the disturbance extent of the relevance score, we use the following equation called "rank privacy" introduced in [12] to quantify this obscureness:

$$P_k' = \sum \frac{|r_i - r_i'|}{k^2}, \tag{9}$$

where r_i is the rank number of the document i in the retrieved top-k documents and r_i' is document i's real rank number in the real ranked results.

We compare our scheme to the schemes of X15 and G19 in terms of "precision" and "rank privacy." Note that an important parameter in the previous two schemes is a standard deviation σ, which is utilized to adjust the relevance score for the dummy keywords. In the comparison, we set $\sigma = 0.05$, which is usually used in the previous schemes. Besides, in our scheme, we set the number of semantic keywords for each keyword in the dictionary is 100, and the dimension of each node's vector is 1000 ($d = 1000$). Based on these settings, the comparison is illustrated in Figure 7.

From Figure 7, as k grows from 10 to 50, the precision of our scheme decreases slightly from 59% to 55%, and the rank privacy increases slightly from 26% to 28%. For the schemes X15 and G19, the precision decreases and the rank privacy increases when k grows. This characteristic exists in all three schemes. Because the vector representations for the index tree and query in our scheme are compressed deeply, some statistical information in the index and the query will be lost.

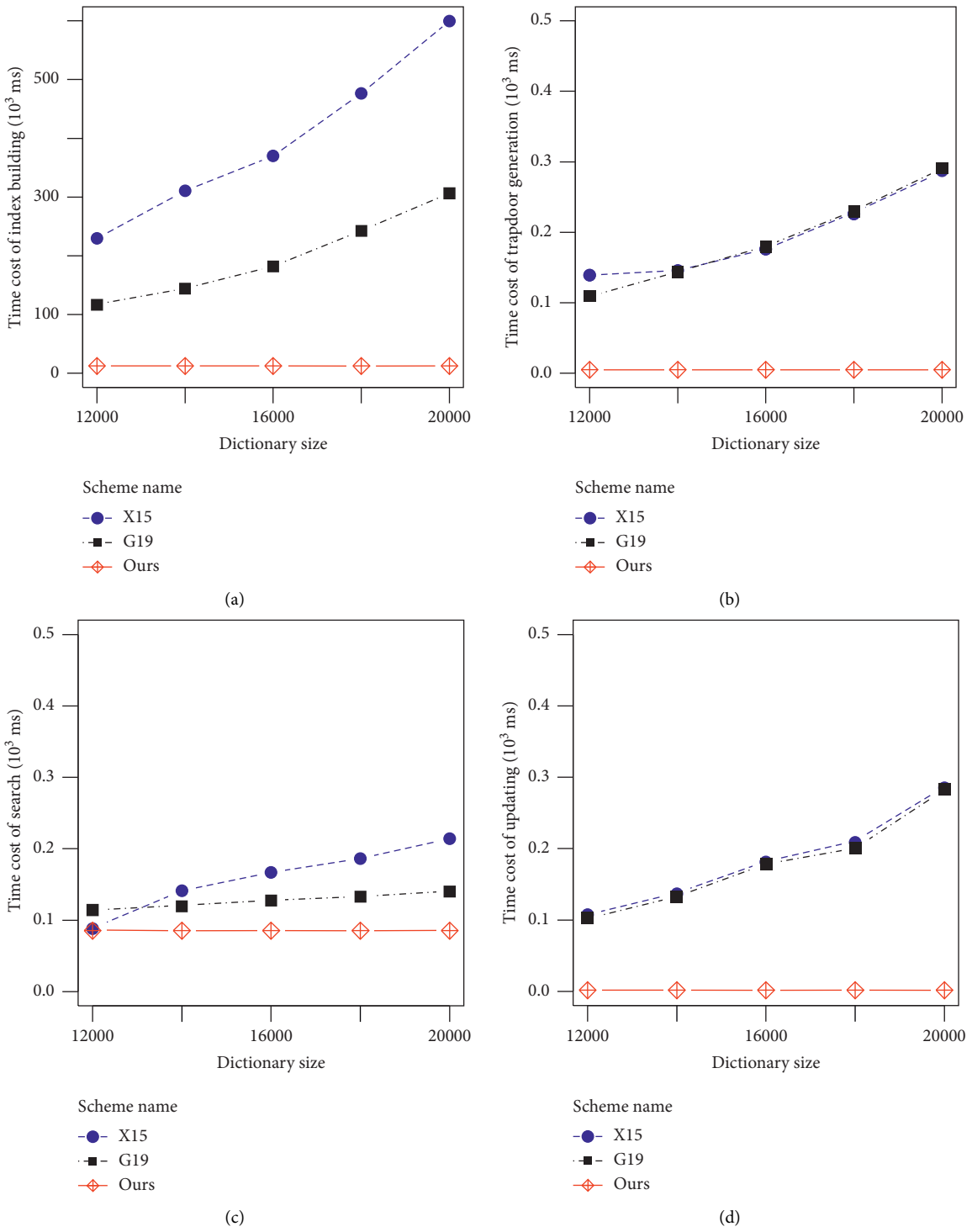

Figure 6: Impact of m on the time cost of index building (a), trapdoor generation (b), search (c), and update (d) ($n = 1000$, $d = 1000$, $b = 10000$, and $m = (12000; 14000; 16000; 18000; 20000)$).

Table 3: Storage consumption of the index tree (MB).

Dictionary size	[14]	[15]	Vector dimension	Proposed
$m = 12000$	188	174	$d = 200$	7
$m = 14000$	219	190	$d = 400$	14
$m = 16000$	251	206	$d = 600$	20
$m = 18000$	283	222	$d = 800$	26
$m = 20000$	315	238	$d = 1000$	33

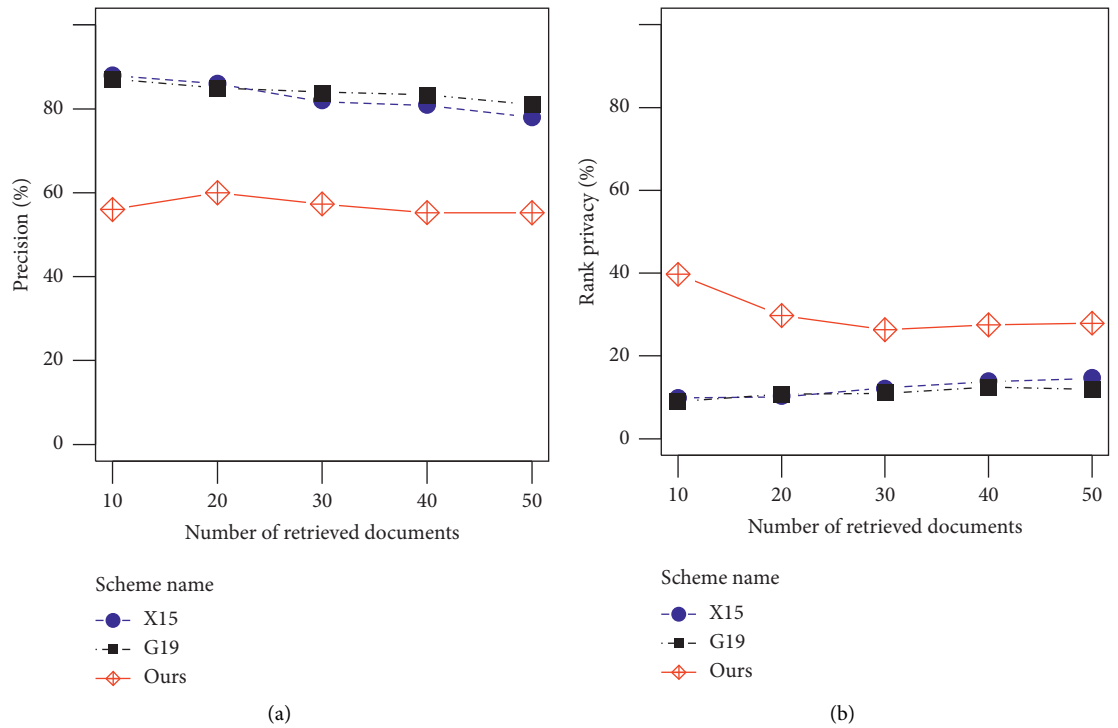

FIGURE 7: The precision (a) and rank privacy (b) of searches with different numbers of retrieved documents ($n = 1000$, $d = 1000$, $b = 10000$, $m = 12000$, and $\sigma = 0.05$).

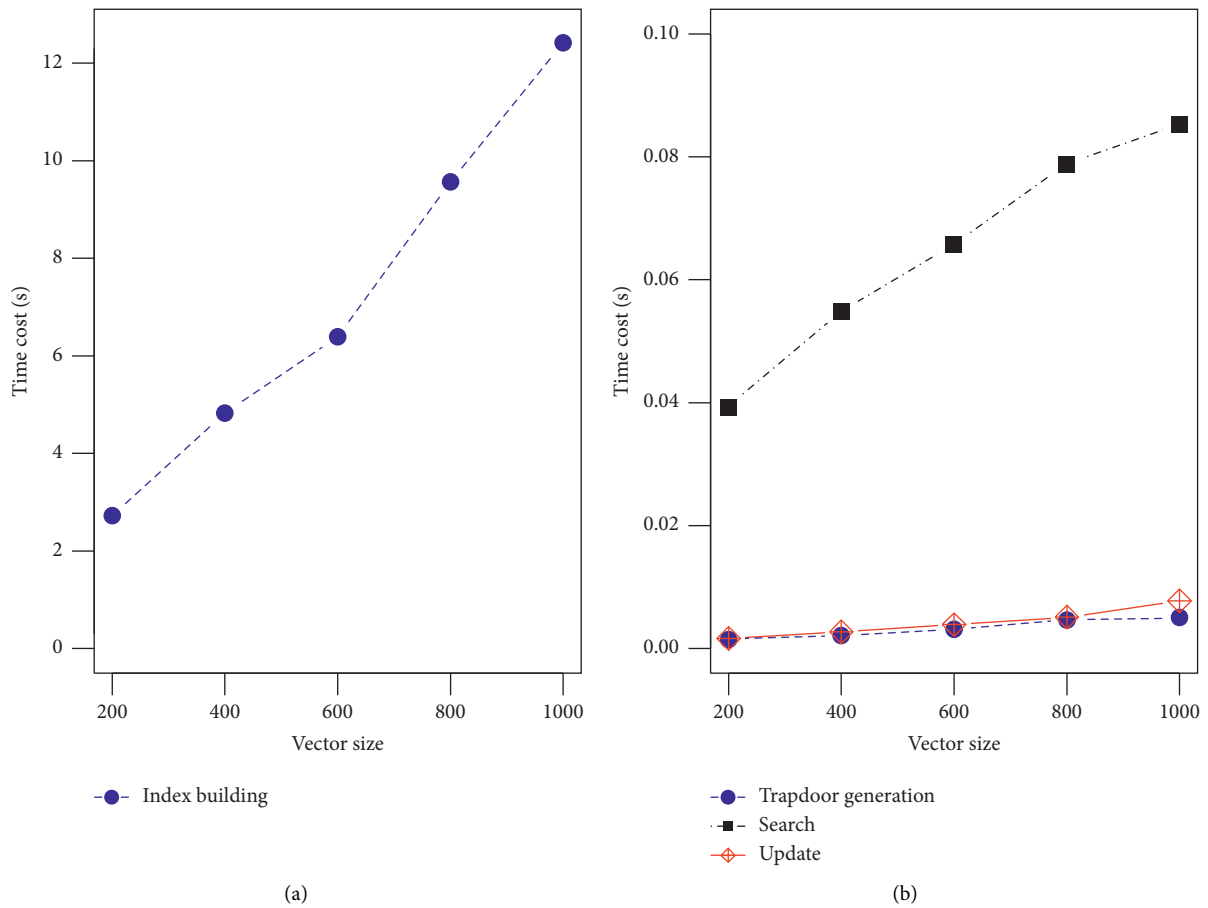

FIGURE 8: Impact of d on the time cost of index building (a) and trapdoor generation, search, and update (b) ($n = 1000$ and $d = (200; 400; 600; 800; 1000)$).

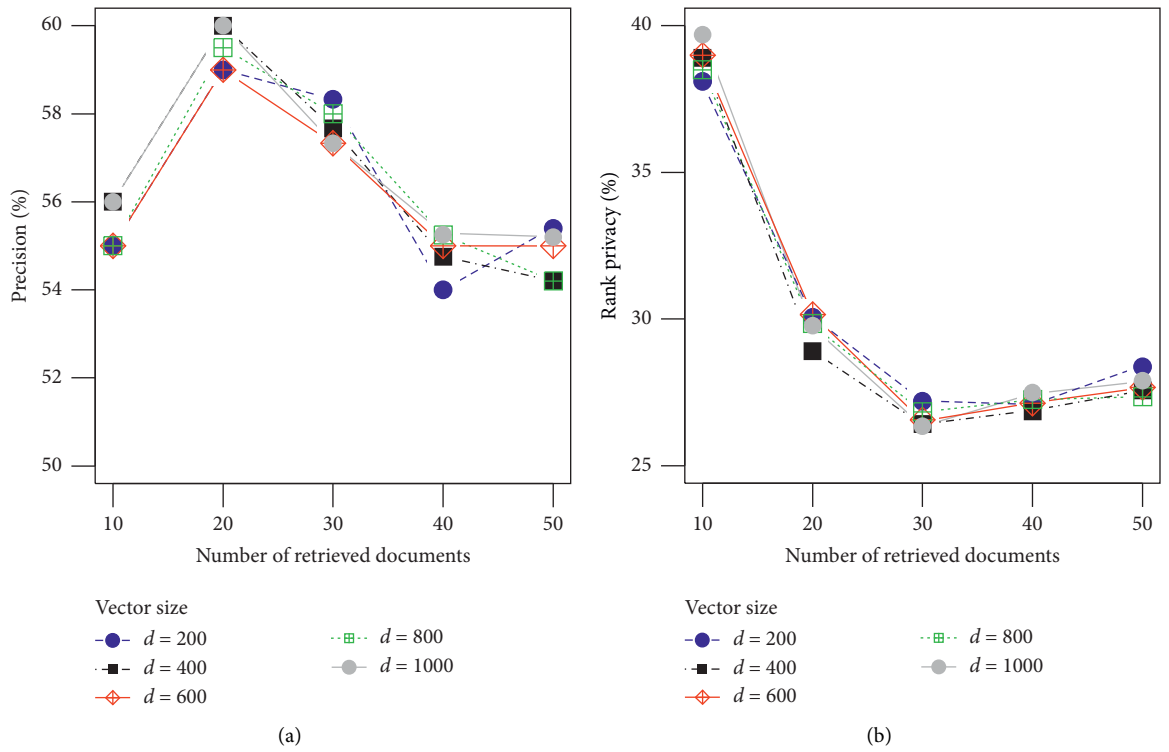

Figure 9: The precision (a) and rank privacy (b) of searches with different vector dimensions ($n = 1000$ and $d = (200; 400; 600; 800; 1000)$).

Thus, the precision of our scheme is less than that in X15 and G19. However, the rank privacy in our scheme is accordingly more than that in X15 and G19.

4.3. Impact of the Dimension of Vector Representation. The dimension of the vector representation (d) which we set in the "Word2vec" is an important parameter in our scheme. Next, we give the discussion of the impact of d for our scheme. The impact of d on the efficiency of our scheme is given in Figure 8. From Figure 8, we know that the time costs of index building, trapdoor generation, search, and update all increase when d grows. Besides, Figure 9 gives an illustration of the impact of d on the precision and rank privacy in our scheme. As d increases from 200 to 1000, the precision of our scheme increases slightly, while the rank privacy decreases gradually accordingly. These phenomena are all consistent with our previous theoretical analysis. So, in the proposed scheme, data users can balance the efficiency and accuracy by adjusting the parameter d to satisfy the requirements of different applications.

4.4. Discussion. From the experiment results, when $n = 1000$, $m = 20000$, $d = 200$, and $b = 10000$, the time cost of index building is 3 s, the generation time of a single trapdoor is 1.5 ms, and the search time is 36 ms, which are all much better than the previous schemes X15 and G19. Efficiency in our scheme demonstrates that our scheme is extremely suitable for practical applications, especially the mobile cloud setting in which the clients have limited computation and storage resources.

The experiment result shows that the precision of our scheme is less than that in the previous two schemes, while the rank privacy is more than that in the previous schemes accordingly. In addition, by using the "Word2vec" method, the vector representations used in our scheme contain the semantic information of the documents and queries. Based on these facts, we argue that the proposed scheme is suitable for applications requiring similarity and semantic search, such as mobile recommendation system, mobile search engine, and online shopping system.

5. Conclusions

In this paper, by applying "Word2Vec" to construct the vector representations of the documents and queries and adopting the balanced binary tree to index the documents, we proposed a searchable symmetric encryption scheme supporting dynamic multikeyword ranked search. Compared with the previous schemes, our scheme can tremendously reduce the time costs of index building, trapdoor generation, search, and update. Moreover, the storage cost of the secure index is also reduced significantly. Considering that the precision of our scheme can be further improved, we will construct a more accurate scheme based on the recent information retrieval techniques in the future work.

Data Availability

The data used to support the findings of this study is available from the following website: Http://www.cs.cmu.edu/~./enron/.

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the publication of this paper.

Acknowledgments

The authors gratefully acknowledge the support of the National Natural Science Foundation of China under Grants nos. 61402393 and 61601396 and the Nanhu Scholars Program for Young Scholars of XYNU.

References

[1] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searching on encrypted data," in *Proceedings of the 2000 IEEE Symposium on Research in Security and Privacy*, Berkeley, CA, USA, May 2000.

[2] Y. Zhu, D. Ma, and S. Wang, "Secure data retrieval of outsourced data with complex query support," in *Proceedings of the 2012 32nd International Conference on Distributed Computing Systems Workshops*, pp. 481–490, Macau, China, June 2012.

[3] Z. Fu, K. Ren, J. Shu, X. Sun, and F. Huang, "Enabling personalized search over encrypted outsourced data with efficiency improvement," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2546–2559, 2015.

[4] E. J. Goh, "Secure indexes," *IACR Cryptology ePrint Archive*, vol. 2003, p. 216, 2003.

[5] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

[6] J. W. Byun, D. H. Lee, and J. Lim, "Efficient conjunctive keyword search on encrypted data storage system," *European Public Key Infrastructure Workshop*, Springer, Berlin, Germany, pp. 184–196, 2006.

[7] L. Ballard, S. Kamara, and F. Monrose, "Achieving efficient conjunctive keyword searches over encrypted data," *Information and Communications Security*, Springer, Berlin, Germany, pp. 414–426, 2005.

[8] Z. Fu, X. Wu, C. Guan, X. Sun, and K. Ren, "Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 12, pp. 2706–2716, 2017.

[9] M. Kuzu, M. S. Islam, and M. Kantarcioglu, "Efficient similarity search over encrypted data," in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, pp. 1156–1167, Washington, DC, USA, April 2012.

[10] S. Zerr, D. Olmedilla, W. Nejdl, and W. Siberski, "Zerber + r: top-*k* retrieval from a confidential index," in *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology*, pp. 439–449, Saint Petersburg, Russia, March 2009.

[11] C. Wang, N. Cao, K. Ren, and W. Lou, "Enabling secure and efficient ranked keyword search over outsourced cloud data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1467–1479, 2012.

[12] N. Cao, C. Wang, M. Li et al., "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 222–233, 2013.

[13] W. Sun, B. Wang, N. Cao et al., "Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pp. 71–82, Hangzhou, China, 2013.

[14] Z. Xia, X. Wang, X. Sun, and Q. Wang, "A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 340–352, 2016.

[15] C. Guo, R. Zhuang, C.-C. Chang, and Q. Yuan, "Dynamic multi-keyword ranked search based on bloom filter over encrypted cloud data," *IEEE Access*, vol. 7, pp. 35826–35837, 2019.

[16] D. Cash, S. Jarecki, C. Jutla et al., "Highly-scalable searchable symmetric encryption with support for boolean queries," *Annual Cryptology Conference*, Springer, Berlin, Germany, pp. 353–373, 2013.

[17] D. Cash, J. Jaeger, S. Jarecki et al., "Dynamic searchable encryption in very-large databases: data structures and implementation," in *Proceedings of the Network and Distributed System Security Symposium*, pp. 23–26, San Diego, CA, USA, February 2014.

[18] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[19] D. Boneh, G. D. Crescenzo, R. Ostrovsky et al., "Public key encryption with keyword search," *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 506–522, Springer, Berlin, Germany, 2004.

[20] Y. Zhang, Y. Li, and Y. Wang, "Conjunctive and disjunctive keyword search over encrypted mobile cloud data in public key system," *Mobile Information Systems*, vol. 2018, Article ID 3839254, 11 pages, 2018.

[21] J. Katz, A. Sahai, and B. Waters, "Predicate encryption supporting disjunctions, polynomial equations, and inner products," *Advances in Cryptology–EUROCRYPT 2008*, pp. 146–162, Springer, Berlin, Germany, 2008.

[22] Y. Zhang, Y. Li, and Y. Wang, "Secure and efficient searchable public key encryption for resource constrained environment based on pairings under prime order group," *Security and Communication Networks*, vol. 2019, Article ID 5280806, 14 pages, 2019.

[23] Y. Wu, J. Hou, J. Liu, W. Zhou, and S. Yao, "Novel multi-keyword search on encrypted data in the cloud," *IEEE Access*, vol. 7, pp. 31984–31996, 2019.

[24] P. Xu, Q. Wu, W. Wang, W. Susilo, J. Domingo-Ferrer, and H. Jin, "Generating searchable public-key ciphertexts with hidden structures for fast keyword search," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 9, pp. 1993–2006, 2017.

[25] P. Xu, S. He, W. Wang, W. Susilo, and H. Jin, "Lightweight searchable public-key encryption for cloud-assisted wireless sensor networks," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 8, pp. 3712–3723, 2017.

[26] F. Han, J. Qin, H. Zhao, and J. Hu, "A general transformation from KP-ABE to searchable encryption," *Future Generation Computer Systems*, vol. 30, pp. 107–115, 2014.

[27] H. Kai, G. Jun, W. Jian, J. Weng, J. K. Liu, and X. Yi, "Attribute-based hybrid boolean keyword search over outsourced encrypted data," *IEEE Transactions on Dependable and Secure Computing*, p. 1, 2018.

[28] M. Sepehri, S. Cimato, E. Damiani, and C. Y. Yeun, "Data sharing on the cloud: a scalable proxy-based protocol for privacy-preserving queries," in *Proceedings of the 2015 IEEE*

Trustcom/BigDataSE/ISPA, pp. 1357–1362, Helsinki, Finland, August 2015.

[29] M. Sepehri, S. Cimato, and E. Damiani, "Efficient implementation of a proxy-based protocol for data sharing on the cloud," in *Proceedings of the Fifth ACM International Workshop on Security in Cloud Computing*, pp. 67–74, New York, NY, USA, April 2017.

[30] Y. Zhang, Y. Wang, and Y. Li, "Searchable public key encryption supporting semantic multi-keywords search," *IEEE Access*, vol. 7, pp. 122078–122090, 2019.

[31] T. Mikolov, K. Chen, G. Corrado et al., "Efficient estimation of word representations in vector space," 2013, https://arxiv.org/abs/1301.3781.

[32] W. K. Wong, D. W.-L. Cheung, B. Kao, and N. Mamoulis, "Secure kNN computation on encrypted databases," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pp. 139–152, New York, NY, USA, 2009.

[33] S. Zerr, E. Demidova, D. Olmedilla, W. Nejdl, M. Winslett, and S. Mitra, "Zerber: r-confidential indexing for distributed documents," in *Proceedings of the 11th International Conference on Extending Database Technology Advances in Database Technology*, pp. 287–298, Nantes, France, March 2008.

[34] C. D. Manning, P. Raghavan, and H. SchÃijtze, *Introduction to Information Retrieval*, Cambridge University Press, Cambridge, UK, 2008.

[35] W. W. Cohen, "Enron E-mail dataset," http://www.cs.cmu.edu/~./enron/.