Implementation of the Searchable Encryption System Sophos

Author
Martin Schwingenschuh

Submission
Institute of Computational Perception

Thesis Supervisor
a.Univ.-Prof, Dr.
Josef Scharinger

March 2023

Abstract

The current trend for document storage is to outsource the physical storage to a cloud provider which has full access to the documents which means that the straight forward solution of storing the documents in clear text on the server is not satisfactory, if the server is not to be trusted. The access to the documents can be restricted for the server by using cryptographic means and encrypt all documents which are stored on the server. While an encryption prevents the server from reading the documents it also prevents the ability of using search queries on those encrypted documents which becomes more important as the number of stored documents increases. Searchable Encryption solves this problem by using additional data structures and search protocols. We identified two major approaches in literature that are used. The first is an index based approach, which always leaks a bit of information, and the second solution is an oblivious ram model in different forms, which usually has a very high overhead cutting into scalability and performance. In this work we look into index based solutions by implementing an application for document storage on a server which uses one specific scheme. We did this to show that such systems are indeed usable in real life applications and not limited to academic prototypes.

Contents

1 Introduction

The setting of searchable encryption as shown in Figure 1.1 is that one party called Alice or client has documents which should be stored not on her local storage but outsourced to a second party called Bob or Server. The catch is, that Alice does not trust Bob meaning that Bob should not be able to read the documents or gain any information on them. This could easily be solved by encrypting the documents with for example a symmetric encryption and storing the cipher text at Bob preventing him from reading them. Now Alice has the problem that she can not instruct Bob to return documents which contain a certain keyword or phrase. This is prevented by the encryption since every information or pattern usable for search problems is destroyed, which we actually want for it to be secure. To support search queries, e.g. return all documents containing a certain keyword, an additional data structure is needed. The whole system with the encrypted documents, the data structure and the problem of search queries is called searchable encryption. In literature there are two major approaches in the field of searchable encryption, indexed based[2, 3] and ORAM (oblivious ram) based systems[4, 5, 6]. The indexed based approach usually is performant and scalable, but has a certain level of information leakage which can not be prevented. ORAM based systems on the other hand are considered the best solution in therms of security and confidentiality, but come with a high overhead and do not scale very well. In this work we took a deeper look into one specific indexed based solution called Sophos and implemented the scheme into a client server application.

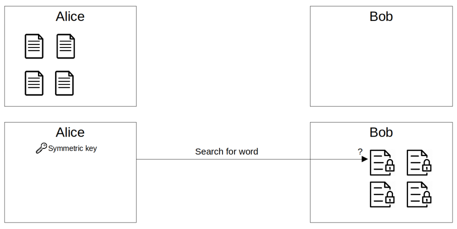


Figure 1.1: Motivation and high level view

2 Theory

2.1 General

In this work we set as goal to implement a system that allows a user (Alice/Client) to store documents at a second party (Bob/Server) where Alice does not trust Bob and is still able to issue search queries which Bob can execute and return the results without learning any information on the stored documents. This means that we have to use some cryptographic system to ensure that Bob can not learn any information on the uploaded documents. This problem of confidentiality could be solved by simply encrypting the documents prior to uploading with a symmetric encryption scheme. The problem is that if we just upload encrypted documents Alice can no longer issue search queries to Bob because the encryption destroys all features which would enable a search. This is exactly why we use encryption to destroy all features which could leak information on the content. Such search queries usually have the form of one given search word w and a result set of documents $Q(w) = \{D_0, D_1, \ldots, D_n\}$ which contain the given search word. The systems to solve such problems is called Searchable Encryption schemes (SE-schemes) which usually add some data structure to the encrypted documents to enable the search functionality. In literature there are mainly two approaches, the ORAM (oblivious ram) approach which does not leak any information but usually has a higher overhead and indexed based solutions which operate by building and maintaining an encrypted index for searches but leak some information on the documents. While all those solutions are valid and should be considered, we concentrate in this work on one specific indexed based solution called $\Sigma o\varphi o\zeta$ (Sophos) proposed by Bost et al. [2]. We used this scheme to build a Server/Client application where the user can upload PDF documents in an encrypted fashion to the Server but still be able to issue search queries for search words where all Documents which contain the given search word are returned.

2.2 Indexed Based Schemes

Index based schemes use additionally to the encrypted documents an encrypted index for searches. This index is a mapping from keywords to documents where each word points to a document. With such an index the Server can search through the list of encrypted words and return the corresponding documents. This index can be constructed beforehand or expanded with update queries. This distinguishes the difference between static schemes and dynamic ones where static scheme mean that the index has to be constructed before uploading the files to the Server and dynamic ones that the index can be expanded with update queries while documents are added one after another. Since in a practical relevant scheme we need a scheme where we can add documents while the system is running we only consider dynamic schemes in this work. For a better understanding we formally define what an SSE scheme is. A dynamic SSE scheme has to implement three algorithms setup, update and search.

1. $Setup$
   $Setup_C()$ brings the Client in an operational state by retrieving or generating the necessary keys and connecting to the database W.
   $Setup_S()$ brings the Server in an operational state by connecting to the two databases T and D.

2. $Update$
   $Update_C(w)$ is in our case an exclusive insert operation since we do not support deletions. This algorithm operates on one word w which is called for each word in a given document D.
   $Update_S(eind, edoc)$ is the counterpart to $Update_C$ and respectively only supports insert operations. The Server has only the encrypted index *eind* and the encrypted document *edoc* as parameters.

3. $Search$
   $Search_C(w)$ is the algorithm to search the Client side stored data structure for the word w and send a corresponding call to the Server.
   $Search_S(w_k)$ The Server gets the parameter w_k from the Client which denotes the encrypted search word and returns a set of documens.

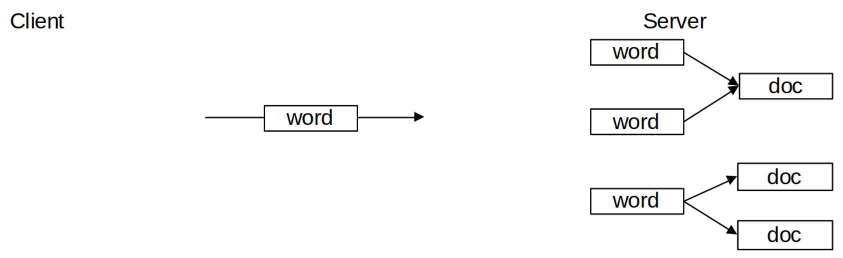
Before we introduce the Sophos scheme formally we first introduce the general SSE scheme informally by developing the scheme step by step starting with no encryption at all and building up to the Sophos scheme by adding one concept at a time.

Figure 2.1a shows a Client/Server application where the Server stores a number of documents and the corresponding index. The Client sends the search word w form of a query in clear text to the Server. The Server can then process the query and return the found documents. In this easy example there is no encryption used, and the Server can read the searched word (the search pattern) and the documents (the information pattern) which obliviously means that the Client has to trust the Server.

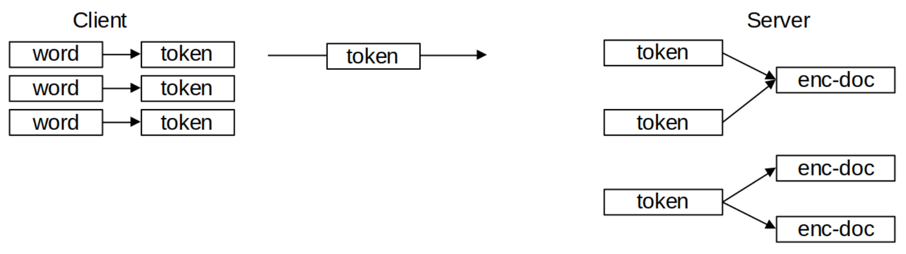
In Figure 2.1b the search words and the index are encrypted with a symmetric encryption scheme. We call those encrypted words tokens since in literature this term is used for cryptographic primitives. The problem with this approach is that the Server can statically analyze the stored documents. Meaning that he can attack the information between token stored in the index and the documents. Further the Client has to store for each word the corresponding token. What we want is a data structure stored at the Server side which does not tell the Server any information which documents have the same words in them until a search query is issued.

To achieve this, we first have to construct a data structure on the Server side which maps one Token to one document depicted in Figure 2.1c. This destroys the link between the tokens and multiple documents. The problem of a static analysis is still there which will be fixed in the next step. But the needed storage on the Client side is larger since we have to store multiple tokens per search word. In this data structure on the Client is now the information stored which word belongs to multiple documents. Further multiple tokens need to be sent to the Server.

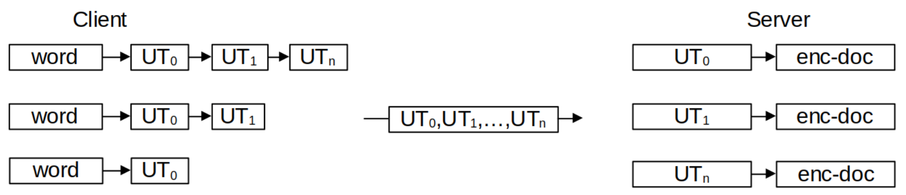
Figure 2.1d shows the addition of a gateway function on the Server side. The Server can only compute the document pointer from the token e if the Client issues K_w which can only be computed by the Client with the symmetric key. This version has still the problem of the high communication with sending multiple tokens. The next step is then the Sophos scheme which solves this issue.



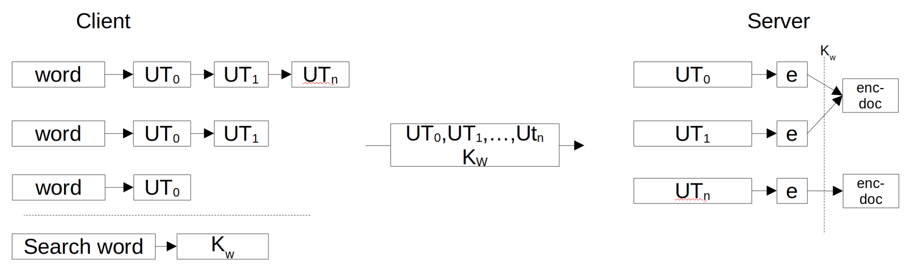
(a) Simple Index



(b) Encrypted Index



(c) Enhanced Encrypted Index



(d) Hidden Index

Figure 2.1: An intuitive build up from a basic index (a) where no encryption whatsoever is used to a naive approach encrypting the index (b) by using cryptographic functions to map words to tokens, to using update tokens in (c) and finally hiding the index in (d) by introducing a gatekeeper function.

2.3 Sophos

2.3.1 Overview

Bost et al. proposed an SSE solution [2] which is forward secure with a possible extension to also make the scheme backward private. In this work however we only consider the basic version. This scheme uses two types of tokens UT (Update Tokens) and ST (Search Tokens). A token is a number used to help in calculations. Those tokens are needed since we need a from we can use mathematics on. The index of documents or whole documents are not well suited since we can not choose what value those should have. Tokens can be anything we want and with the use of mappings we can translate tokens in anything we need. As shown in Figure 2.2 Sophos utilizes a special relation between tokens to gain a performance boost we will discuss later. Search-Tokens can be viewed as a list of integer values that can be traversed using a trapdoor function. That means that the function call is easy to calculate but the inverse without knowing the secret very hard. In Sophos an asymmetric key pair is used to achieve this property.

$$ST_{i+1} = \pi_{PK}(ST_i)$$
$$ST_{i-1} = \pi_{SK}(ST_i)$$

PK is the public and SK the secret key from the key pair and i the index of the current token. In Sophos the Client generates the key pair and only publishes the public key to the Server. With that we can see that only the Client is able to generate the previous token ST_{i-1} of the list. The Server on the other hand can only calculate the next element from a given start point. In practice this means that the Client only needs to send one Search Token and the size of the list to the Server to transmit a list of Tokens which improves the communication overhead and achieves the forward security property. In addition to the ST-ST relation there is an ST-UT relation as well. This is achieved with a Hash function H_1 on the Search Token ST_i. That means that we have on the Server side a list of search tokens where we can calculate for each search token an update token resulting in a list of update tokens. On the Server update tokens are used to identify the location where documents are stored. Update tokens can be viewed as the index of an array on the Server storing the encrypted indices e resulting in a mapping

$$\begin{tikzcd}
ST_0(w) \arrow[r, bend left, "\pi_{\mathsf{SK}}^{-1}"] \arrow[d, "H_{K_w}"'] & ST_1(w) \arrow[l, "\pi_{\mathsf{PK}}"] \arrow[d, "H_{K_w}"'] & \cdots & ST_c(w) \arrow[ll, dashed, "\pi_{\mathsf{PK}}^{(c-1)}"] \arrow[r, bend left, "\pi_{\mathsf{SK}}^{-1}"] \arrow[d, "H_{K_w}"'] & ST_{c+1}(w) \arrow[l, "\pi_{\mathsf{PK}}"] \arrow[d, "H_{K_w}"'] \\
UT_0(w) & UT_1(w) & \cdots & UT_c(w) & UT_{c+1}(w)
\end{tikzcd}$$

Figure 2.2: Relation between search tokens ST and update tokens UT where π denotes a trap door permutation and H a hash function.

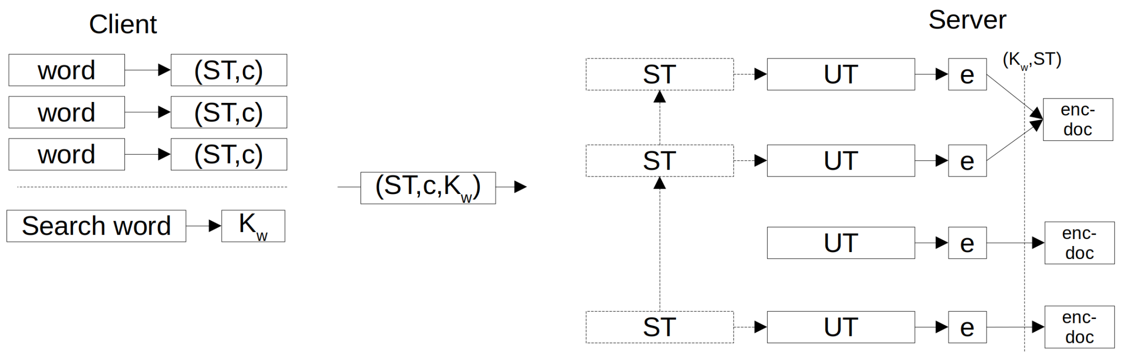


Figure 2.3: Simplified representation of the Sophos scheme showing the logical connections and an example of the token structure.

$$ST \to UT \to e \to ind$$

Figure 2.3 shows the token structure again. On the Client side is the array W which stores for each word in the database the word and a search token tuple (ST, c). On the Server side we see the token structure in the Figure given as example. We can see that this token structure is the main data structure of this scheme.

In the following we describe the scheme step by step by explaining the three algorithms of SSE (Init, Update, Search).

2.3.2 Init

Algorithm 1 is the initialization of Client and Server. On the Client side we need to generate a symmetric secret key with the length λ and an asymmetric key pair with the same length. Further we initialize the array W. We call it array but in reality it can be any data structure that can map words to Tokens, for example a hashmap. On the Server we only have to initialize the array T mapping the update tokens, which the Server calculates in the token structure, to encrypted indices.

Algorithm 1 Sophos - Init

```
Client:
1: $K_S \xleftarrow{\$} \{0,1\}^\lambda$
2: $(SK, PK) \leftarrow KeyGen(1^\lambda)$
3: $W \leftarrow emptymap$
Server:
1: $T \leftarrow emptymap$
```

2.3.3 Update

Algorithm 2 describes the update function where a big part is the generation of the token structure. First the Client searches in a local map W if for a given search word w a search token is stored. If no entry is found a new token is generated by uniformly sampling an l bit long number from the set of Integers, in other words a random number with l bits length. If an entry is found, the next node in the search token list is generated by applying π_{SK}^{-1} on the stored search token ST_c. The old search token in W at the position of the search word is replaced with the new search token and the counter is increased by one. Like we mentioned previously we only need to store the last element of the list since the other elements can be computed with the TDP. Next we calculate K_w from the secret key K_S and the search word w with a PRF (pseudo random function) F. This is done to get the search word in a form where all words have the same length and hide the clear text since we use the PRF with the secret key K_S. With the search word token K_w and the new search token ST_{C+1} the update token UT_{C+1} is calculated with a hash function $H1$. Further an encrypted index e is calculated to prevent the Server from static attacks against the DB because this encrypted index is only with information from the Client in the search algorithm dereferencable. For that the clear text index *ind* is XORed with a Hash from K_w and ST_{c+1}. The new update token UT_{c+1} and the encrypted index are sent to the Server and the Client side from the update function is done. On the Server side we only need to store the encrypted index e at the position UT_{c+1} in the map T.

Algorithm 2 Sophos - Update(Add)

```
Client:
 1: $(ST_c, c) \leftarrow W[w]$
 2: if $(ST_c, c) = null$ then
 3:     $ST_0 \xleftarrow{\$} M, c \leftarrow -1$
 4: else
 5:     $ST_{c+1} \leftarrow \pi_{SK}^{-1}(ST_c)$
 6: end if
 7: $W[w] \leftarrow (ST_{c+1}, x+1)$
 8: $K_w \leftarrow F(K_S, w)$
 9: $UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
10: $e \leftarrow ind \oplus H_2(K_w, ST_{c+1})$
11: $Send(UT_{c+1}, e)\ to\ Server$

Server:
  $T[UT_{c+1}] \leftarrow e$
```

2.3.4 Search

Algorithm 3 describes the search of a word in the DB. The Client initiates the protocol by first searching in the local map W at the position of the search word w for a search token counter tuple (ST_c, c). The protocol terminates if no entry is found since that means that for the given word no documents are stored in the DB on the Server. Next the search word is transformed using a PRF using the secret key K_S. The transformed search word K_w and the retrieved Search token ST with the counter c are sent to the Server. The Server has to calculate the token structure from the given search token ST and counter c. As mentioned before the token structure is like a list and the counter c is the number of elements this list contains. That means that the Server has to iterate over the whole list starting from the

Algorithm 3 Sophos - Search

```
Client:
1: $ST_c, c \leftarrow W[w]$
2: **if** $(ST_c, c) == null$ **then**
3:     return
4: **end if**
5: $K_w \leftarrow F_{K_S}(w)$
6: $Server \xleftarrow{send} (K_w, ST_c, c)$

Server:
1: **for** i = c to 0 **do**
2:     $UT_i \leftarrow H_1(K_w, ST_i)$
3:     $e \leftarrow T[UT_i]$
4:     $ind \leftarrow e \oplus H_2(K_w, ST_i)$
5:     $returndoc_i$
6:     $ST_{i-1} \leftarrow \pi_{PK}(ST_i)$
7: **end for**
```

head c to the last element 0. In each iteration the current update token UT_i is calculated by using a hash function on the transformed search word and the current search token ST_i. With the update token the encrypted index can be retrieved from the map T. Here we can see that the update token represents the logical location where the encrypted index is stored. This encrypted index needs to be decrypted by XORing it with a hash from the transformed search word K_w and the current search token ST_i. Note that the transformation of the search word w into K_w can only be done by the Client with the secret key K_S. With the decrypted index the Server can now return the stored document to the Client. Now the next element in the list ST_{i-1} needs to be calculated with the TDP π_{PK} on the current search token ST_i. Note that here the public key is used for the TDP.

2.4 Our Changes to Sophos

While Sophos does not need changes to work properly or securly we had to make small changes to comply to our usecase. The changes we made are not essential to the algorithm, only for the usability. The first is that we also use a RocksDB data base on the Client for the tokenstorage instead of a hashmap. We did that to have a persistent storage on the Client since a user would expect his changes in one session be there in the following session. The next is the generation and storage of keys (symmetric and asymmetric key pair) which is not properly discussed for Sophos, so we decided to use the Java crypto framework and the provided keystore. Lastly we changed the indices *ind* for the document storage on the Server. We took the name of the document file and encrypted it using the symmetric key on the Client to generate an encrypted index *eind*. This encrypted index is then used as index for the Server. With that the Server can not learn anything on the index but on the Client side we can at anytime decrypt the index to get the filename. This is especially useful in the additional query we support to get all indices stored on the Server. We support uploading and downloading of single documents by uploading The encrypted document to the server and in the same step indexing them. This also enables us to expand the functionality in the sense that we do not nescecarily need to return whole files at search queries but only the indices which are decrypted on the Client side and the user decides which file he wants to download.

2.4.1 Init

The init function is pretty much the same only that we initialize three seperate databases and the cryptographic keys. The Databases we initialize are the Tokenstorage on the Client W, The TokenStorage on the Server T and the Documentstorage on the Server D. All of those are RocksDB instances and initialization means that if they are missing a new empty database is created and if a database is already on disk it is opened. Similar to that we either read existing cryptographic primitives or generate new ones if there is nothing found on disk. Algorithm 4 shows the altered algorithm where we first read the symmetric key K_S from the keystore. With the secret key we can read the asymmetric key pair from disk. We need the secret key because the private key of the key pair is encrypted. If the keys are not found we generate a new set of keys. After that we open the Token Storage W where $DB.open()$ connects to an existing database or if no database is found generates a new one. Lastly on the Client side we connect to the Server with an RMI-stub. On the Server side we only need to open the Token storage T, open the Document storage D and init the RMI infrastructure by registering the Server.

Algorithm 4 SophosSE - Init

```
Client:
1: $K_S \leftarrow Keystore.read()$
2: $(SK, PK) \xleftarrow[K_S]{} Disk.read()$
3: if $K_S == null || (SK, PK) == null$ then
4:     $K_S \xleftarrow{\$} \{0,1\}^\lambda$
5:     $(SK, PK) \leftarrow KeyGen(1^\lambda)$
6: end if
7: $W \leftarrow DB.open()$
8: $ServerStub \leftarrow connect\ to\ Server$
Server:
1: $T \leftarrow DB.open()$
2: $D \leftarrow DB.open()$
3: $RMI.register()$
```

2.4.2 Update

The original algorithm $Update(w)$ does not need to be changed much to be usable in our system. Algorithm 5 shows those changes to the original algorithm. Additionally, to the search word parameter we added the encrypted index *eind* of the corresponding document. We did this to change the calculation of *e* which is stored on the Server and later used in the search algorithm in which the Server learns *eind* but in the changed version not *ind*. For us this is important, because *ind* is the document name which should be kept confidential. Further, since we implemented the Client/Server communication with the Java RMI infrastructure we send the update with a function call of the RMI Object.

Algorithm 5 SophosSE - Update(w, eind)

```
Client:
 1: $(ST_c, c) \leftarrow W[w]$
 2: if $(ST_c, c) = null$ then
 3:     $ST_0 \overset{\$}{\leftarrow} M, c \leftarrow -1$
 4: else
 5:     $ST_{c+1} \leftarrow \pi_{SK}^{-1}(ST_c)$
 6: end if
 7: $W[w] \leftarrow (ST_{c+1}, c+1)$
 8: $K_w \leftarrow F(K_S, w)$
 9: $UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
10: $e \leftarrow eind \oplus H_2(K_w, ST_{c+1})$
11: $ServerStub.update(UT_{c+1}, e)$

Server:
    $T[UT_{c+1}] \leftarrow e$
```

2.4.3 Search

The only changes to the search algorithm we had to make are the communication and analog to the update algorithm the encrypted indices and documents. Algorithm 6 shows the changes to the original algorithm. On the Client side we use a function call on an RMI-Object for the communication. To note is that we store the counter c in the Token object and thus only send the search token to the Server.

Algorithm 6 SophosSE - Search(w)

```
Client:
1: $ST_c, c \leftarrow W[w]$
2: if $(ST_c, c) == null$ then
3:     return
4: end if
5: $K_w \leftarrow F_{K_S}(w)$
6: $ServerStub.search(K_w, ST_c)$

Server:
1: for  i = c to 0 do
2:     $UT_i \leftarrow H_1(K_w, ST_i)$
3:     $e \leftarrow T[UT_i]$
4:     $eind \leftarrow e \oplus H_2(K_w, ST_i)$
5:     $return\ encr\ doc$
6:     $ST_{i-1} \leftarrow \pi_{PK}(ST_i)$
7: end for
```

2.4.4 Additional Algorithm Add Document

The Algorithms of the original Sophos solution only operate on the word level. Meaning that we can add words and search on a word basis where the Server returns document ids. In our understanding a usable system would have to support a document level on top of the word level, since users would more likely have the use case of just adding whole documents. To support this use case we introduce a new algorithm which operates on top of the changed Sophos algorithms and extends its capability to support documents. Algorithm 7 shows the functionality of adding a document to the system from the Client point of view. First we read the file from disk and encrypt the name which will be the index or ID of the file for the Server with a symmetric encryption. Next the whole file D is encrypted also using symmetric encryption resulting in a cipher text D'. The function $ServerStub.uploadFile(eind, D')$ uploads the key value pair $(ID, Document)$ to the Server. We use the encrypted index *eind* as key and the encrypted Document D' as value. With the encryption of both parts the Server should not be able to learn anything about the content or context of the document. With that the document is stored on the Server, and we have to build up the index for searching. To do that we need to process each word w contained in the document D. We separate words by spaces which can become a problem in the indexing since the format is not consistent and sometime makes not much sense, like indexing single numbers or words with white spaces as prefix or suffix. To fix that we preprocess the words where we delete all white spaces, control sequences and single numbers. After preprocessing we call the update algorithm on each word resulting in the whole document being indexed and ready for search queries.

Algorithm 7 SophosSE - AddDocument(D)

```
Client:
1: $D \leftarrow$ load File
2: $eind \leftarrow F(D_{name})$
3: $D' \leftarrow F(D)$
4: $ServerStub.uploadFile(eind, D')$
5: for each $w$ in $D$ do
6:     $w \leftarrow preprocess(w)$
7:     $ServerStub.update(w, eind)$
8: end for
```

3 Implementation

3.1 General

We decided to use JAVA for both the Client and Server application with the integrated crypto functionalities cryptofx so that we are independent of the underlying machine. Both the Client and Server application are built with a persistent storage, meaning that the state is stored on disk between executions. On the Client the map *W* is stored in a RocksDB and the cryptographic keys are stored in a Java Keystore and encrypted files. On the Server the map *T* and the Document storage *D* are both implemented using two RocksDB instances. Figure 3.1 shows the different components and the mapping of the databases. At the first execution or if components are missing both on the Client and Server the missing component will be generated. This is applicable for all the databases and the key management on the Client. To note is that generating means in this case crating a new empty database or generating new keys which is only intended to be used at the first execution or a reset of the whole system since those operations. The source code is published on GitHub[1].

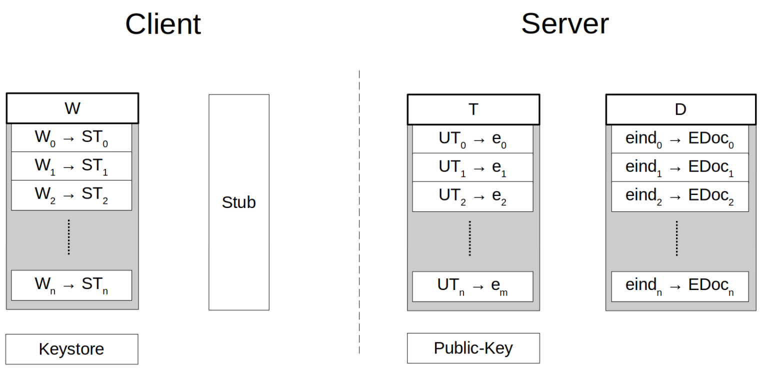


Figure 3.1: Overview of the implemented components without dataflow.

3.2 Components / Building Blocks

3.2.1 Key Management

We understand key management as the generation of cryptographic keys and storing them securely in a persistent way on disk. We need for the implemented scheme a total of three keys. Concrete we need one symmetric key also called secret key for the symmetric encryption and decryption of files. And one Public key pair consisting of one private key and one public key for the Trapdoor function π. Since we use the JAVA language we can use the cryptographic library Package javax.crypto which provides a key Store. This key Store can be used as interface between our software and the persistent storage on disk. We use this storage to store the symmetric secret key. This key is protected with a password, so we can consider this key secure since reading from disk without knowing the password is not easily possible. The Asymmetric key pair can only be stored if we build a certificate chain for the key pair. This would be too much effort just for the purpose of storing keys since we do not need the certificates. We only need the key pair for the TDP π. We solved this by storing the public key pair in an encrypted form on the disk using symmetric encryption and our secure secret key. This means that we also have to store the initialization vector which was used in the encryption. This vector needs not to be secret meaning we can just store it in a clear text file which means we have four files stored on disk "iv.key", "keystore.key", "private.key" and "public.key". The key structure is only generated once on the Client and then stored on disk at the first execution of the program and every start after that retrieves the stored keys. It is important to consider that if the keys are lost on the Client a recovery of the system is impossible. Figure 3.2 shows a schematic how the data flow of keys is implemented. On the left side is the permanent storage from which the access to the Keystore is regulated with the Password (PW) and the access to the private key with the encryption with the secret key. The SE-program can use all three keys after the retrieval process. This means that the keys are stored in clear text in the main memory as the Client program is executed. If an adversary can access the main memory this scheme would be broken. But for us this is sufficient since we concentrate in this work on searchable encryption and key protection in memory is out of scope.

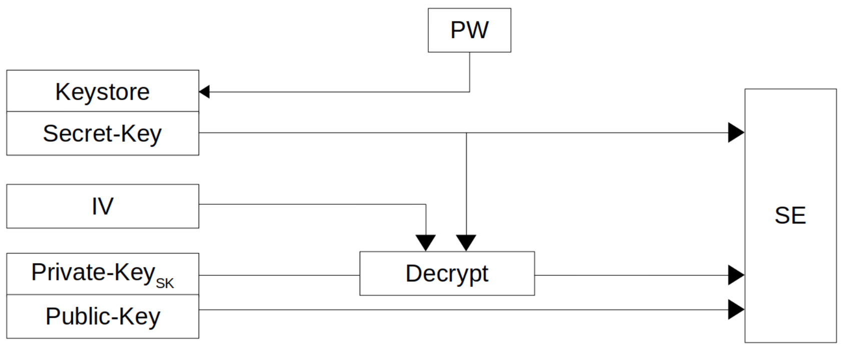


Figure 3.2: Key Management with the data flow and the decryption process with the user password that unlocks the Keystore for the secret key which can decrypt the private key.

3.2.2 Token

A Token is basically just a large integer number used for the TDP π to generate a list of integers, as shown in Figure 3.3, meaning that we can generate a number of integers but only have to store the first one. Additionally, we store the counter needed for the Sophos algorithms in the same object which also distinguishes if the Token is a search token ST or an update token UT. This distinction can also be done with the context where the token is provided. To note is that only the BigInteger field is used to identify a token and is the only part that represents the value of the token.

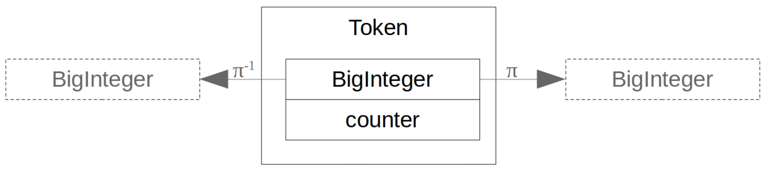


Figure 3.3: Token Structure with the TDP π to traverse the List

3.2.3 Token Storage W on Client side

In our use case a persistent storage solution is needed so that the system can store the current state and resume at a later time with the same restored state. To achieve this we need a persistent storage solution for all the databases. Since we have for all databases a simple mapping from one data block (Key) to another data block (Value) we can use a simple storage solution. In the original paper the database RocksDB is used which we also integrated in our solution since it is good at key value mappings. RocksDB can store tuples of the form (K, V) where both K and V are byte blocks. In our application we can just use serialization to change this into a mapping of Object to Object which makes the integration very easy.

We use an instance of the database system RocksDB with a very simple and straight forward database scheme. Since we just need to store a mapping from words w to a Tuple of search token and counter (ST, c) we only need a key-value mapping. In our implementation the Token Object also contains the counter c so that we only need the mapping $w \rightarrow ST_w$. For this use case the key is the word w and the value is a Token Object described in subsection 3.2.2. Figure 3.4 shows the dataflow when the Client retrieves the searchtoken ST_w for a given searchword w. To note is that a new database is generated at the program start and if no existing database is found. But it is important that the system only functions properly if the state of the database represents the real state. If the database is lost there is no chance to recover at the current state of development.

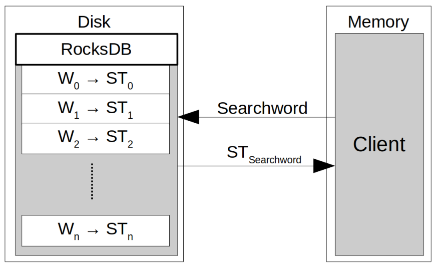


Figure 3.4: Token Storage W with RocksDB

3.2.4 Token Storage T on Server side

This token storage, as shown in Figure 3.5, maps Update tokens UT to an intermediate encrypted index e which is used to calculate the encrypted index $eind$ that is needed for the access of the document storage D. The update tokens UT are stored as byte blocks in the database where only the BigInteger value is serialized, the counter is disregarded. Similar the encrypted index e is also a BigInteger value which will be retrieved as byte block and then cast to an Object.

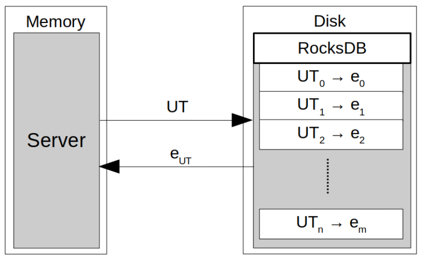


Figure 3.5: Token Storage T on Server with RocksDB

3.2.5 Document Storage D on Server side

Similar to the token storage we use here an RocksDB instance. This storage implements the mapping from indices to documents $eind \rightarrow EDoc$ as shown in Figure 3.6. We use the filename as index and key value for the mapping but since we want the filename and the file content confidential, the Client has to encrypt the name and file changing the mapping to $ind_{encr} \rightarrow Doc_{encr}$. We want to point out, that the storage solution is independent of the encryption and the Client has to manage the decryption.

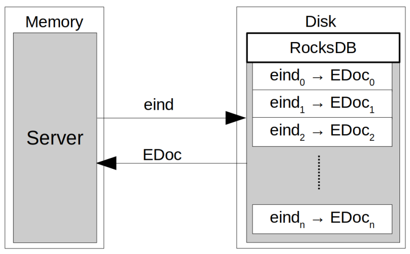

Memory
Server
eind
EDoc
Disk
RocksDB
$eind_0 \rightarrow EDoc_0$
$eind_1 \rightarrow EDoc_1$
$eind_2 \rightarrow EDoc_2$
$eind_n \rightarrow EDoc_n$

Figure 3.6: Document Storage D on Server with RocksDB

3.3 Client-Server Communication

We use the Java RMI infrastructure as implementation between Client and Server. This infrastructure greatly improves the implementation by abstracting the underlying network overhead. The RMI system provides an Object stub to the Client which works as accessing members of a local object as shown in Figure 3.7. With this implementation we can focus on the high level view of our algorithms. We only need to define the interface, what functions the Server should provide. This interface can be viewed as API of the Server and is in our implementation defined as follows.

- void uploadFile(BigInteger ind, SealedObject file) throws RemoteException;
- SealedObject getFile(BigInteger ind) throws RemoteException;
- List<File> getAllFiles() throws RemoteException;
- List<BigInteger> getAllFileIndices() throws RemoteException;
- List<SealedObject> search(Token ST, BigInteger Kw) throws RemoteException;
- void update(Token UT, BigInteger e) throws RemoteException;
- void setPublicKey(RSAPublicKey key) throws RemoteException;

In the Client implementation we can then call the functions defined in this interface directly on the Server stub object. An example would be ServerStub.upload(eind, edoc); which uploads the given document with the given index to the Server where those parameters can be directly used to store the tuple in the database.

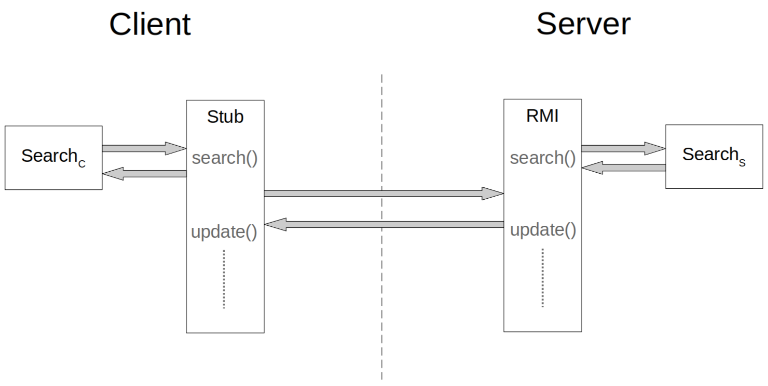
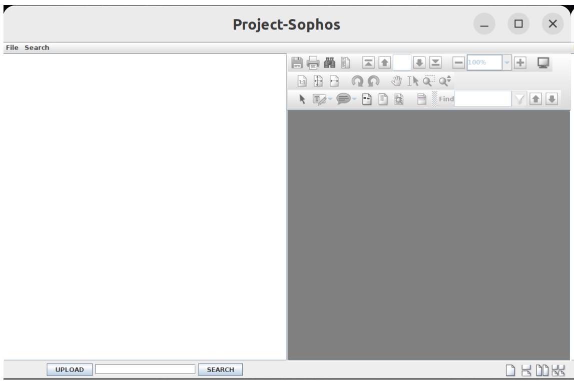


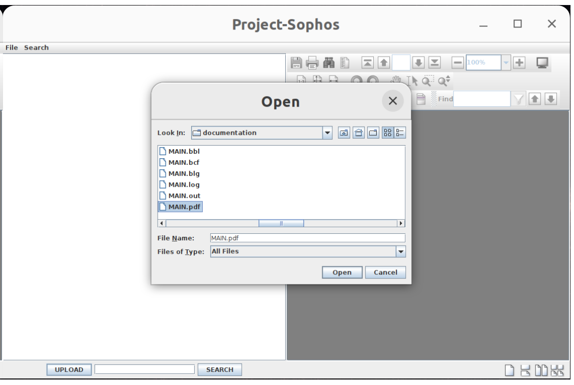
Figure 3.7: Illustration of the usage of the RMI infrastructure to abstract the Client/Server communication to function calls. The Client calls functions of the stub which will be propagated to the Server where the corresponding function is executed.

Usage

In the following we describe the intended usage of our Application from the view of the client which is depicted in Figure 3.8. Since we have a headless server implementation we only describe the client side, the server needs only to be started. To note is that in the current state the server and client need to be executed on the same machine, but the system already uses the network stack meaning that an external server is possible with a couple of changes in the source code. For our Application to function properly it is necessary for the server to be started first and only after the server is done with the initialization the client can be started. Since this is only a showcase how an SSE system can be integrated in a real life scenario we have no visual indication if something is not working properly, but it is necessary to look at the terminal output in the IDE. Figure 3.8a shows the client at the first execution with no documents stored in the system. On the left side is the result panel which will show a list of documents when the system has documents stored on the server and on the right side a PDF viewer where PDFs can be opened. The Client automatically fetches the document names from the server at start up, so the user can easily see what documents are stored on the server prior to the current session. Figure 3.8b shows the upload popup after pressing the upload button where a PDF document can be selected and uploaded to the server. In Figure 3.8c we see a search for the search word "Encryption" entered in the bottom left text field. We want to point out that the search words in this scheme are case-sensitive and only whole words can be searched. So the word "encryption" and "Encr" do not result in a match. After entering a word and pressing the Search button the system will send a query to the server, download the corresponding file and list it in the result panel on the left. The PDF can be shown in the integrated PDF viewer by selecting one document in the result panel.



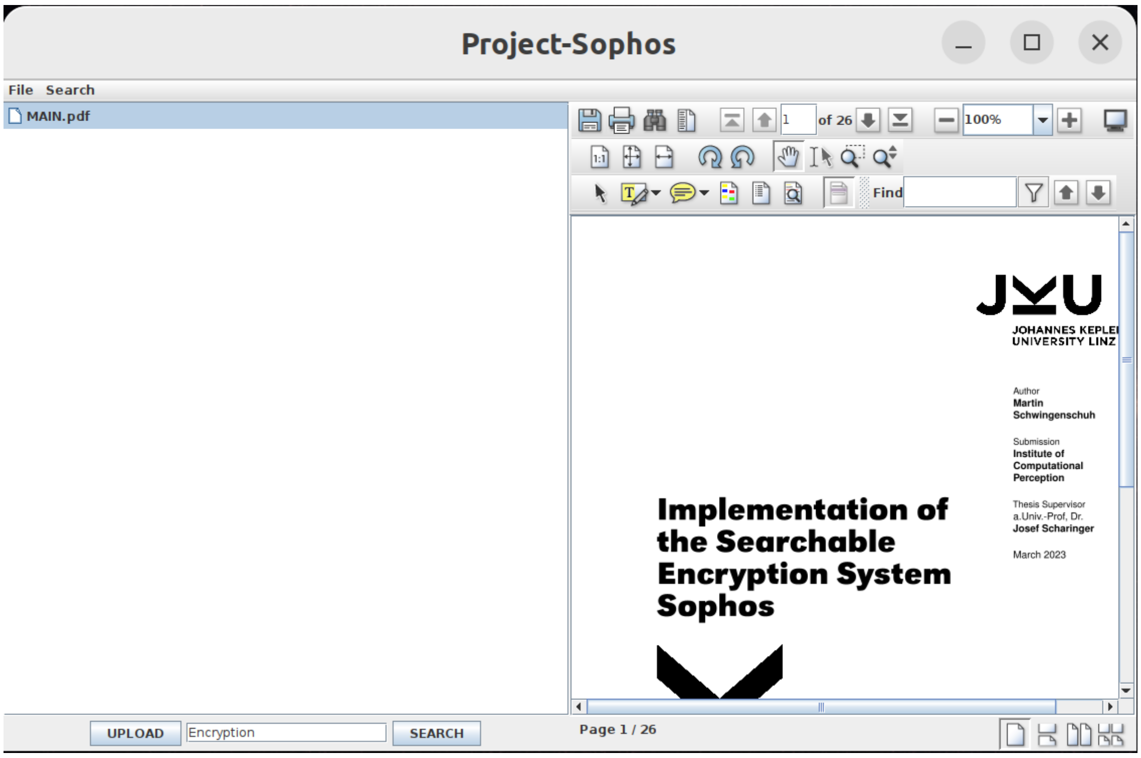
(a) Client View



File Name: MAIN.pdf

Files of Type: All Files

(b) Document Selection



Encryption

JKU
JOHANNES KEPLER UNIVERSITY LINZ

Author
Martin Schwingenschuh

Submission
Institute of Computational Perception

Thesis Supervisor
a.Univ.-Prof. Dr. Josef Scharinger

March 2023

Implementation of the Searchable Encryption System Sophos

(c) Search and Document Selection

Figure 3.8: Graphic of the User Interface of the Client Application with the first execution at (a) the document selection for the upload to the server in (b) and the result view when searching for a word and viewing a document in (c).

Bibliography

[1] Source Code: `https://github.com/MartinSchwingenschuh/Sophos_SE`

[2] Bost, R. $\Sigma o\varphi o\zeta$: Forward secure searchable encryption. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communication Security, CCS'16, Association for Computing Machinery, pages 1143-1154, 2016

[3] Bost, R., Minaud, B., and Ohrimenko, O. Forward and backward private searchable encryption from constrained cryptographic prim- itives. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, Association for Computing Machinery, pages 1465-1482, 2017.

[4] Goldreich, O., and Ostrovsky, R. Software protection and simula- tion on oblivious rams. J. ACM 43, 3 (may 1996.), pages 431-473.

[5] Garg, S., Mohassel, P., and Papamanthou, C. Tworam: Round- optimal oblivious ram with applications to searchable encryption. Cryptology ePrint Archive, Paper 2015/1010, 2015.

[6] Zahur, Samee and Wang, Xiao and Raykova, Mariana and Gascón, Adrià and Doerner, Jack and Evans, David and Katz, Jonathan, "Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation," 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, pp. 218-234, 2016.