

# Dynamic Similarity Search over Encrypted Data with Low Leakage

Daniel Homann<sup>(✉)</sup>, Christian Göge, and Lena Wiese

Institut für Informatik, Universität Göttingen,  
Goldschmidtstraße 7, 37077 Göttingen, Germany  
{homann,christian.goege,wiese}@cs.uni-goettingen.de

**Abstract.** Though cloud databases offer advantages in terms of maintenance cost, they require encryption in order to protect confidential records. Specialized searchable encryption schemes are needed to provide the functionality of privacy preserving search on encrypted data. In many use cases, a search which also returns the correct documents when the search term was misspelled is very desirable. Therefore, we present a novel similarity searchable encryption scheme. Our scheme uses symmetric encryption primitives, is dynamic, i.e. allows the efficient addition and deletion of search terms and has sub-linear search cost. We prove that the leakage of our scheme is low and that it provides forward security. Our scheme is built by employing a new construction technique for similarity searchable encryption schemes. In this construction a searchable encryption scheme is used as storage layer for a similarity searchable encryption scheme.

**Keywords:** Similarity search · Fuzzy search · Searchable encryption · Symmetric encryption · Cloud databases

## 1 Introduction

Outsourcing data to cloud providers is becoming more and more common. While this offers great advantages in terms of maintenance cost, the confidentiality of the data is in peril. One solution to this problem is the use of encryption. While standard symmetric encryption algorithms like e.g. AES offer good security by encrypting all records, the cloud data store loses any search functionality and degrades to just a bunch of encrypted files. However, among the desirable functionality of a database is the ability to efficiently perform full-text search.

This paper deals with the problem of efficient searching in encrypted cloud databases. Efficiency means to perform the searches in less than linear time in the number of document-keyword pairs. For searching encrypted data several algorithms have been proposed. They can be divided in symmetric and public-key solutions. Public-key searchable encryption schemes offer multi-user capabilities but they require computationally costly public-key operations. Therefore, it appears to be more practical to pursue symmetric key searchable encryption.

Users sometimes misspell search terms. However, in many use cases it is very desirable that small spelling errors still lead to the correct results. One solution for this is checking the keyword in a dictionary before searching. This makes the search more expensive on the client-side and adds additional complexity by requiring the client to decide between several correct words which are within a certain distance of the entered term. Furthermore, the client might want to search for a keyword that is not contained in the dictionary, like a specialist term or a name.

To overcome these problems, we propose a new similarity searchable encryption scheme. Our scheme has the following key properties:

1. It is dynamic, i.e. it allows to add new documents and keywords efficiently. The amortized runtime of adding a document with  $m$  keywords to an index of size  $N$  is  $\mathcal{O}(m \cdot \lambda \cdot N \cdot \log^2 N)$ . Here  $\lambda$  is an encoding dependent constant.
2. It has a small leakage and provides forward security, i.e. when adding a new document it does not reveal which of the old documents contains similar words to the new document.
3. The search takes sub-linear time in the number of document-keyword pairs. Its runtime is  $\mathcal{O}(\lambda \min \{\beta + \log N, \gamma \log^3 N\})$ , where  $\gamma$  is the number of documents which are similar to the search term and  $\beta$  is the number of historically added, but perhaps deleted, documents similar to the search term.
4. It supports similarity search (also known as: fuzzy search). This means it also finds documents containing slightly misspelled keywords.

To the best of our knowledge no other existing searchable encryption scheme provides this desirable combination of properties. Furthermore our scheme is defined in such a way, that it is not restricted to the problem domain of fuzzy text search. By specifying an appropriate family of locality sensitive hash functions it could be applied to other domains e.g. biometric data.

Our scheme achieves these properties by combining two existing searchable encryption schemes. The first scheme is a secure and dynamic searchable encryption scheme without similarity search capabilities [19], while the second scheme is a non-dynamic similarity search scheme [14]. They are combined in such a way that the former acts as a storage layer for the latter. This technique of combining a non-fuzzy and a fuzzy searchable encryption scheme, seems very powerful to us, as it allows to build very good similarity searchable encryption schemes from existing searchable encryption schemes.

In the next section, we will describe the state of the art regarding symmetric searchable encryption. In Sect. 3, we will give a formal security definition for our scheme. Our scheme for general similarity search is described in Sect. 4. In Sect. 5 we explain how to apply this general scheme towards similarity search for text. We prove that our scheme fulfils the security definition in Sect. 6. In the following sections, we will give an experimental analysis of our scheme (Sect. 7) and draw a conclusion (Sect. 8).

## 2 Background

### 2.1 Symmetric Searchable Encryption (SSE)

For symmetric searchable encryption there is a wide range of proposed algorithms (for an overview see [2]). They all consist of at least two different protocols. The first protocol generates a secure index structure from a set of *documents*. Each of these documents contains a set of *features* (e.g. words). When searching for a feature, the server returns the *identifiers* of all documents containing this feature.

The first searchable symmetric encryption schemes were given by [4, 6, 8, 18]. Most of the early schemes were not dynamic, i.e. when adding a new document-keyword pair the complete index had to be rebuilt from scratch. The runtime of this rebuild operation is at least linear in the number of document-keyword pairs, since every document-keyword pair has to be processed at least once. As a result, index changes in such schemes might be possible but incur large performance penalties. Furthermore, during the rebuild the index is usually stored on the client meaning that the index may not become larger than the client's storage. In typical Big Data applications, the amount of data stored on the server is much larger than the storage of a single client. As a result, in such use cases a rebuild would not only be inefficient but simply impossible.

The security of searchable encryption schemes is usually evaluated in a client-server setting with a honest-but-curious server as adversary. Its security is then quantified by the amount of information that is *leaked* to the server, i.e. that the server could compute in polynomial-time from its observation of the protocol execution. The common security definitions were first given by [6].

*Dynamic* schemes allow for the addition of new document-keyword pairs at runtime with lower rebuilding cost. In the following, we will consider schemes to be dynamic if the amortized rebuilding cost is sub-linear in the number of document-keyword pairs. Of the proposed dynamic SSE schemes [3, 10, 12, 13, 18, 19], the best security properties are offered by the scheme of [19], as it provides forward security. Dynamic schemes are called *forward secure* if they do not leak to an attacker if an added document contains a keyword which was previously searched for. A dynamic scheme is called *backward secure* if an already deleted document containing feature  $w$  is not contained in the leakage of a search for feature  $w$ . To our knowledge, no existing SSE scheme achieves backward security.

Recent research highlights the importance of forward security [26]. They showed in a modified threat model, that SSE schemes without forward security are more vulnerable to attacks. In their threat model the server is able to inject files in the index, but otherwise behaves in an honest-but-curious manner.

### 2.2 Similarity Searchable Symmetric Encryption (SSSE)

Often spelling errors occur while typing search queries. Nevertheless, the correct result set should be found. To achieve this, the notion of searchable symmetric encryption was extended to similarity searchable symmetric encryption.

Similarity search can be understood in two ways. In the following, we will use the term similarity search as the search for a single feature (e.g. word), which may be slightly altered (e.g. misspelled) but nevertheless should be found. In the literature, a search returning documents which contain a subset of a given feature set is also called similarity search [17, 20, 21, 24].

In the following, we will only consider the first meaning of similarity search. Known SSSE schemes in this sense are given by [1, 5, 11, 14–16, 22, 23]. While searching misspelled words in larger bodies of text is the most important application, similarity search can also be applied to other problem domains, e.g. similarity of biometric data [1]. An important property for the classification of SSSE schemes is their generality: They can either be focused on a single problem domain (e.g. text search) or be of general use. In the latter case, they can be adopted to a specific domain by a problem specific similarity mapping.

Very specific to the problem domain of searching text are schemes using wild cards letters [11, 15, 16, 23]. Such schemes were first sketched in [15] and described in more detail in [23]. Several modifications of this approach were proposed [11, 16]. The disadvantage of all these SSSE is that they do not allow dynamic updates of the index.

The scheme of [14] is of much greater generality. It encodes features in Bloom filters. Afterwards, it employs locality sensitive hashing (LSH) on the Bloom filter and stores the generated *subfeatures* in an index structure. By specifying an encoding to Bloom filters and choosing a suitable LSH family, any kind of data, i.e. not just text, can be subject to similarity search. However, the main disadvantage of this scheme is that data cannot be added dynamically.

In [5] the authors also encode words in Bloom filters but store the index in a tree-based structure. Their scheme is dynamic but has a very high leakage, as the similarity between all documents is leaked even if no search operations have been performed. Another scheme given by [22] also uses Bloom filters. Searches are not efficient in this scheme, as the runtime of search operations is linear in the number of stored documents.

The authors of [1] provide a definition of SSSE which is very strict with regard to result quality and permissible leakage. They show that no space-efficient scheme satisfying their definition can exist. For a slightly relaxed version of their definition they give a SSSE scheme for fingerprint data which is still quite space-inefficient.

In the introduction of [25] which addresses similarity of entire documents, a similar scheme to the one we describe here is briefly considered but dismissed because it “does not achieve practical efficiency” for their use case. However, for our use case our scheme achieves a competitive asymptotic search and update runtime. This will be shown in Sect. 4.3.

### 3 Security Definition

Our scheme is a set of protocols executed between a client (user) and a server. The server stores the index structure of the scheme and is queried by the client.

The client is a trusted computing device while the server is untrustworthy. As in the standard security model of SSE, we consider the server to be honest-but-curious. Thus, the server might inspect the request and the stored data and try to deduce information from it but otherwise follows the prescribed protocol. This section adapts previous work [6, 14, 19] to our setting. In order to give a formal security definition we first define our understanding of a dynamic similarity searchable encryption scheme.

**Definition 1 (DSSSE).** *A dynamic similarity searchable symmetric encryption (DSSSE) is given by a set of three protocols executed between a client and a server:*

- *Setup( $1^\kappa$ ): Create an empty data structure with security parameter  $\kappa$  on the server.*
- *Search( $w, k$ ): Based on feature  $w$ , the client calculates a matrix of trapdoors  $T$  and sends them to the server. The server executes the search and returns an encrypted result vector. The client decrypts the vector and returns the identifiers of the top- $k$  results.*
- *Update( $D, id, op$ ): If  $op = \text{add}$ , then the server adds the features of document  $D$  with identifier  $id$  to the index, otherwise ( $op = \text{del}$ ) it deletes the keywords of document  $D$  with identifier  $id$  from the index.*

In contrast to non-dynamic searchable encryption schemes, our scheme does not need a protocol that builds an index from a document collection as the index can be build step-by-step by adding documents with the update protocol.

The concept of the history captures the operation of the scheme from the client’s point of view. The definition of the history from [6] has to be changed to account for the possible dynamic addition of documents.

**Definition 2 (History).** *Between a client and a server a DSSSE is initialized via the setup protocol. After a total number of  $n$  executions of the search and update protocols the history  $\mathcal{H}_n$  is given by:*

$$\mathcal{H}_n = (h_1, \dots, h_n).$$

Let  $S_i$  be a data structure describing which feature is associated to which document identifier after the execution of the  $i$ -th step. The  $h_i$  are defined depending on the type of protocol executed in the  $i$ -th step. If the  $i$ -th operation was a search operation, then  $h_i = (S_i, w_i)$ , where  $w_i$  is the feature that was searched for. Otherwise  $h_i = (S_i, (D_i, id_i, op_i))$  where  $(D_i, id_i, op_i)$  are the parameters of the executed update protocol.

Next we want to define which of this information is leaked to the server. The different types of leakage to the server are given by the following five definitions.

**Definition 3 (Protocol Pattern).** *For a history  $\mathcal{H}_n$  the protocol pattern  $\mathcal{P}_i$  is a vector given by:*

$$\mathcal{P}_i = \begin{cases} 1 & \text{if the } i\text{-th step is a search operation} \\ 0 & \text{else} \end{cases} \quad \forall 1 \leq i \leq n$$

**Definition 4 (Search Pattern).** For every search protocol execution  $h_i$  the search pattern is given by a vector  $\mathcal{F}$ :

$$\mathcal{F}_j = \begin{cases} 1 & \text{if } \mathcal{P}_j = 1 \text{ and } w_j = w_i \\ 0 & \text{else} \end{cases} \quad \forall 1 \leq j < i$$

**Definition 5 (Access Pattern).** Let  $h_i$  be a search protocol execution. Let  $w_i^1, \dots, w_i^\lambda$  be the subfeatures of  $w_i$ . Then the access pattern is given by:

$$\mathcal{A} = (\mathcal{A}_1, \dots, \mathcal{A}_\lambda)$$

where  $\mathcal{A}_j$  are the identifiers of documents added or removed in the past containing the subfeature  $w_i^j$ .

**Definition 6 (Similarity Pattern).** Let  $h_i$  be a search protocol execution. Let  $w_i^1, \dots, w_i^\lambda$  be the subfeatures of  $w_i$ . Then the similarity pattern  $\mathcal{S}$  is given by:

$$\mathcal{S}_{a,j,b} = \begin{cases} 1 & \text{if } \mathcal{P}_j = 1 \text{ and } w_i^a = w_j^b \\ 0 & \text{else} \end{cases} \quad \forall 1 \leq j < i, \forall 1 \leq a, b \leq \lambda$$

**Definition 7 (Update Pattern).** For every update operation  $h_i$  the update pattern  $\mathcal{U}$  is given by:

$$\mathcal{U} = (op_i, id_i, |D_i|)$$

where  $|D_i|$  is the number of subfeatures in  $D_i$ .

With these patterns we could now define the trace which is the maximal amount of information the adversary (server) should be able to compute from the leaked information.

**Definition 8 (Trace).** The trace  $\mathcal{T}(\mathcal{H}_n)$  for a history  $\mathcal{H}_n$  consists of the protocol pattern  $\mathcal{P}$  and for each search operation the search pattern  $\mathcal{F}$ , access pattern  $\mathcal{A}$  and similarity pattern  $\mathcal{S}$  and for each update operation the update pattern  $\mathcal{U}$ .

Our definition of the update pattern does not leak whether a feature of the updated document  $D$  was searched for previously. Therefore, a scheme with a trace as given above provides forward security. Since deleted documents which match the search term are still included in the access pattern, this definition of the trace does not provide backward security.

For a given history  $\mathcal{H}_n$ , the information that is leaked to the server is called the view and denoted by  $\mathcal{V}(\mathcal{H}_n)$ . We want that the server in our scheme cannot deduce more information than the trace  $\mathcal{T}(\mathcal{H}_n)$  from this view. Therefore, the following security definition should hold for our scheme. We will prove that it indeed satisfies this security definition in Sect. 6.

**Definition 9 (Adaptive Semantic Security for DSSSE).** A scheme provides adaptive semantic security if one can define a simulator  $S$  such that for all polynomial size distinguishers  $D$  and for all polynomials  $p$  and a large  $r$  holds:

$$\mathbb{P}[D(\mathcal{V}(\mathcal{H}_n)) = 1] - \mathbb{P}[D(S(\mathcal{T}(\mathcal{H}_n))) = 1] < \frac{1}{p(r)}$$

with probabilities taken over  $\mathcal{H}_n$  and the coins of the scheme's key generation and encryption.

## 4 Our Scheme

Our scheme combines the dynamic SSE of Stefanov et al. [19] with the similarity search of Kuzu et al. [14]. This way, the scheme achieves very good security (see Sect. 3) and an efficient similarity search at the same time. In this section we will describe our scheme in detail.

### 4.1 Preliminaries

Our construction is based on the following primitives. Oblivious sorting and the LSH family are described below in more detail.

- A function  $\text{keygen}(\kappa)$  which generates a new, random symmetric encryption key, the length of which is determined by the security parameter  $\kappa$ .
- Symmetric (probabilistic) encryption and decryption functions  $\text{Encrypt}$  and  $\text{Decrypt}$ .
- Symmetric, deterministic encryption and decryption functions  $\text{DetEncrypt}$  and  $\text{DetDecrypt}$ .
- A keyed hash function  $H_{\text{key}}$ .
- A random oracle  $H_{\text{key}}^*$ . For the security proof we need this keyed hash function to be modelled as random oracle (see Sect. 6).
- A hash function  $h$ .
- An oblivious sorting protocol  $\text{o-sort}$ .
- A metric space embedding  $\rho$  which maps features  $w$  into a metric space  $F$ .
- A  $(r_1, r_2, p_1, p_2)$ -sensitive LSH family  $G = (g_i)_{1 \leq i \leq \lambda}$ .

*Oblivious Sorting.* With the oblivious sorting protocol, the client sorts the data stored on the server in such a way that the server remains oblivious about the order of the items. In each step of this protocol the client downloads a chunk of  $\mathcal{O}(N^\alpha)$  entries from the server, decrypts them for sorting and afterwards uploads them in encrypted form to the server. The parameter  $0 < \alpha \leq 1$  can be chosen so that the chunk still fits in the client's memory. We used the oblivious  $k$ -way mergesort algorithm from [9] as this algorithm was specifically created for external oblivious sorting.

*LSH Family.* Let  $\text{dist}$  be a metric on a metric space  $F$ . Then a family of hash functions  $G$  is called  $(r_1, r_2, p_1, p_2)$ -sensitive if for any features  $x, y \in F$  and for any  $g \in G$  holds:

- if  $\text{dist}(x, y) \leq r_1$  then  $\mathbb{P}[g(x) = g(y)] \geq p_1$
- if  $\text{dist}(x, y) \geq r_2$  then  $\mathbb{P}[g(x) = g(y)] \leq p_2$ .

An LSH family of a desired sensitivity can be constructed from another LSH family with a different sensitivity as a result of an AND- and OR-construction of this existing family. For further details see [14]. How to choose the metric space embedding  $\rho$  and the LSH family  $G$  for text search is given in Sect. 5.

The following parameters of our scheme can be chosen by a user to customize it to a certain setting:

- $\kappa$  Security parameter defining the length of the symmetric encryption keys.
- $\alpha$  This value is used to adapt the scheme to available storage on the client.
- $\rho$  Mapping of the features to a metric space depending on the problem's domain.
- $G$  A family of LSH functions. The value of  $\lambda = |G|$  is very critical for the scheme's performance as well as storage demand.

## 4.2 Detailed Construction

The index structure of our scheme consists of  $L = \lceil \log_2 N \rceil + 1$  hash maps  $\mathbb{H}_i$  of size  $2^i$  for  $0 \leq i \leq L$  which are stored on a server. Here  $i$  denoted the level of an entry in the data structure. Added or deleted document-subfeature pairs will always be added to the first empty hash map. For each hash map  $\mathbb{H}_i$  the client possesses a symmetric encryption key  $k_i$ .

The scheme consists of the three protocols Setup (Protocol 1), Update (Protocol 2) and Search (Protocol 4). While the Update protocol sometimes requires several roundtrips between the client and the server, the Setup and Search protocols are non-interactive (1-round protocol). Furthermore, there are two helper functions for encoding (Algorithm 5) and searching (Algorithm 6) entries in the hash map structure. During updates an additional Rebuild protocol (Protocol 3) is used to partially rebuild the hash map structure.

---

### Protocol 1. Setup(Security parameter $\kappa$ )

---

```
esk  $\leftarrow$  keygen( $\kappa$ )
 $L \leftarrow 0$ 
 $\mathbb{H}_0 \leftarrow \emptyset$ 
```

---



---

### Protocol 2. Update(document $D$ , identifier id, operation op)

---

```
 $V \leftarrow \emptyset$ 
for all features  $w \in D$  do
   $\vec{w} \leftarrow \rho(w)$ 
  for all  $g \in G$  do
     $V \leftarrow V \cup \{g(\vec{w})\}$ 
for all  $v \in V$  in random order do
  if  $\mathbb{H}_0$  is empty then
     $k_0 \leftarrow \text{keygen}(\kappa)$ 
     $\mathbb{H}_0 \leftarrow \text{encodeEntry}(v, \text{id}, \text{op}, 0, \text{esk}, k_0)$ 
  else
    Execute protocol: Rebuild( $v, \text{id}, \text{op}$ )
```

---

*Setup (Protocol 1).* The Setup algorithm generates a encryption key  $\text{esk}$ , initializes the number of levels  $L$  with 0 and creates an empty hash map  $\mathbb{H}_0$  for level 0.



**Protocol 3.** Rebuild(subfeature  $v$ , identifier  $id$ , operation  $op$ )

---

```

 $l_{\text{new}} \leftarrow$  Smallest  $l$  for which  $\mathbb{H}_l$  is empty
Set  $e^* \leftarrow \text{encodeEntry}(v, id, op, 0, \text{esk}, k_0)$  and store  $e^*$  on the server
Let  $B$  denote  $\{e^*\} \cup \bigcup_{i=0}^{l-1} \mathbb{H}_i$ 
for all  $e = (hkey, c_1, c_2) \in B$  on server do
     $(v, id, op, cnt) \leftarrow \text{Decrypt}_{\text{esk}}(c_2)$ 
    Replace  $e$  on server with  $\text{Encrypt}_{\text{esk}}(v, id, op, cnt)$ 
 $B \leftarrow \text{o-sort}(B)$  according to sorting key  $(v, id, op)$ 
for all  $e = \text{Encrypt}_{\text{esk}}(v, op, id, cnt) \in B$  on server do
    if  $e$  is the start of a new feature  $v$  for operation  $op$  then
         $cnt_{op,v} \leftarrow 0$ 
        Replace  $e$  on server with  $\text{Encrypt}_{\text{esk}}(v, id, op, 0)$ 
    if  $e$  and next entry in  $B$  are add and del operations for the same word then
        Replace each of them by  $\text{Encrypt}(\perp)$  on the server
    else
         $cnt_{op,v} \leftarrow cnt_{op,v} + 1$ 
        Replace  $e$  on server with  $\text{Encrypt}_{\text{esk}}(v, id, op, cnt_{op,v})$ 
Permute  $B$  randomly by  $B \leftarrow \text{o-sort}(B)$  according to sorting key  $hkey$ 
 $k_l \leftarrow \text{keygen}(\kappa)$ 
for all  $e \in B$  on the server do
     $(v, id, op, cnt) \leftarrow \text{Decrypt}_{\text{esk}}(e)$ 
    Add  $\text{encodeEntry}(v, id, op, cnt)$  to  $\mathbb{H}_l$  on server
Remove all entries from  $\mathbb{H}_i$  for  $i = 0, \dots, l - 1$ 

```

---

*Update (Protocol 2).* The Update algorithm is used to add or delete documents to the search scheme. An addition operation is specified by  $op = \text{add}$  and a deletion operation by  $op = \text{del}$ . The algorithm processes all features  $w$  of the document. These features are embedded in a metric space where the LSH family  $G$  is applied. The set of all these hash values is then shuffled randomly and inserted in the data structure on the server. During this process two cases have to be distinguished. If the hash map on the first level ( $\mathbb{H}_0$ ) is empty, the current hash will be inserted in this level. Otherwise a rebuild (see Protocol 3) of the data structure is required. This works as follows: First the first completely empty level  $l$  is determined. The rebuild will operate on the entries of the hash maps  $\mathbb{H}_i$  with  $i < l$  and the new entry  $e^*$ . For better readability we denote this set of entries by  $B$ . At the end of the protocols all the entries of  $B$  will be inserted in  $\mathbb{H}_l$  and the levels  $\mathbb{H}_i$  with  $i < l$  will be empty. In a for loop the encoding of the elements of  $B$  is changed for easier subsequent processing. This loop as well as the later for loop only requires constant storage on the client, as the entry, after processing, is stored again on the server. Now, all entries of  $\mathbb{H}_l$  are sorted according to  $(v, id, op)$ . The sorting employs oblivious sorting to not leak any information about the entries to the server. In Subsect. 4.1 we describe how this can be achieved using only sublinear client's storage. Then the  $cnt$  values of entries belonging to the same  $(v, id)$  tuple are enumerated in the sorted order.

**Protocol 4.** Search(feature  $w$ , number of results  $k$ )**Client:**

$$\vec{w} \leftarrow \rho(w)$$

$$t \leftarrow (g_i(\vec{w}))_{1 \leq i \leq \lambda}$$

$$T \leftarrow (H_{k_j}(h(t_i)))_{1 \leq i \leq \lambda, 0 \leq j \leq L}$$

Send  $T$  to server

**Server:**

$$C \leftarrow \emptyset$$

**for all**  $1 \leq i \leq \lambda$  **do**

**for all**  $l \in \{L, L-1, \dots, 0\}$  **do**

$\text{cnt} \leftarrow 0$

$\text{eid} \leftarrow \text{Lookup}(T_{i,l}, \text{add}, \text{cnt})$

**while**  $\text{eid} \neq \perp$  **do**

$C \leftarrow C \cup \{\text{eid}\}$

$\text{cnt} \leftarrow \text{cnt} + 1$

$\text{eid} \leftarrow \text{Lookup}(T_{i,l}, \text{add}, \text{cnt})$

$\text{cnt} \leftarrow 0$

$\text{eid} \leftarrow \text{Lookup}(T_{i,l}, \text{del}, \text{cnt})$

**while**  $\text{eid} \neq \perp$  **do**

$C \leftarrow C \setminus \{\text{eid}\}$

$\text{cnt} \leftarrow \text{cnt} + 1$

$\text{eid} \leftarrow \text{Lookup}(T_{i,l}, \text{del}, \text{cnt})$

**return**  $C$  to client

**Client:**

$$D \leftarrow \text{empty dictionary}$$

**for all**  $\text{eid} \in C$  **do**

$v, \text{id} \leftarrow \text{DetDecrypt}(c)$

**if**  $\text{id}$  not in  $C$  **then**

$D[\text{id}] \leftarrow 0$

$D[\text{id}] \leftarrow D[\text{id}] + 1$

**return** the identifiers from  $C$  with top- $k$  values

To remove complementary operations pairs of add and del operations belonging to the same  $(v, \text{id})$  tuple both of them are replaced with  $\perp$ . Afterwards, a new level key  $k_l$  is chosen to re-encode the entries.

When the client possesses enough storage to hold all the entries in  $B$  a simpler rebuild is possible by transferring all the entries of  $B$  to the client, decrypting them, performing a processing similar to line 8–16 in Protocol 3 locally and storing them in  $\mathbb{H}_l$  on the server again. (see [19] for details).

*Search (Protocol 4).* For the execution of the Search algorithm as described in Protocol 4 only a single round of interaction between the client and server suffices. In a first step the client embeds the search term in a metric space and applies the locality sensitive hash functions on this vector. Based on these hash values trapdoor values  $T$  are generated and sent to the server. The server starts with initializing an empty dictionary  $C$  where the found encrypted ids of possible

**Algorithm 5.** Algorithm for the generation of the hash map entries

---

```

procedure ENCODEENTRY( $v, id, op, cnt, esk, k_l$ )
  token  $\leftarrow H_{k_l}(h(v))$ 
  hkey  $\leftarrow H_{token}^*(0||op||cnt)$ 
   $c_1 \leftarrow \text{DetEncrypt}_{esk}(v||id) \oplus H_{token}^*(1||op||cnt)$ 
   $c_2 \leftarrow \text{Encrypt}_{esk}(v||id||op||cnt)$ 
  return (hkey,  $c_1, c_2$ )

```

---

**Algorithm 6.** Algorithm for the lookup of a certain entry in the hash maps

---

```

procedure LOOKUP(token, op, cnt)
  hkey  $\leftarrow H_{token}^*(0||op||cnt)$ 
  if hkey  $\in \mathbb{H}_l$  then
    return  $\mathbb{H}_l[hkey].c_1 \oplus H_{token}^*(1||op||cnt)$ 
  else
    return  $\perp$ 

```

---

hits are stored. Now the server iterates over all hash values. Starting in the last one, it searches the hash maps for documents containing an add operation for the considered subfeature. In order to execute this search operation the trapdoor  $T$  is required. When successful the identifier is added to a list of search results  $C$ . Afterwards, the server searches for del-operations for the same subfeature. The corresponding identifiers are removed from  $C$  as this document-subfeature pair was deleted from the index. Concluding its operation, the server sends  $C$  to the client. The client decrypts the values of  $C$  with the key  $esk$ . Then the contained identifiers are ranked according to their number of occurrence in  $C$ . Finally, the client returns the identifiers with top- $k$  scores.

### 4.3 Analysis

*Runtime Update.* The rebuild of level  $l$  requires  $\mathcal{O}(2^l \cdot l)$  operations as this level contains at most  $2^l$  entries. A rebuild of level  $l$  is always required after  $2^l$  update operations. Therefore, for a total number of  $N$  inserted subfeatures we get an amortized update cost of  $\mathcal{O}(N \cdot \log^2 N)$  per update. As a result, the amortized update cost of inserting a document with  $m$  features is given by:  $\mathcal{O}(m \cdot \lambda \cdot N \cdot \log^2 N)$ , where  $\lambda$  is the size of the used LSH family.

*Runtime Search.* In the worst case the protocol described here does not achieve sub-linear search time. This can be seen when adding a subfeature multiple times to the index and then deleting all of them except one. However, the authors of [19] describe how to slightly modify their scheme to achieve a sub-linear search time. This is achieved by storing the level of the corresponding add entry in del entries. As the algorithm works exactly as described in their paper, we decided to leave it out here. With these additions our scheme achieves a runtime for searching a single feature of  $\mathcal{O}(\lambda \min\{\beta + \log N, \gamma \log^3 N\})$ , where  $\gamma$  is the number of

documents having a subfeature in common with the search query and  $\beta$  is the number of added (but maybe in the meantime deleted) documents containing one of the searched subfeatures.

*Roundtrips.* As stated before, the Setup and Search protocols consist of a single round of communication between the client and the server. The number of roundtrips for the Update Protocol is determined by the Rebuild Protocol. As most rebuilds concern the upper levels of the hash maps, the data of the concerned levels will fit into the client's memory of size  $\mathcal{O}(N^\alpha)$  and hence only a single round of communication is required. In the worst case, the whole data structure has to be rebuild. A result of [9] on his oblivious sorting algorithm gives us the number of  $\mathcal{O}(N^{1-\alpha} \log^2 N^{1-\alpha})$  roundtrips in this case. Although such large rebuilds happen only very occasionally the high number of roundtrips favour fast RAM over disk storage as storage location for the hash maps.

*Required Storage.* For every subfeature, the three values  $\text{hkey}, c_1$  and  $c_2$  are stored in a hash map. When using AES-256 for encryption and an SHA-256 based keyed hash function,  $\text{hkey}$  and  $c_1$  will require 32 Bytes, while  $c_2$  requires 48 Bytes of storage. Thus,  $112\lambda m$  Bytes are necessary for storing a document with  $m$  features.

## 5 Parameters for Text Search

In the last section our scheme was presented in great generality so that it is applicable to a large range of problem domains. However, in order to apply the scheme to fuzzy text search, we have to specify a metric space embedding and a LSH family. Their parameters are chosen as in [14].

*Metric Space Embedding  $\rho$ .* We will define the metric space embedding as function  $\rho : w \mapsto \{0, 1\}^{500}$ . This is accomplished by considering the word  $w$  as a set of bigrams. All the bigrams of a word are then encoded in a common Bloom filter of length 500 by using 15 hash functions per bigram. The bits of the Bloom filter hit by a hash function then give  $\rho(w)$ .

*LSH Family.* We use the following LSH family  $G = (g_i)_{1 \leq i \leq \lambda}$  with  $\lambda = 37$ . We define:

$$g_i(x) = h_{i,1}(x) || \dots || h_{i,k}(x)$$

with  $k = 5$ . For all  $1 \leq i \leq \lambda$  and for all  $1 \leq j \leq k$  we set:

$$h_{i,j} : \{0, 1\}^{500} \rightarrow \{0, 1, \dots, 499\}, \quad x \mapsto \min \{y \mid 0 \leq y \leq 499 \text{ and } x_{\pi_{i,j}(y)} = 1\},$$

where  $\pi_{i,j}$  is a random permutation on  $\{0, \dots, 499\}$ . The parameters given here will be used in the Implementation in Sect. 7.

## 6 Security Proof

In this chapter we will prove that our proposed protocol indeed provides Adaptive Semantic Security for DSSSE. For this purpose we will adopt the security proof of [19].

**Theorem 1.** *The DSSSE given in Sect. 4 provides Adaptive Semantic Security for DSSSE (Definition 9) in the random oracle model.*

*Proof.* We will show that there exists a polynomial-size simulator  $S$ , such that the simulated view  $S(\mathcal{T}(\mathcal{H}_n))$  and the real view  $\mathcal{V}(\mathcal{H}_n)$  are computationally indistinguishable. To show the existence of such a simulator  $S$  we will state how it constructs the different elements of the view from the available trace  $\mathcal{T}(\mathcal{H}_n)$ . The number of executed protocol steps is available to the simulator via the trace. Furthermore, it can access via the protocol pattern whether a protocol step was an update or a search operation. We can therefore consider update and search operations separately:

*Update.* We describe the behaviour of  $S$  separately for every subfeature  $v \in V$  in the update protocol. The update pattern of previous updates contains the number of subfeatures which have been inserted in the data structure. As the entries are inserted deterministically into the hash maps  $\mathbb{H}_i$ , the simulator can calculate the level  $l$  in which the new subfeature  $v$  will be stored by the number of already inserted entries. The simulator  $S$  can also calculate the number of entries in  $\mathbb{H}$ . During the oblivious sorting protocol chunks of size  $\mathcal{O}(N^\alpha)$  are transferred to the client, sorted in its memory and uploaded back to the adversary. The simulator  $S$  simulates this step by generating correctly sized chunks of encoded entries  $e = (\text{hkey}, c_1, c_2)$  and uploading them to the adversary. The values of  $\text{hkey}$  and  $c_1$  are chosen at random, while  $c_2$  as semantically-secure ciphertext is chosen as the value of  $\text{Encrypt}(0)$ . Due to the obliviousness of the sorting algorithm this successfully simulates the rebuild protocol.

*Search.* In the case of search operation, the simulator knows the access pattern  $\mathcal{A}$ , the search pattern  $\mathcal{F}$  and the similarity pattern  $\mathcal{S}$ . By the update patterns of all the previous update operations, the simulator knows the size of the data structure  $N$ . Furthermore it can deduce by the update and access pattern the levels in which the subfeatures  $v$  of the query are stored. The client generates the following trapdoor  $T$ . If the subfeature has not been searched before or its level  $l$  has been rebuilt meanwhile, the trapdoor token $_l$  for this subfeature and level  $l$  consists of random values. Otherwise, it resends the old trapdoor token $_l$ . Due to the pseudorandomness of the keyed hash function  $H$ , the adversary cannot distinguish between these pseudorandom values and the output of  $H$ . Furthermore, due to this pseudorandomness the probability that this same token has already been sent to the adversary is also negligible.

We now want to program the random oracle  $H^*$  in the right way, i.e. such that  $H^*_{\text{token}_l}(0||\text{op}||\text{cnt})$  and  $H^*_{\text{token}_l}(1||\text{op}||\text{cnt})$  return the “right” values, when

queried by the adversary. If the oracle  $H_{\text{token}_l}^*$  is queried for  $(0||\text{op}||\text{cnt})$  or  $(1||\text{op}||\text{cnt})$  where  $\text{cnt}$  is greater than  $\text{cnt}_{\text{op},v}$  it returns random values. In all other cases it should return a valid pair of  $(\text{hkey}, \text{id} \oplus c_1)$  values when queried for  $(0||\text{op}||\text{cnt})$  and  $(1||\text{op}||\text{cnt})$ .

Now two cases can be considered. It could be the case that all items  $(v, \text{id}, \text{op})$  belonging to the same  $(v, \text{id})$  are in the same level. In this case, the random oracle chooses a random, unused entry from this level and returns it when queried for the values mentioned above. If these items are not in the same level, the simulator randomly chooses a level according to the distribution of the entries. As the insertion order of entries belonging to the same document is random, this return values are indistinguishable from real values for the adversary. Since by this construction all elements of  $S(\mathcal{T}(\mathcal{H}_n))$  and  $\mathcal{V}(\mathcal{H}_n)$  are computationally indistinguishable, our scheme fulfils the security definition.  $\square$

By slightly changing the protocol, the assumption of the random oracle is not necessary [19]. As this change increases the amount of computation performed on the client as well as the communication bandwidth between client and server, we will not describe it here.

## 7 Experimental Results

We implemented the proposed scheme in Java to evaluate its performance. The encryption and decryption is implemented as AES-256. For the hash function we used SHA-256, and for the keyed hash function  $H$  as well as the random oracle  $H^*$  we used HMAC-SHA-256. Within the rebuild algorithm we used a 8-way mergesort and assumed a client storage of 1024 entries, which is significantly lower than in [19]. Our implementation is single-threaded and does not contain the optimizations described in Sect. 4.3 to achieve the asymptotic search runtime also in the worst case. In our opinion, this worst case occurs too seldom to justify the overhead introduced by improving its asymptotic runtime. For our benchmark the client as well as the server code was deployed on the same machine.

For measuring the update and search runtime we used the bodies of the mails in the Enron e-mail data set [7]. This dataset contains 517 401 mails with a total of 688 270 unique keywords. The total number of document-keyword pairs in this dataset is 60 920 970, i.e. each mail contains an average number of about 118 unique keywords. We generated the keywords by changing all letters to lower case and thereafter considering all strings of a length greater than one character. The large number of unique keywords can be explained by the fact that the mails sometimes contain misspelled words or e-mail addresses.

In order to test the update operations, we inserted part of the mail dataset in a random order. For 100 000 document-keywords pairs, we achieved an insertion performance of about 26 document-keyword pairs per second on our test system (Intel i7-3770 @ 3.40 GHz, 16 GB RAM, Ubuntu 14.04, Oracle Java 8).

The search operation was benchmarked by queries chosen at random from the set of unique keywords. The probability of choosing a certain keyword was

given by its relative frequency in the whole dataset. When executing search operations over an index containing 100 000 document-keyword pairs, we achieved a average performance of 20 queries per second. Since all identifiers have to be ranked, this figure is independent of the number  $k$  of returned identifiers. We did not benchmark the quality of our search results as the results of [14] regarding precision and recall remain unchanged.

## 8 Conclusion

Our SSSE scheme is dynamic, provides sub-linear search time and has low leakage. This is accomplished by combining a fuzzy and a non-fuzzy searchable encryption scheme. In our opinion, this technique of combining fuzzy and non-fuzzy schemes is interesting in its own right. It allows to consider searchable encryption and similarity search separately. By combining such schemes, very promising candidates for feature-rich similarity search schemes could be created.

Our scheme could be easily extended to a multi-keyword search by slightly changing the Search Protocol. To achieve this the client would send the trapdoors belonging to several words to the server. When the client ranks the results, identifiers of documents containing all search terms (conjunctive search) then are ranked higher than documents containing only a subset of the search terms.

To apply our scheme to a specific problem domain a appropriate metric space embedding has to be used. In Sect. 5 we showed how this embedding can be chosen in the case of text search. It would be an interesting task to find such mappings for other problem domains, e.g. different types of biometric data as fingerprints or iris images.

The (amortized) asymptotic runtime of search and update operations of our scheme is good compared to other fuzzy search schemes. However, as our implementation shows, the involved constants are high. Although a further optimisation of the scheme might achieve better runtime, the most compelling aspect of our scheme is something different: By achieving good asymptotic runtimes with low leakage, our work complements the theoretical work of [1]. They investigated how much performance in terms of asymptotic runtime, low leakage and storage efficiency an SSSE scheme can achieve. On the one hand, their impossibility result for a certain combination of these properties can be seen as an upper bound for the performance of any DSSSE scheme. On the other hand, our construction of a good scheme with regard to this performance characteristics can be understood as lower bound for an optimal DSSSE scheme. Further research is necessary in order to find the “optimal” scheme.

**Acknowledgment.** This work was funded by the DFG under grant number Wi 4086/2-2.

## References

1. Boldyreva, A., Chenette, N.: Efficient fuzzy search on encrypted data. In: Cid, C., Rechberger, C. (eds.) FSE 2014. LNCS, vol. 8540, pp. 613–633. Springer, Heidelberg (2015). doi:[10.1007/978-3-662-46706-0\\_31](https://doi.org/10.1007/978-3-662-46706-0_31)

2. Bösch, C., Hartel, P.H., Jonker, W., Peter, A.: A survey of provably secure searchable encryption. *ACM CSUR*. **47**(2), 18:1–18:51 (2014)
3. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: *NDSS 2014* (2014)
4. 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). doi:[10.1007/11496137\\_30](https://doi.org/10.1007/11496137_30)
5. Chuah, M., Hu, W.: Privacy-aware bedtree based solution for fuzzy multi-keyword search over encrypted data. In: *IEEE ICDCS 2011*, pp. 273–281 (2011)
6. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: *CCS 2006*, pp. 79–88 (2006)
7. Enron email dataset (2015). [www.cs.cmu.edu/~enron/](http://www.cs.cmu.edu/~enron/)
8. Goh, E.: Secure indexes. *IACR Cryptology ePrint Archive* (2003)
9. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) *ICALP 2011*. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-22012-8\\_46](https://doi.org/10.1007/978-3-642-22012-8_46)
10. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: *ACM SIGSAC*, pp. 310–320 (2014)
11. Hu, C., Han, L.: Efficient wildcard search over encrypted data. *Int. J. Inf. Sec.* **15**(5), 539–547 (2016)
12. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.-R. (ed.) *FC 2013*. LNCS, vol. 7859, pp. 258–274. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-39884-1\\_22](https://doi.org/10.1007/978-3-642-39884-1_22)
13. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: *CCS*, pp. 965–976 (2012)
14. Kuzu, M., Islam, M.S., Kantarcioglu, M.: Efficient similarity search over encrypted data. In: *IEEE ICDE*, pp. 1156–1167 (2012)
15. Li, J., Wang, Q., Wang, C., Cao, N., Ren, K., Lou, W.: Fuzzy keyword search over encrypted data in cloud computing. In: *INFOCOM*, pp. 441–445 (2010)
16. Liu, C., Zhu, L., Li, L., Tan, Y.: Fuzzy keyword search on encrypted cloud storage data with small index. In: *IEEE CCIS*, pp. 269–273 (2011)
17. Örencik, C., Kantarcioglu, M., Savas, E.: A practical and secure multi-keyword search method over encrypted cloud data. In: *IEEE CLOUD*, pp. 390–397 (2013)
18. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: *IEEE S&P*, pp. 44–55 (2000)
19. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: *NDSS* (2014)
20. Strizhov, M., Ray, I.: Secure multi-keyword similarity search over encrypted cloud data supporting efficient multi-user setup. *TDP* **9**(2), 131–159 (2016)
21. Sun, W., Wang, B., Cao, N., Li, M., Lou, W., Hou, Y.T., Li, H.: Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. In: *ACM ASIA CCS*, pp. 71–82 (2013)
22. Wang, B., Yu, S., Lou, W., Hou, Y.T.: Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In: *INFOCOM*, pp. 2112–2120 (2014)
23. Wang, C., Ren, K., Yu, S., Urs, K.M.R.: Achieving usable and privacy-assured similarity search over outsourced cloud data. In: *INFOCOM*, pp. 451–459 (2012)



24. Xia, Z., Wang, X., Sun, X., Wang, Q.: A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE TPDS* **27**(2), 340–352 (2016)
25. Yuan, X., Cui, H., Wang, X., Wang, C.: Enabling privacy-assured similarity retrieval over millions of encrypted records. In: Pernul, G., Ryan, P.Y.A., Weippl, E. (eds.) *ESORICS 2015*. LNCS, vol. 9327, pp. 40–60. Springer, Cham (2015). doi:[10.1007/978-3-319-24177-7\\_3](https://doi.org/10.1007/978-3-319-24177-7_3)
26. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of file-injection attacks on searchable encryption. In: *USENIX*, pp. 707–720 (2016)

Security and Trust Management  
13th International Workshop, STM 2017, Oslo, Norway,  
September 14-15, 2017, Proceedings  
Livraga, G.; Mitchell, C. (Eds.)  
2017, X, 235 p. 66 illus., Softcover  
ISBN: 978-3-319-68062-0