# Forward Secure Searchable Encryption over Medical Cloud Data

Yang Mi, Cheng Guo, Qianqian He(✉), and Xinyu Tang

School of Software Technology, Dalian University of Technology, Dalian, China
he_qianqian@126.com

**Abstract.** In medical cloud computing, medical data (e.g., electronic health records and diagnostic report) are allowed to be outsourced to the remote cloud server. Since cloud service providers are semi-trusted, it is important to protect the privacy of medical data while ensuring the convenient access of stored data. Several schemes have been proposed for this purpose. In this paper, we propose a novel scheme for high-dimension medical data based on searchable symmetric encryption (SSE) and locality sensitive hashing (LSH), which is able to provide greater security for the stored on the medical cloud—forward privacy, while ensuring efficient search and update performance. Combined with the experimental results, we carried out a detailed security analysis and performance analysis of the proposed scheme. The results show that the server cannot observe which query request is associated with the update object. Therefore, the proposed scheme is forward secure and efficient in practice.

**Keywords:** Medical cloud computing · searchable encryption · electronic health records · data privacy · forward privacy

## 1 Introduction

With the increasing development of cloud computing, cloud storage has become a particularly popular data storage solution used by many data owners due to its scalability, high availability, easy managerial advantages and low costs. Like other organizations, health care systems with a large amount of medical data are considering outsourcing their electronic health records to remote cloud servers to reduce storage overhead and management burden. However, cloud storage as a third-party platform is semi-trusted and puts sensitive and private medical data above certain security problems. At the same time, the biggest threat comes from the cloud service provider. Therefore, how to guarantee data security during the process of storage and use of cloud medical data is a significant problem.

To address the above security problem of data in cloud storage, searchable symmetric encryption (SSE) [1] designed to allow users to search over encrypted data has been proposed. Specifically, SSE allows one to encrypt data using a symmetrical encryption algorithm and subsequently to search over the encrypted data. To ensure the security


V. C. M. Leung et al. (Eds.): Qshine 2023, LNICST 573, pp. 384–397, 2024.
https://doi.org/10.1007/978-3-031-65126-7_33

of medical cloud data, many schemes based on SSE have been proposed [2–4]. These schemes based on SSE technology aim at protecting sensitive medical data and further expand from multiple keywords, security strength and access control. In addition, most of these existing schemes use the form of retrieving documents by keywords. However, such a form does not apply to scenarios where high-dimension data is retrieved by high-dimension data, e.g., searching similar genes for a particular gene or searching similar images for a certain CT image.

In this paper, we propose a novel forward secure scheme for high-dimension medical data in cloud servers. Our scheme uses a combination of SSE and locality sensitive hashing (LSH) [5] as a framework to support searching similar high-dimension data by high-dimension data. LSH is a technology that maps the closer points in high-dimension space to the same hash bucket with a higher probability. In the framework of the combination of SSE and LSH, searching similar high-dimension data by high-dimension data is transformed into searching similar high-dimension data by the hash value of the high-dimension data. In particular, our proposed scheme can achieve forward privacy of SSE by modifying the hash table structure. Forward privacy [6] that requires that the newly added data to not be related to a previous query request is the goal that current dynamic SSE schemes are pursuing [7–10], and it has almost become an essential property since the file-inject attacks [11]. The traditional hash table stores a list of data IDs in each bucket. The data mapped to the same bucket through the LSH function can be regarded as similar data, so the data in the list of the same bucket can be regarded as similar data. Such a structure can easily expose the data's bucket location, so it can reveal the association between the newly added data and the previous query request. In this paper, we separate the hash table and the list, and merge several small lists in all hash buckets into a large list. We store the hash table locally and the large list in the cloud server. The new index makes our scheme forward secure. We summarize the main features of our proposed approach as follows:

**Applicability to High-Dimension Medical Cloud Data.** We implement searching high-dimension medical data by high-dimension medical data using a combination of SSE and LSH.

**Forward Privacy.** We modify the traditional hash table structure and design a new index structure allows our scheme to achieve forward privacy.

**Practical Implementation and Efficiency.** We use two datasets to evaluate our scheme, and the experimental results indicate that our scheme is efficient.

The rest of the paper is organized as follows. Section 2 summarizes related work and background information is given in Sect. 3. Then, we present our scheme in Sect. 4. In Sect. 5, we give the security analysis and experimental results. Finally, Sect. 6 presents conclusions.

## 2 Related Work

SSE was first proposed by Song et al. [1], and their scheme and their scheme encrypts each word in the document and then allows searching for documents using a keyword. Subsequently, SSE was formalized by Curtmola et al. [12], who developed the security

definitions for SSE schemes. The security notions known as SSE were improved, and a protocol was given that was compatible with the definitions. Later, Kamara et al. [13] constructed the first dynamic and sublinear scheme, although the dynamic SSE had been studied earlier.

Zhang et al. [11] introduced file injection attacks on SSE that highlighted the importance of forward privacy and promoted extensive research on forward privacy. Since then, forward privacy has almost become an essential property for dynamic SSE schemes. Bost et al. [7] presented the forward-privacy scheme known as Sophos using trapdoor permutations. The efficiencies of communication and computation have been improved significantly compared with the ORAM-based schemes. Even so, the computational efficiency has been degraded because the trapdoor permutations are based on a public key algorithm. Later, some forward secure SSE schemes based on symmetric cryptographic primitives were been proposed [8–10].

The emergence of SSE provides a secure and effective technical solution foundation for many cloud storage applications, especially for the medical systems with large amounts of sensitive and private medical data. To ensure the security of medical cloud data, many schemes based on SSE have been proposed [2–4]. Most of these existing schemes retrieve documents by keywords. However, this process does not apply to scenarios where high-dimension data is retrieved by high-dimension data, e.g., searching similar genes for a particular gene or searching similar images for a certain CT image. In this paper, we propose a novel forward secure scheme for high-dimension medical data in cloud server.

## 3 Preliminaries

### 3.1 Locality Sensitive Hash (LSH)

LSH is an effective similarity search algorithm in high-dimension space [5]. The basic idea is to hash similar objects into the same bucket with a greater probability than dissimilar objects. When a query operation is performed, first, the query point is hashed into buckets in hash tables, and then all the points in these hit buckets are returned as a result of the query.

**Definition 1** (LSH family, $H$)**:** $S$ is a set of $n$ points, and $U$ is a set of hash values, $S \subset R^d$. A family $H = \{h : S \to U\}$ of functions is called $(R, cR, p_1, p_2)$-sensitive if $\forall p, q \in R^d$:

- If $D(p, q) \leq R$, then $\Pr[h(p) = h(q)] \geq p_1$.
- If $D(p, q) > cR$, then $\Pr[h(p) = h(q)] < p_2$.

where $c > 1$ and $p_1 > p_2$ for similarity search.

**Definition 2** (Concatenation LSH Functions, $G$)**:** $G = \{g : R^d \to U^k\}$ is a set of concatenation LSH functions, where $\forall g_i \in G$ is composed of $k$ hash functions randomly extracted from $G$. Formally,

$$g_i(p) = (h_{i1}(p), \ldots, h_{ik}(p)) \tag{1}$$

where $k$ is the number of functions in each concatenation function, and $h_{i1}, \ldots, h_{ik}$ are randomly chosen from the $H$.

We use these concatenation LSH functions to construct hash tables. Therefore:

- If $D(p, q) \leq R$, then $\Pr[g(p) = g(q)] \geq p_1^k$.
- If $D(p, q) > cR$, then $\Pr[g(p) = g(q)] < p_2^k$

To increase the recall, $l$ concatenation LSH functions typically are used to build $l$ hash tables.

### 3.2 Cryptographic Primitive

A private-key encryption scheme is a tuple that is composed of three algorithms *(Gen, Enc, Dec). Gen* is the probabilistic key generation algorithm that takes a security parameter, $\lambda$, as input and returns a secret key, $K$, where $|K| > \lambda$. *Enc* is the probabilistic encryption algorithm that takes a key, $K$, and a plaintext, $M \in \{0, 1\}^*$, as input and returns a ciphertext $C \in \{0, 1\}^*$. *Dec* is the deterministic decryption algorithm that takes $K$ and $C \in \{0, 1\}^*$ as inputs and returns $M \in \{0, 1\}^*$.

### 3.3 SSE Scheme for High-Dimension Data

**Definition 3:** An SSE scheme for high-dimension data that enables the similarity search, and the dynamic update consists of three algorithms:

- $((\sigma, key_q); (I, C)) \leftarrow Setup((\lambda, \text{LSH family}); \perp)$: This is a probabilistic algorithm run by the data owner. It takes a security parameter, $\lambda$, and LSH family as input. It returns the secret key, $key_q$, and the client's state, $\sigma$, to the data owner and returns the encrypted index, $I$, and encrypted dataset $C$ to the server.
- $((\sigma', r_u); (I\prime, C\prime)) \leftarrow Update((K, \sigma, q, op); (I, C))$: This is a deterministic algorithm run by the client and the server. On the client side, it takes the secret key, $K$, the client's state, $\sigma$, the update object, $q$, and operation $op$ (*add* or*del*) as input. On the server side, it takes the encrypted index, $I$, and encrypted dataset,$C$, as input. It returns the updated client state, $\sigma'$, and the update result,$r_u$, to the client, and returns the updated encrypted index, $I'$, and updated dataset, $C'$, to the server.
- $(r_s; \perp) \leftarrow Search((K, \sigma, q); (I, C))$: This is a deterministic algorithm run by the client and the server. On the client side, it takes the secret key, $K$, the client's state, $\sigma$, and a search object, $q$, as input. On the server side, it takes the encrypted index, $I$, and encrypted dataset, $C$, as input. It returns a result set, $r_s$, to the client.

### 3.4 Privacy Definition

The security definition in this paper follows the simulation-based model [12]. It is parameterized by a collection of leakage functions, $\mathcal{L} = \{\mathcal{L}_{Setup}, \mathcal{L}_{Update}, \mathcal{L}_{search}\}$. These functions describe the information that the scheme leaks to the adversary.

**Definition 4** (Adaptive Secure SSE scheme)**:** Let $\Omega$ be an SSE scheme with $\mathcal{L} = \{\mathcal{L}_{Setup}, \mathcal{L}_{Update}, \mathcal{L}_{search}\}$, $\mathcal{A}$ be an adversary, $S$ be a simulator, two games are defined as follows:

$\mathbf{Real}_{\mathrm{A}}^{\Omega}$: The adversary $a$ is given $C$ generated by $Setup((\lambda, \text{LSH family}); \perp)$. Then, $\mathcal{A}$ performs adaptive update $u$ or query $q$. The challenger answers by running $Update((K, \sigma, q, op); (I, C))$ for $u$. The challenger answers by running $Search((K, \sigma, q); (I, C))$ for $q$. Finally, $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.

$\mathbf{Ideal}_{\mathrm{A}}^{\Omega}$: An encrypted index $I$ and an encrypted dataset $C$ are generated by simulator $\mathcal{S}$ through running $\mathcal{S}(Setup)$ based on the leakage function $\mathcal{L}_{Setup}$ and then given to $\mathcal{A}$. Then, $\mathcal{A}$ performs adaptive update $u$ or query $q$. For $u$, $\mathcal{S}$ is given $\mathcal{L}_{Update}(q)$, and answers it by running $\mathcal{S}(Update)$. For $q$, $\mathcal{S}$ is given $\mathcal{L}_{search}(q)$, and answers it by running $\mathcal{S}(Search)$. Finally, $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.

$\Omega$ is adaptively secure with leakage functions $\mathcal{L}$, if for any polynomial-time adversary $\mathcal{A}$, there exists an efficient polynomial-time simulator $\mathcal{S}$ such that:

$$\Pr\left(\mathbf{Real}_{\mathcal{A}}^{\Omega}(\lambda) = 1\right) - \Pr(\mathbf{Ideal}_{\mathcal{A}}^{\Omega}(\lambda) = 1)| \leq \mathbf{negl}(\lambda) \tag{2}$$

where $\mathbf{negl}(\lambda)$ is a negligible function in $\lambda$.

Informally, forward privacy requires that an update operation does not reveal whether the newly added data object relates to previous search objects. In this paper, we propose a definition of forward privacy for high-dimension data by extending the definition in [7].

**Definition 5** (Forward privacy): An SSE scheme for high-dimension data is forward private if the update leakage function $\mathcal{L}_{Update}$ can be formalized as:

$$\mathcal{L}_{Update} = \mathcal{L}'(qid, l) \tag{3}$$

where $qid$ denotes the identifier of the data object, $l$ denotes the number of hash tables and $\mathcal{L}'$ is stateless.

## 4 The Proposed Scheme

### 4.1 System Model

Figure 1 shows our proposed system model composed of two main participants, i.e., the medical client and the medical cloud server. Initially, on the medical client side, the client state $\sigma$ and the cloud server index $I$ are generated, then $I$ is sent to the cloud server. Then the update and search operations are performed separately by sending update token and search token. Specifically, when the medical client wants to perform an update operation, an update trapdoor $t_u$ is generated based on the client state $\sigma$, then $t_u$ is sent to the medical cloud server. The medical cloud server updates index $I$ and the encrypted dataset $C$ according to the received trapdoor $t_u$. Similarly, when the medical client wants to perform an update operation, an update trapdoor $t_s$ is generated based on the client state $\sigma$, then $t_s$ is sent to the medical cloud server. The medical cloud server gets the search result according to $t_s$ and index $I$, then returns the result to the medical client.

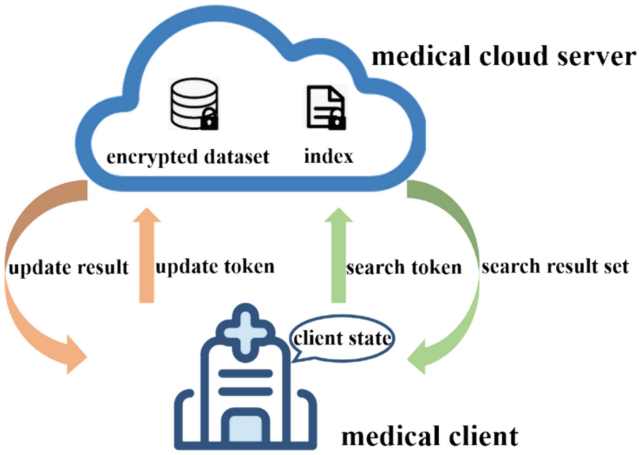


**Fig. 1.** System model of our scheme.

### 4.2 Storage Structure

In this paper, the core innovation is the storage structure of the index we designed. Therefore, before we introduce the complete execution flow of the system, we give a detailed introduction to the storage structure of the index. Essentially, we improve the structure of the traditional hash table to design a novel index that can guarantee forward privacy. Thus, we first analyze the security flaws of traditional hash tables and then introduce our index storage structure.

The traditional hash table storage structure is to store a list in each bucket. In previous SSE schemes for high-dimension data, the hash table is always stored in the cloud server as an index. When adding a data object $q$,$q$ is mapped to one of the buckets by LSH function, and then the ID of $q$, $qid$, is added to the list in this bucket. When searching similar objects of a data object $q$, we map $q$ with LSH function, map it to a bucket, and then the data objects in this bucket's list are returned as similar objects of $q$. With such storage structure, both query and update operations are mapped to a bucket first, so it is easy for the server to associate the newly added data object with search objects that were previously mapped to the same bucket for searching. Therefore, such a storage structure cannot provide forward privacy guarantee.

We designed a new storage structure that can provide forward privacy. Our storage structure is transformed from the traditional hash table storage structure as shown in Fig. 2. We transform by two steps: separate and merge. First, we separate the hash table from the list in each bucket, which means that the list is no longer stored in each bucket and only a state value information is stored. Then, we merge the lists that existed in each bucket. If we consider the list in each bucket as a small list and the merged list as a large list, then the large list contains all the elements stored in all the original small lists. In our scheme, we store the hash table locally as part of the client state value $\sigma$ and store the large list as the index $I$ in the cloud.

When performing the update operation, the data object $q$ to be updated is locally mapped to a bucket of the hash table by using LSH, the information in the bucket is

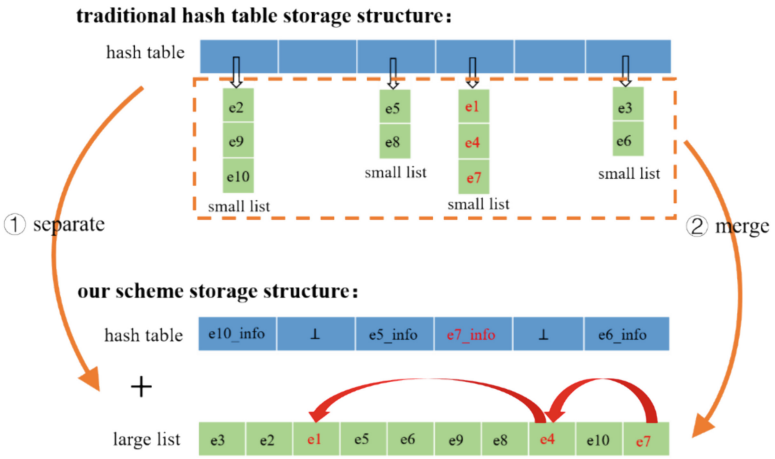


**Fig. 2.** An example of our storage structure.

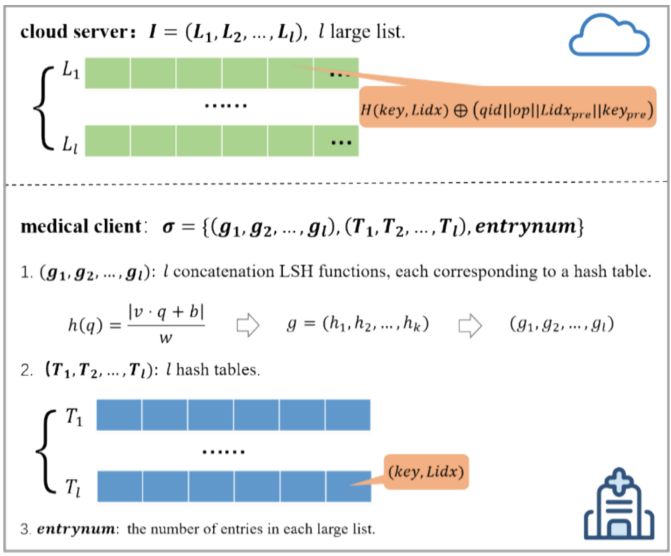


**Fig. 3.** Index of our scheme.

modified, and a new large list element is generated for $q$ according to the old and new information stored in the bucket. Then this element is sent to the cloud and added to the end of the large list. When performing the search operation, the search object $q$ is locally mapped to a bucket of the hash table by using LSH, and the search token $t_s$ for $q$ is generated based on the information stored in this bucket. Then $t_s$ is sent to the cloud server. The server locates an element of the large list according to the $t_s$ and then locates the next element according to the value of this element until the locating cannot be continued, and then it returns the all data objects traversed as the search result. Because

our update operation is to add an encrypted element directly to the end of the large list, the server cannot associate the update object with any previous search. Thus, the storage structure we design can provide forward privacy.

### 4.3 Our Construction

**Setup.** Algorithm 1 gives the description of setup phase. It takes as input a security parameter λ and LSH family, selects the symmetric encryption key $key_q$ used to encrypt data objects, determines the concatenation LSH functions $(g_1, g_2, \ldots, g_l)$, and initializes $l$ hash tables $(T_1, T_2, \ldots, T_l)$, $l$ large lists $(L_1, L_2, \ldots, L_l)$ and a variable *entrynum* that represents the number of elements in each large list. Each bucket of the newly initialized hash tables contains a null tuple pair $(\perp, \perp)$. For $\forall i \in [1, l]$, a one-to-one mapping exists between $g_i$ and $T_i$, and also exists between $T_i$ and $L_i$. The encryption key $key_q$ and client state $\sigma = \{(g_1, g_2, \ldots, g_l), (T_1, T_2, \ldots, T_l), entrynum\}$ are stored locally, The cloud index $I = (L_1, L_2, \ldots, L_l)$ and the data set $C$ used to hold encrypted data objects are sent to the cloud server. We consider $\sigma$ and $I$ as the index of our scheme, as shown in Fig. 3.

Algorithm 1 Setup

**Input**: $(\lambda, \text{LSH family}); \perp$
**Output**: $(\sigma, key_q); (I, C)$

$i \leftarrow 0$
**Repeat**:
$j \leftarrow 0$
  **Repeat**:
    $h \xleftarrow{\$} \text{LSH family}$
    $g_i \leftarrow g_i \cup h$
  **Until** $j = k$
  $T_i \leftarrow$ empty hash table
  $L_i \leftarrow$ empty list
  $i \leftarrow i + 1$
$entrynum \leftarrow 0$
$\sigma = \{(g_1, g_2, \ldots, g_l), (T_1, T_2, \ldots, T_l), entrynum\}$
$key_q \leftarrow (0,1)^{\lambda}$
$I = (L_1, L_2, \ldots, L_l)$
$C \leftarrow \emptyset$
Send $(I, C)$ to the cloud server

**Update.** A detailed description of update phase is provided in Algorithm 2. To update a data object $q$ with identiffier $qid$, for $\forall i \in [1, l]$, $q$ is mapped to a bucket of $T_i$ through $g_i$. The tuple pair in each bucket is a secret key *key* of the keyed hash function $H$ and an index value *Lidx* of the large list in the cloud. Then the update trapdoor $t_u$ is generated according to the tuple pairs in the buckets being mapped and is sent to the cloud server. Notice that $(e_1, e_2, \ldots, e_l)$ in $t_u$ are elements to be added to the large lists. Therefore, after receiving the update trapdoor $t_u$, the server adds these elements to the end of $l$ large lists in turn.

Algorithm 2 Update

**Input**: $(K, \sigma, q, op); (I, C)$
**Output**: $(\sigma', r_u); (I', C')$
On the client side:
$i \leftarrow 0$
$t_u \leftarrow \emptyset$
$entrynum \leftarrow entrynum + 1$
**Repeat**:
  $Tidx \leftarrow g_i(q)$
  $(key_{pre}, Lidx_{pre}) \leftarrow T_i[Tidx]$
  key $\xleftarrow{\$} (0,1)^{\lambda}$
  $Lidx \leftarrow entrynum$
  $T_i[Tidx] \leftarrow (key, Lidx)$
  $e_i \leftarrow H(key, Lidx) \oplus \left(qid||op||Lidx_{pre}||key_{pre}\right)$
  $t_u \leftarrow t_u \cup e_i$
  $i \leftarrow i + 1$
**Until** $i = l$
$q_c \leftarrow Enc(key_q, q)$
$t_u \leftarrow t_u||q_c$
Send $t_u$ to the cloud server
On the cloud server side:
Parse $t_u$ into $(q_c, (e_1, e_2, \ldots, e_l))$
$C \leftarrow C \cup q_c$
$i \leftarrow 0$
**Repeat**:
  $L_i$.add($e_i$)
  $i \leftarrow i + 1$
**Until** $i = l$
$r_u \leftarrow$ "$success$"
Send $r_u$ to the client

**Search.** In Algorithm 3, the search phase is presented. To search similar data objects for data object $q$, for $\forall i \in [1, l]$, $q$ is mapped to a bucket of $T_i$ through $g_i$. The search trapdoor $t_s$ is generated with the tuple pairs in the buckets being mapped. Then $t_s$ is sent to the cloud server. After receiving the search trapdoor $t_s$, the server can locate an element based on the index value $Lidx_i (1 \leq i \leq l)$. The data object and another index information ($Lidx_{pre}$, $key_{pre}$) hidden in this element are taken out. The server continues the locating operation based on ($Lidx_{pre}$, $key_{pre}$) until the extracted location information is null tuple pair ($\perp$, $\perp$). In this way, the server collects similar data objects scattered in large lists and returns them to the client as a result of the search operation.

Algorithm 3 Search

**Input**: $(K, \sigma, q); (I, C)$
**Output**: $r_s; \perp$
On the client side:
$i \leftarrow 0$
$t_s \leftarrow \emptyset$
**Repeat**:
  $Tidx \leftarrow g_i(q)$
  $(key_i, Lidx_i) \leftarrow T_i[Tidx]$
  $t_s \leftarrow t_s \cup (key_i, Lidx_i)$
  $i \leftarrow i + 1$
**Until** $i = l$
Send $t_s$ to the cloud server
On the cloud server side:
Parse $t_s$ into $(key_1, Lidx_1), \ldots, (key_l, Lidx_l)$
$i \leftarrow 0$
$r_s \leftarrow \emptyset$
**Repeat**:
  $key \leftarrow key_i$
  $Lidx \leftarrow Lidx_i$
  **Repeat**:
    $e \leftarrow L_i(Lidx)$
    $mask \leftarrow H(key, Lidx)$
    $(qid||op||Lidx_{pre}||key_{pre}) \leftarrow e \oplus mask$
    $r_s \leftarrow r_s \cup qid$
    $key \leftarrow key_{pre}$
    $Lidx \leftarrow Lidx_{pre}$
  **Until** $Lidx = \perp$
  $i \leftarrow i + 1$
**Until** $i = l$
Send $r_s$ to the client

## 5 Analysis

### 5.1 Security Analysis

**Theorem 1** (Adaptive security)**:** Our scheme with leakage functions $\{\mathcal{L}_{Setup}, \mathcal{L}_{Update}, \mathcal{L}_{search}\}$ is adaptive secure, where $\mathcal{L}_{Setup} = \mathcal{L}'(l)$, $\mathcal{L}_{Update} = \mathcal{L}'(qid, l)$, $\mathcal{L}_{Search} = \mathcal{L}'(r_s)$.

**Proof:** We give the proof of Theorem 1 through games below.

**Game$_{\text{Setup}}$:** The simulator $\mathcal{S}$ generates a randomized index $\tilde{I}$ based on $\mathcal{L}_{Setup}$ and $\mathcal{L}_{Update}$, which is the same size as the real encrypted index $I$. $\tilde{I}$ consists of $l$ lists, and each element in $\tilde{I}$ is a random string with the same length as the real encrypted element. Adversary $\mathcal{A}$ cannot differentiate $\tilde{I}$ from $I$ because of the semantic security of secure symmetric encryption.

**Game$_{\text{Update}}$:** $\mathcal{A}$ Generates random strings as simulated token $\widetilde{t_u}$ when the update object $q$ is sent. Then a random oracle $H$ makes some modifications to the index $\tilde{I}$ based on $\widetilde{t_u}$. Since secure symmetric encryption is semantically secure, $\widetilde{t_u}$ is not computationally indistinguishable from $t_u$, the index modified based on $\widetilde{t_u}$ and the index modified based on $t_u$ cannot be distinguished.

**Game$_{\text{Search}}$:** $\mathcal{A}$ Generates random strings as simulated token $\widetilde{t_s}$ when the search object $q$ is sent. Then a random oracle $H$ gets the results $\widetilde{r_s}$ based on $\widetilde{t_s}$. The term $\widetilde{r_s}$ is identical to the real result $r_s$ indicated in $\mathcal{L}_{Search}$. Because secure symmetric encryption is semantically secure, $\widetilde{t_s}$ is not computationally indistinguishable from $t_s$. Similarly, the results derived from $\widetilde{t_s}$ are the same as the real results. Thus, $\mathcal{A}$ cannot differentiate between simulated $\widetilde{t_s}$ generated by $\mathcal{S}$ and real $t_s$ or between simulated $t_s$ and real $\widetilde{r_s}$.

**Conclusion:** Summarizing the above game, we can say that for all probabilistic polynomial time adversaries $\mathcal{A}$, there exists a probabilistic polynomial time simulator $\mathcal{S}$, such that:

$$|\Pr\left(\mathbf{Real}_{\mathcal{A}}^{\Omega}(\lambda) = 1\right) - \Pr(\mathbf{Ideal}_{\mathcal{A}}^{\Omega}(\lambda) = 1)| \leq \mathbf{negl}(\lambda)$$

Thus, our proposed scheme is adaptive secure.

**Forward Privacy.** As described in Sect. 4, we split the hash table and lists and merge all small lists into a large list. Moreover, we store the hash table locally and store the large list in the cloud. Since the search trapdoor $t_s$ for $q$ is generated from the bucket that $q$ mapped to and this bucket updates once a newly added data object $p$ that also maps to this bucket is added to the encrypted dataset in the cloud, the server cannot know the search trapdoor for $p$ until the next query that finds it appears. In other words, the server cannot observe which query request is associated with the update object. Therefore, our proposed scheme is forward secure.

### 5.2 Performance Analysis

We implement our scheme in Python. The experiments run on a machine with an Intel Core i7-8700U, 3.20 GHz processor, 8 GB RAM, and a 240 GB SSD disk.

The characteristics of the three datasets we used are summarized in Table 1. Moreover, Table 1 shows the index size of each dataset. Specifically, we show the size of the hash table and the size of the lists.

Figure 4 shows the update time. The update time is taken as the average of the update times of all data objects in the dataset. As we can see, the update time is linear with the number of hash tables. Further, we analyze the update time complexity from Algorithm 2 as $O(l)$, where $l$ is the number of hash tables.

Figure 5 shows the search time. We can analyze the search time complexity is $O(r)$, where $r$ is the number of data points in the query result set. This is consistent with the experimental result of Fig. 5.

**Table 1.** The characteristics of datasets.

| Dataset | Dimension | Number of data objects | Size of hash tables | Size of lists |
|---|---|---|---|---|
| Epileptic | 178 | 11500 | 32 KB | 2752 KB |
| HCV | 29 | 1385 | 32 KB | 332 KB |
| Parkinson | 22 | 5875 | 32 KB | 1406 KB |

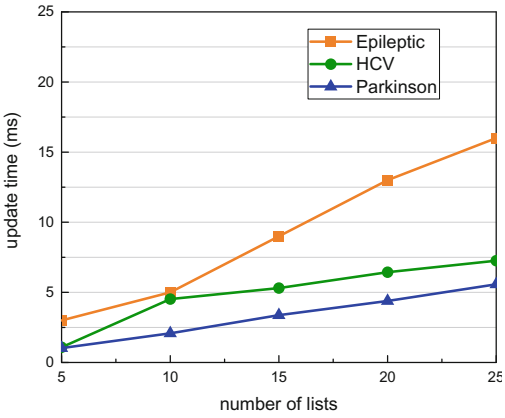


**Fig. 4.** Update performance.

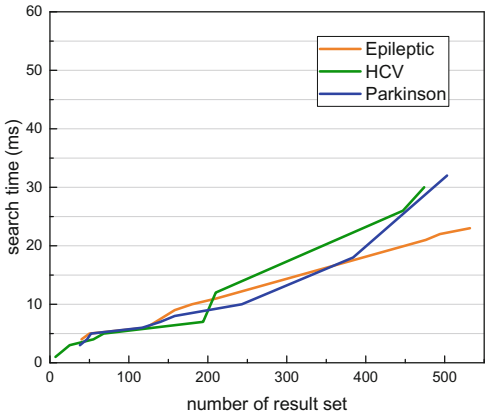


**Fig. 5.** Search performance.

## 6 Conclusion

Medical cloud storage is growing popularity and the demand for cloud security continues to increase. Thus, we proposed a novel forward secure scheme for high-dimension medical data in cloud server. We prove it is adaptive secure, and the experimental results demonstrate it is efficient in practice.

**Acknowledgment.** This paper is supported by the National Science Foundation of China under grant No. 61871064, 61501080, and 61771090 the Fundamental Research Funds for the Central Universities under No. DUT19JC08, and the China Postdoctoral Science Foundation under grant No. 2019M661097.

## References

1. Song, D.X., Wagner, D.A., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE Symposium on Security and Privacy, Berkeley, California, USA, 14–17 May 2000, pp. 44–55 (2000)
2. Yang, Y., Ma, M.: Conjunctive keyword search with designated tester and timing enabled proxy re-encryption function for E-health clouds. IEEE Trans. Inf. Forensics Secur. **11**, 746–759 (2016)
3. Li, H., Yang, Y., Dai, Y., et al.: Achieving secure and efficient dynamic searchable symmetric encryption over medical cloud data. IEEE Trans. Cloud Comput. (2017). https://doi.org/10.1109/TCC.2017.2769645
4. Zhang, Y., Xu, C., Li, H., et al.: HealthDep: an efficient and secure deduplication scheme for cloud-assisted e-health systems. IEEE Trans. Industr. Inf. **14**, 4101–4112 (2018)
5. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, 23–26 May 1998, pp. 604–613 (1998)
6. Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005). https://doi.org/10.1007/11496137_30
7. Bost, R.: $\sum o\varphi o\varsigma$: forward secure searchable encryption. In: Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016, pp. 1143–1154 (2016)
8. Kim, K.S., Kim, M., Lee, D., et al.: Forward secure dynamic searchable symmetric encryption with efficient updates. In: Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–03 November 2017, pp. 1449–1463 (2017)
9. Etemad, M., et al.: Efficient dynamic searchable encryption with forward privacy. In: Proceedings on Privacy Enhancing Technologies, vol. 2018, pp. 5–20 (2018)
10. Ghareh Chamani, J., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018, pp. 1038–1055 (2018)
11. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of file-injection attacks on searchable encryption. In: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, 10–12 August 2016, pp. 707–720 (2016)

12. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Conference on Computer and Communications Security, Alexandria, VA, USA, 30 October–3 November 2006, pp. 79–88 (2006)
13. Kamara, S., Papamanthou, C., et al.: Dynamic searchable symmetric encryption. In: Conference on Computer and Communications Security, Raleigh, NC, USA, 16–18 October 2012, pp. 965–976 (2012)