UNIVERSITY OF PIRAEUS
DEPARTMENT OF DIGITAL SYSTEMS

Postgraduate Programme

"TECHNO-ECONOMIC MANAGEMENT &

SECURITY OF DIGITAL SYSTEMS"

"Dynamic Searchable Symmetric Encryption"

Thesis of

Daskalopoulos Ioannis

A.M. : 1312

Supervisor Professor: K. Lamprinoudakis

P. Rizomiliotis

Athens, 2015

Contents of Thesis

Chapter 1

Summary of Thesis

1.1. Summary of Thesis

The subject of this thesis is "Dynamic Searchable Symmetric Encryption" (DSSE). DSSE is one of the solutions that can be used for implementations of searchable encryption schemes. Searchable encryption allows a client to outsource the storage of its encrypted data to a server, while maintaining the ability to search over the data without downloading it. The encryption of the outsourced data, is part of the privacy that client requires. Moreover, the search over that data should not be done with plaintext queries. The queries are done through encrypted keywords, so that it is not possible for the cloud server to decrypt the plaintext query, based only on the received encrypted query.

1.2. Introduction

The thesis is structured as follows. The second chapter is an introduction to searchable encryption, whereas in the third one, the possible cryptographic primitives that can be used in order to achieve a searchable encryption scheme are analyzed. These solutions are also compared in terms of security and efficiency. In the fourth chapter, a thorough analysis of the principles that most DSSE schemes make use of is conducted and the necessary security definitions of a scheme are discussed.

The fifth chapter is an analysis of the first efficient DSSE proposal that could also meet the appropriate security definitions. Most of future proposals were based on this scheme. The sixth chapter is about Blind Storage scheme. Although not adopted by the majority of the researchers, it gave an idea on how a DSSE scheme could rely only on a cloud storage service, without the need of a cloud computation server.

Chapters seven and eight are the analysis of two schemes that were the main subject of this thesis. The first one was based on inverted indexes solutions, as most of DSSE schemes, while the second one was based on dictionary structures and seems to be one of the most efficient schemes until now. Both of them managed to support forward privacy.

The scope of this thesis was to understand DSSE theory, study and analyze these two schemes and then try to add some more features on them, such as the ability to execute Boolean queries on encrypted data and support multi-client scenarios. Both of these add-ons are analyzed in chapters ten and eleven. Furthermore, in chapter eleven the scheme is further extended in order to become more parallel in terms of search and in chapter twelve the scheme is made more secure, by not allowing the Dataowner, who authorize the queries, to learn the keyword of the queries, but only their attribute.

Chapter 2

Searchable encryption

In the cloud era, data outsourcing is becoming the dominant storage model. As businesses and even individuals have their data hosted by an untrusted storage service provider, data privacy has become an important concern. Over the years, the problem of encrypted search has become an important problem in security and cryptography. This is due to a combination of three things:

1) Search is now the primary way we access our data.
2) We are outsourcing more and more of our data to third parties.
3) We trust these third parties less.

Because of these reasons, the problem of encrypted search is now of interest to many sub-fields in computer science such as databases, security, cryptography and privacy. Moreover industry and governments start paying attention more and more to encrypted search.

While a-priori searching on encrypted data may sound impossible, there are a lot of ways known, in order to achieve it. Some methods are more efficient than others, some are more secure than others and some are more flexible.

One straightforward solution is to upload all data encrypted using one of the symmetric encryption techniques. However, by adopting this approach, the client has to download all its data and decrypt them in order to perform even simple computations, such as data search. Such a solution is not practical, even for small databases.

The problem of searching on encrypted data was first considered explicitly by Song, Wagner and Perrig in "Practical Techniques for Searches on Encrypted Data", from 2001. Relative work must be consider prior work on oblivious RAMs by Goldreich and Ostrovsky and on secure two-party computation by Yao who also provided solutions to this problem. Both of them are a lot less efficient nowadays.

Every techniques used in searchable encryption, like any cryptographic technology have tradeoffs that must be taken under consideration. A basic goal of this thesis is to make clarify these tradeoffs.

A simple Searchable Encryption scheme consist of at least two phases. Usually there is a Client, who is the Dataowner of the database and a Server who hosts the encrypted Database.

1) A setup phase, where the database is encrypted and loaded on a Server. The encryption is mandatory because the Server is not totally trusted by the Dataowner.

Figure 2.1 - Encryption of the Database

2) A search phase, where the Client uses tokens in order to query the Server, without revealing the plaintext of his query.

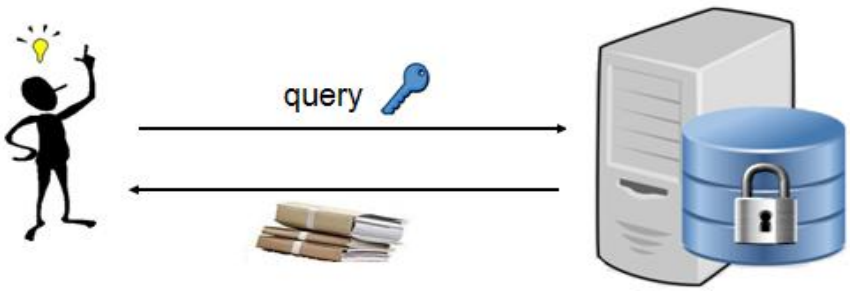


Figure 2.2 - Query on the Encrypted Database

The Setup Phase

During the setup phase, the client will take n documents (D_1,....,D_n) and generate an encrypted database (EDB) and a set of encrypted documents (c_1,....,c_n). Usually documents are encrypted using a standard encryption scheme such as AES. What searchable encryption should figure out, is the way that EDB should be constructed.

It is not necessary that encrypted documents and EDB are hosted in the Same Cloud Server. Encrypted documents could be stored in a Cloud Storage System by using appropriate labels, in order to retrieve them. The main purpose of the Server, that host the EDB, is to answer to Client's queries, based on his token, by retrieving the labels of the encrypted files that match the query. Finally, client should ask to download these labels from the Cloud Storage.

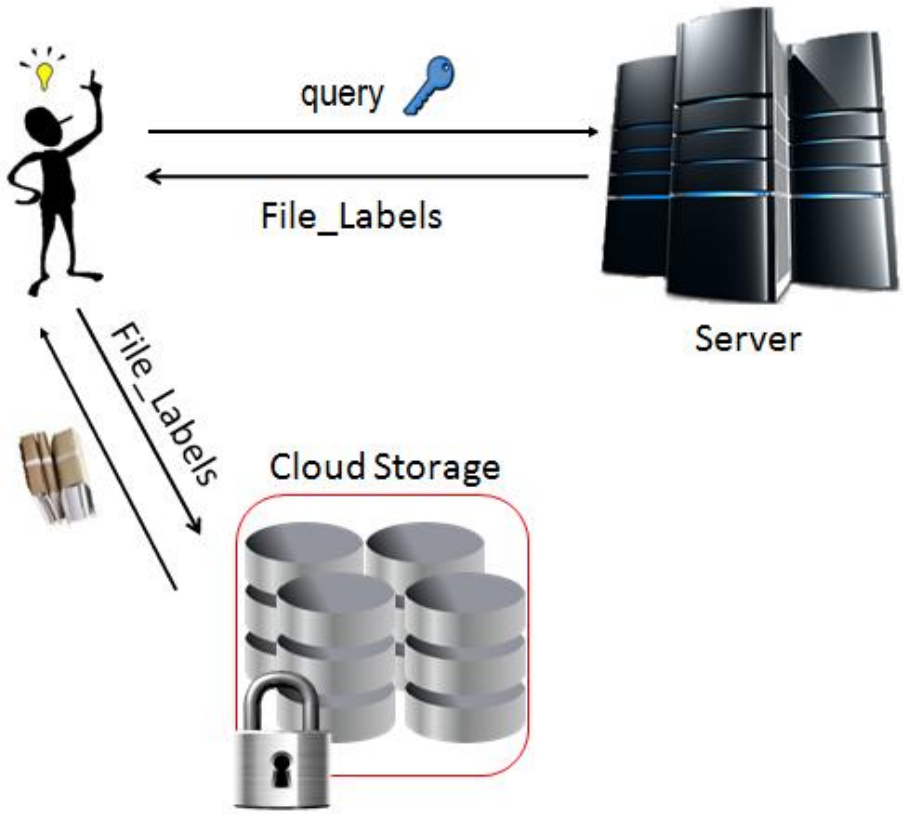


Figure 2.3 - Two steps of the Search phase

A concrete example is considering the documents as an e-mail collection. The client will generate an EDB and then encrypt each e-mail separately. After generating the EDB and encrypted documents, client will send EDB to the server and encrypted documents to Cloud Storage.

Search Phase

During the search phase, the client wants the server to send him back all the labels of the encrypted documents associated with a keyword w. To do this, the client will send a token that encapsulates w without revealing information about the plaintext of the keyword w. The server will then use the token with the EDB to somehow figure out which encrypted documents, or labels, it should send back to Client.

These two phases exist on every searchable encryption scheme. Six different ways are known for searching on encrypted data. Each way is based on one of the following cryptographic primitives:

- property-preserving encryption
- functional encryption
- fully-homomorphic encryption
- searchable symmetric encryption
- oblivious RAMs
- secure two-party computation

Every scheme has a tradeoff between efficiency, security and functionality. Some of them will be described briefly and some of them more thoroughly in the following chapters.

Chapter 3

Proposals for Searching in Encrypted Data

3.1. Deterministic Encryption

In this chapter, the simplest way to search on encrypted data will be covered. This is usually the solution people come up with when they first think of the problem of encrypted search. This approach has some nice properties, but on the other hand, there are also many limitations.

Deterministic Encryption needs a special type of encryption scheme called property-preserving encryption (PPE) scheme. PPE schemes encrypt messages in a way that leaks certain properties of the underlying message.

There are different types of PPE schemes that each leak different properties. The simplest form is deterministic encryption which always encrypts the same message to the same ciphertext. The property preserved by deterministic encryption is equality since, given two encryptions

$$c_1 = E_K(m_1) \quad \text{and} \quad c_2 = E_K(m_2)$$

then, by just checking if $c_1=c_2$, one can test if the underlying messages are equal.

PPE-based encrypted search was first proposed in the Database community and later studied more formally in the Cryptography community. The first scheme to provide a cryptographic treatment of this approach was by Bellare, Boldyreva and O' Neill from 2006.

We will now discuss a high-level idea of a Deterministic Encryption scheme. Suppose there exist both a deterministic encryption scheme E^D and a standard CPA-secure encryption scheme E^R. An encrypted database EDB can be created as follows.

For each document D_i in the collection $(D_1,.....,D_n)$, the client computes deterministic encryptions of each keyword of D_i. Assuming each document D_i has m keywords $(w_{i,1} ,.....,w_{i,m})$, the EDB then simply consists of n tuples,

$$r_i = (d_{i,1},....., d_{i,m}, ptr(c_i))$$

where $d_{i,j} = E^D_{K2}(w_{i,j})$, $c_i = E^R_{K1}(D_i)$ and ptr(c_i) is a pointer to ciphertext c_i.

Recall that in that scheme, the client sends the encrypted database

$$EBD=(r_1,.....,r_n)$$

to the server and the randomized encryptions of the documents $(c_1,.....,c_n)$ at the cloud Storage.

To search for keyword w, the client just sends a deterministic encryption of w to the server. This encryption of the keyword (d_w)

$$d_w = E_{K2}^{D}(w)$$

will serve as the token. Now all the server has to do is compare d_w to all the deterministic encryptions in EDB. If d_w is equal to any of them, the server follows the corresponding pointer and returns the encrypted document.

In other words, for all $1 \le i \le n$ and $1 \le j \le m$, the server tests if $d_w=d_{i,j}$ and if they are equal, then server sends the $ptr(c_i)$ back to client. This $ptr(c_i)$ is the label of the file c_i stored at Storage.

Obviously there is a limitation in the way the scheme is constructed. The search time for the server is O(nm). This means that the search time, even assuming that the set of keywords is small, is linear in the number of documents. Of course, linear-time search is too slow for practice. On the other hand, in reality this is not a problem, because the deterministic encryptions of EDB can be stored in data structures that support fast search, such as a binary search tree, so that search can be performed very quickly (O(log(n)).

As far as efficiency is concerned, deterministic schemes might be efficient. From Security point of view, there are some limitations with respect to security

The first problem is that the encrypted database EDB leaks a lot of information to the server about the data collection, even from the setup phase. During the search phase it leaked even more data. The keywords in EDB are encrypted using a deterministic encryption scheme. Because of that, the same keyword w will always encrypt to the same ciphertext. This means that if the server sees two or more equal ciphertexts in EDB, it knows that the corresponding encrypted documents contain a keyword in common.

In addition to the previous leakage, the server learns the frequency with which keywords appear. This makes the encrypted database vulnerable to frequency analysis.

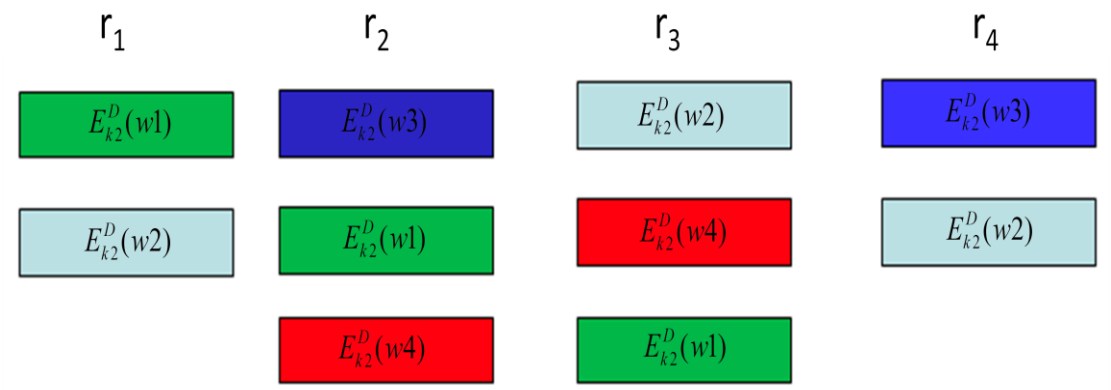


Figure 3.1 - EDB of Deterministic Encryption

Another issue is that since tokens are deterministic encryptions of the search terms, the server will always know whether the client is repeating a search a not. This is a result of the fact that the encryption of keyword w_1, will always be the same ciphertext.

A final issue occurs when the deterministic encryption scheme uses public-key. In this case, all the deterministic encryptions, including the EDB and the tokens, are encrypted under the client's public key which is, obviously, public and available to the server. The server can then mount a dictionary attack on the encrypted database by encrypting a list of possible keywords and comparing them to the ones found in EDB and in the tokens. If it gets a match then Server can learn the plaintext of the keyword.

This attack clearly shows that public-key PPE-based solutions should probably not be used for "normal" data such as text and emails. But it doesn't mean that they should not be used at all. As Bellare et al. observe , such a solutions can be used when the data has high min-entropy. High min-entropy means that the data looks random to the server. The problem of course is that it is not clear when this applies in practice. Also, note that even if the keywords do have high min-entropy, the other security issues still remain.

Concluding, an approach for searching over encrypted data was presented, which is based on property-preserving encryption. It supports fast search on encrypted data, but on the other hand, the scheme leaks quite a bit of information to the server.

3.2 Functional Encryption

Now, a different approach that provides the opposite properties from the previous chapter, will be described. It has slow search, but better security. At a high-level, one can view this approach as simply replacing the PPE scheme from the previous solution with a functional encryption (FE) scheme.

The notion of Functional Encryption was first described by Sahai and Waters and later formalized by Boneh, Sahai and Waters and by O'Neill . Starting with the work of Boneh and Franklin on identity-based encryption (IBE), there was a slew of new encryption schemes achieving various properties, such as attribute-based encryption, hidden vector encryption or predicate encryption. Though everything that will be presented can be done with Functional Encryption, for concreteness, the special case of Identity Based Encryption will be considered, which was first suggested by Shamir and realized by Boneh and Franklin.

A public-key Identity Base Encryption (IBE) scheme consists of four algorithms:

1. A setup algorithm "SETUP" is used to generate a master secret and public key pair (msk , mpk).

2. An encryption algorithm "ENC", which takes as input the master public-key mpk, an identity id and a message m as input and returns a ciphertext c.

3. A key generation algorithm "KEYGEN" that takes as input the master secret key msk and an identity id and returns a secret key sk_{id}.

4. And finally a decryption algorithm "DEC" that takes as input a secret key sk_{id} and a ciphertext c and returns a message m or a failure symbol $\perp$.

The motivation behind IBE is key distribution. In particular, using an IBE scheme should be easier than using a standard (public-key) encryption scheme where public keys have to be certified, revoked and verified.

In order to better understand how this scheme is working a example will be presented. Suppose Alice wants to send an encrypted message to Bob who works at Microsoft. The idea is that Microsoft would first generate a pair of master keys (msk, mpk) and distribute mpk together with a certificate. To send her message m, Alice would retrieve Microsoft's master public key mpk, verify Microsoft's certificate and then encrypt m under Bob's identity (bob@microsoft.com):

c=Enc(mpk, "bob@microsoft.com",m)

To decrypt the ciphertext c, Bob needs to hold a secret key for his identity under Microsoft's master key. So, he needs to create his decryption key sk by:

sk=Keygen(msk, "bob@microsoft.com")

Now, Bob can recover the message by computing:

m=Dec(sk,c)

Notice that Alice never needed to know what Bob's public key was or to verify any certificate for his key. The only certificate she had to verify was for Microsoft's master public key but once that key is authenticated, Alice can send email to anyone at Microsoft without any additional work.

Continuously it will be shown how (anonymous) IBE can be used to search over encrypted data. This idea was first proposed by Boneh, Di Crescenzo, Ostrovsky and Persiano and is best explained by considering again the following email scenario where Alice wants to send an encrypted email to Bob.

1) Bob generates a master secret and public key pair for the IBE scheme (msk,mpk)
2) Bob generates a secret and public key pair for a standard public-key encryption scheme (sk,pk).

3) Bob reveals the public keys (mpk , pk) publicly and keeps the secret keys (msk , sk) private.

4) Alice encrypts her message under pk using the standard public-key encryption scheme, resulting in a ciphertext c.

5) She then attaches IBE encryptions of "1" under Bob's master public key mpk with the keywords as the identity. This results in a set of IBE encryptions (e_1,.....,e_m) where each e_j (for $1 \leq j \leq m$) is defined as:

e_j=ENC(mpk,w_j,1) , where (w_1,.....,w_n) are the keywords.

6) When Email server will have received n emails of this form, he will holds a set of encrypted emails (c_1,.....,c_n) and an encrypted database (EDB).

7) Now, if Bob wants to retrieve the emails with keyword w, he just needs to generate a secret IBE key as:

$$sk_w = Keygen(msk, w)$$

and send it as the token to the server.

8) The server then tries to decrypt each IBE ciphertext in EDB. When it is successful, it send the associated label back to client. Client will request the label from the Cloud Storage.

An important observation is that a standard IBE scheme here will not be enough. The problem is that the notion of IBE does not necessarily guarantee that a ciphertext hides information about the identity used to create it. This means that, if a standard IBE scheme is used, EDB could leak the keywords to the server.

To address this, Boneh et al. observe that what you actually need is an anonymous IBE scheme which essentially means that the ciphertexts do not reveal information about the identities. Fortunately, it is known how to construct such schemes efficiently, so this is not a major concern from a practical point of view.

The Drawback of this scheme is Search time. The Search time for the server is O(nm) since it has to try to decrypt each ciphertext in the EDB. Even Assuming that m<<n, the search time is O(n), which is a a lot slower than the solution based on deterministic encryption described in the previous post which required time O(log(n)) which is sub-linear in n.

From Security point of view, although this approach is slower than the PPE-based approach, it has better security properties. First, the encrypted database by itself does not reveal much useful information to the server since—unlike the deterministic approach—keywords are encrypted using a randomized identity-based encryption scheme. So, even if two documents have keywords in common, the encrypted keywords in EDB will be different. This means that it is not needed to make unnatural assumptions about the data, such as high entropy, in order to use it safely.

On the other hand there is a security issue, with this approach. It does not protect the search terms. Particularly, the server could launch the following attack to figure out which keyword the client is searching for.

Suppose the server has some dictionary W of d words. For each keyword $w \in W$ it encrypts "1" with key mpk and identity w. This results in a set of d (identity-based) encryptions (e'_1,.....,e'_d). Now, given some token sk_w, the server can learn w by simply trying to decrypt each of the ciphertext e'_i with sk_w. If the decryption works for some e'_i, then the server knows that sk_w is for the identity used to generate e'_i.

The previous attack does not result from a deficiency of any particular IBE scheme, but that it applies to any public-key encrypted search solution. The fundamental problem is that the server has both the ability to create EDBs, since it has the public-key and to search over them. The conclusion is that, search on publicly-encrypted data cannot protect search terms.

The solution to that security issue was given recently by Boneh, Raghunathan and Segev and Arriaga and Tang who design public-key encrypted search solutions that achieved the best possible level of confidentiality for search terms. Those schemes suppose that search terms are hard enough to guess in order the schemes proposed to be able to protect them. If that assumption does not exist there is no known solution to the problem. In that case using symmetric solution is the only secure solution, since only the client can generate EDBs.

3.3 Oblivious RAM

Until now, two approaches were presented, in order to search on encrypted data. The first one was the Property Preserving Encryption approach, which resulted in schemes with fast search (O(log(n)), but with relatively weak security characteristics. The second one, which was the FE-based approach, resulted in schemes with slow search (O(n)), but with better security guarantees.

In the chapter, solutions that are even slower will be presented, but which on the other hand achieve the strongest possible levels of security. In order to have a better understanding about what is secure, while speaking about searchable encryption, it should be mentioned which security properties were initially proposed and which we should expected from an encrypted search solution.

- The encrypted database EDB generated by the scheme, should not leak any information about the database DB of the user

- The tokens t_w generated by the user, should not leak any information about the underlying search term w to the server.

This definition of security is reasonable, but there are several issues. There are many details that impact security that are not taken into account in this high-level intuition. For example, what does it mean not to leak information? This is why cryptographers are so pedantic about security definitions. The details really do matter and so security definitions must be very accurate.

Moreover the previous security definition mention nothing about the search results. More precisely, it does not specify whether it is appropriate or not for an encrypted search solution to reveal to the server which encrypted documents match the search term. We usually refer to this information as the client's access pattern and for concreteness you can think of it as the matching encrypted documents' identifiers or their locations in memory. All is really needed as an identifier, is a per-document unique string ($label_f$), that is independent of the contents of the document and of the keywords associated with it (w).

So, is it appropriate to reveal the access pattern? There are two possible answers to this question. Some researchers believe that it is fine to reveal the access pattern since the whole point of using encrypted search is so that the server can return the encrypted documents that match the query. And if it is expected from the server to return those encrypted documents, then it clearly has to know which ones to return, without leaking any information about the content of the document. On the other hand, one could argue that, in theory, the access pattern reveals some information to the server. In fact, by continuously observing search results the server could use some sophisticated statistical attack to infer something about the client's queries and data.

Furthermore, the argument that the server needs to know which encrypted documents match the query in order to return the desired documents is not technically true. In fact, it is known how to design cryptographic protocols that allow one party to send items to another without knowing which item it is sending. With private information retrieval and oblivious transfer, this principle can be achieved.

Nowadays, it is known how to design systems that allow us to read and write to memory without the memory device knowing which locations are being accessed. The latter are called oblivious RAMs (ORAM) and can be used for searching on encrypted data without revealing the access pattern to the server. Oblivious RAM (ORAM) is a cryptographic primitive designed to conceal access patterns, when a client with small local memory executes a sequence of reads and writes to remotely stored data. An ORAM algorithm is secure, if for any two access patterns the client performs, the corresponding two physical access patterns sequences are computationally indistinguishable. The issue, of course, is that using ORAM will slow things down and so there are not efficient schemes.

So, the answer to the previous question depends on what kind of tradeoff we are willing to make between efficiency and security. If efficiency is the priority, then revealing the access pattern might not be too much to give up in terms of security for certain applications. On the other hand, if some inefficiency is tolerable, then it is always best to be conservative and not reveal anything. In this chapter, ORAMs structures are explored and the security analysis of those schemes is presented.

Oblivious RAM was firstly proposed by Goldreich and Ostrovsky for software protection. That work turned out to be really ahead of its time as several ideas explored in it turned out to be related to more modern topics like cloud storage.

An ORAM scheme consists of three phases, SETUP, READ, WRITE.

- A Setup algorithm that takes as input a security parameter 1^k and a memory (array) RAM of N items. It outputs a secret key K and an oblivious memory ORAM.

- A two-party protocol READ executed between a client and a server that works as follows. The client runs the protocol with a secret key K and an index i as input, while the server runs the protocol with an oblivious memory ORAM as input. At the end of the protocol, the client receives RAM[i] while the server receives $\perp$ (nothing). This is written sometimes as $\mathrm{Re}\,ad((K,i),ORAM) = (RAM[i],\perp)$.

- A two-party protocol WRITE executed between a client and a server that works as follows. The client runs the protocol with a key K, an index i and a value "u" as input and the server runs the protocol with an oblivious memory ORAM as input. At the end of the protocol, the client receives $\perp$ and the server receives an updated oblivious memory ORAM' such that the i^{th} location now holds the value "u":

$$Write((K,i,u),ORAM) = (\perp, RAM').$$

Usually ORAM schemes are designed by using fully-homomorphic encryption or symmetric encryption. The simplest way to design ORAM is FHE, and so an overview will be presented.

Suppose that there exists an FHE scheme FHE=(Gen, Enc, Eval, Dec). Then it is easy to construct an ORAM as follows:

- Setup(1^k , RAM): Generate a key for the FHE scheme by computing K=FHE.Gen(1^k) and encrypt RAM as c=FHE.Enc$_k$(RAM). Output c as the oblivious memory ORAM.

- Read((K,i),ORAM): The client encrypts its index i as c_i=FHE.Enc$_K$(i) and sends c_i to the server. The server computes:

  $$c'=FHE.Eval(f,ORAM,c_i)$$

  where f is a function that takes as input an array and an index i and returns the i^{th} element of the array. The server returns c' to the client who decrypts it to recover RAM[i].

- Write((K,i,u),ORAM): The client encrypts its index i as c_i=FHE.Enc$_K$(i) and its value as c_u=FHE.Enc$_K$(u) and sends them both to the server. The server computes:

  $$c'=FHE.Eval(g,ORAM,c_i,c_u)$$

  where g is a function that takes as input an array, an index i and a value u and returns the same array with the i^{th} element updated to u.

The security properties of FHE will guarantee that ORAM leaks no information about RAM to the server and that the READ and WRITE protocols reveal no information about the index and values either.

The obvious downside of this FHE-based ORAM is efficiency. Even supposing that there is a very fast FHE scheme, this ORAM would still be too slow simply because the homomorphic evaluation steps in the READ and WRITE protocols require O(N) time, which is linear in the size of the memory. Again, assuming we had a very efficient FHE scheme, this would only be usable for small memories.

After having understand how to build an ORAM, it will be explained how to use ORAM for encrypted search. There are two possible ways to do this:

First approach

A naive approach could be the client to just dump all the n documents $\mathbf{D}=(D_1,.....,D_n)$ in an array RAM, setup an ORAM by:

$$(K, ORAM)=Setup(1^k,RAM)$$

and send ORAM to the server. For the search phase, the client can just simulate a sequential search algorithm via the READ protocol, by replacing every read operation of the search algorithm with an execution of the READ protocol. To update the documents the client can similarly simulate an update algorithm using the WRITE protocol.

This approach is obviously very slow. Assuming that all the documents have bit-length d and that RAM has a block size of B bits. The document collection will then fit in $N = n \cdot d \cdot B^{-1}$ blocks. The sequential scan algorithm is itself O(N), but on top of that, an entire READ protocol should be executed for every address of memory read. Additionally, if the FHE-based ORAM described above is used, which requires O(N) work for each READ and WRITE, then a single search would take $O(N^2)$ time!

Second approach

A better approach is for the client to build two arrays RAM_1 and RAM_2. In RAM_1 it will store a data structure that supports fast searches on the document collection, such as an inverted index. In RAM_2 it will store the documents $\mathbf{D}$ themselves. It then builds and sends:

$$ORAM_1=Setup(1^k,RAM_1)$$

and

$$ORAM_2=Setup(1^k,RAM_2)$$

to the server. For the search operation, the client simulates a query to the data structure in $ORAM_1$ via the READ protocol. This can be done by replacing each read operation in the data structure's query algorithm with an execution of READ. From this operation, the client will recover the identifiers of the documents that contain the keyword and with this information it can just read those documents from $ORAM_2$.

There are also works that tradeoff client storage for access efficiency. For example, Williams, Sion and Carbunar propose a solution with O(logN*loglogN) amortized access cost and $O(\sqrt{N})$ client storage while Stefanov, Shi and Song propose a solution with O(logN) amortized overhead for clients that have O(N) local storage, where the underlying constant is very small.

There is also a line of work that tries to de-amortize ORAM in the sense that it splits the re-structuring operation so that it happens progressively over each access. This was first considered by Ostrovsky and Shoup in and was further studied by Goodrich, Mitzenmacher, Ohrimenko, Tamassia and by Shi, Chan, Stefanov and Li .

All in all this may not seem that bad and, intuitively, the two-RAM solution might actually be reasonably practical for small to moderate-scale data collections, especially considering all the recent improvements in efficiency that have been proposed. However, for large scale collections, other schemes should be used.

Concluding, the ORAM-based solution for encrypted search were presented, which provides the most secure solution to our problem since it hides everything, even the access pattern. For the rest of the thesis, a different approach will be analyzed for searching on encrypted data, which strikes to balance between efficiency and security. In particular, this solution is as efficient as the deterministic-encryption-based solution, while being only slightly less secure than the ORAM-based solution.

3.4 Searchable Symmetric Encryption

Searchable Symmetric Encryption (SSE) was first introduced by Song, Wagner and Perrig in 2001. SSE tries to achieve the best of all worlds. It is as efficient but provides a lot more security than most of other schemes. Two basic keys for SSE are security definitions and leakage which will be analyzed in the following chapter. Leakage is an important issue that was overlooked in previous work. As it was explained before, non-ORAM solutions leak some information. Usually SSE schemes reveal the search results, which are the identifiers of the documents that contained the keyword. This was the whole point of SSE and most people wrongly believed that this was why it was more efficient than ORAM.

There are many variants of SSE including interactive schemes, where the search operation is interactive such as two-party protocol and response-hiding schemes, where search results are not revealed to the server but only to the client.

For simplicity, the document collection itself will be ignored and will be just assumed that the individual documents are encrypted using some symmetric encryption scheme and that the documents each have a unique identifier ($label_f$), which is independent of their content.

Now assuming that the client processes the data collection $\mathbf{D}=(D_1,.....,D_n)$ by setting up a "database" DB . That Database maps every keyword w in the collection, to the identifiers of the documents that contain it. For a keyword w, DB[w] will refer to the list of identifiers of documents that contain w.

A non-interactive and response-revealing SSE scheme consists of the three following algorithms.

1. a SETUP algorithm executed by the client, that takes as input a security parameter 1^k and a database DB. It returns a secret key K and an encrypted database EDB.

2. A TOKEN algorithm, which is also run by the client and takes as input a secret key K and a keyword w. It returns a token tk.

3. A SEARCH algorithm executed by the server, that takes as input an encrypted database EDB and a token tk. It returns a set of identifiers DB[w].

In addition to security, the most important thing that is expected from an SSE solution is to have low search complexity. Low search complexity means that the solution is efficient. For our purposes fast will mean sub-linear in the number of documents. Moreover, ideally, it should be linear in the number of documents that contain the search term. Note that the latter is optimal since at a minimum the server needs to fetch the relevant documents just to return them. Requiring sub-linear search complexity is fundamental for practical purposes.

However, the sub-linear requirement has consequences. Particularly it means that working in a offline/online setting cannot be avoided, where an one-time pre-processing phase is executed in order to setup a search structure in linear time. On the other hand, it would be possible then to execute search queries on the data structure in sub-linear time. This is exactly the approach that most researchers follow.

Because DSSE is the main subject of this thesis, it will be explained analytically in the following chapters. At the next chapter, the principles of the majority of DSSE schemes will be presented thoroughly and the security definitions, which must be taken under consideration at DSSE schemes, will also be mentioned.

Chapter 4

Principles of DSSE and security definitions

4.1 DSSE Principles

In the cloud era, as more and more businesses and individuals have their data hosted by an untrusted storage service provider, data privacy has become an important concern. In this context, searchable symmetric encryption (SSE) has gained a lot of attention. Searchable symmetric encryption allows a client to encrypt its data in such a way that this data can still be searched. So, it aims to protect the privacy of the outsourced data by supporting, at the same time, outsourced search computation. The most immediate application of SSE is to cloud storage.

SSE has been the focus of active research and a multitude of schemes that achieve various levels of security and efficiency have been proposed. Any practical SSE scheme, however, should, at a minimum satisfy the following properties:

- Sublinear search time,
- Security against adaptive chosen keyword attacks,
- Compact indexes and
- Ability to add and delete files efficiently.

Unfortunately, the design of an efficient SSE has been shown to be a challenging task. The first schemes that has been proposed on SSE couldn't achieve all these properties at the same time. This was severely limiting the practical value of SSE and decreased its chance of deployment in real-world cloud storage systems. Nowadays, after some years of research on Searchable Symmetric Encryption, there are finally schemes that can achieve these properties in a very efficient way that can achieve a search in less than 100ms.

Most efficient approaches of SSE use one or more inverted indexes. Because of its popularity we will give an example of an inverted index approach. Assuming that there are four files $\mathbf{f}=(f_1,f_2,f_3,f_4)$ and 5 keywords, that can be used for single-word queries, $w=(w_1,w_2,w_3,w_4,w_5)$. The keywords that its file contains are:

$$f_1\{w_1,w_4,w_5\}$$
$$f_2\{w_2,w_3,w_4\}$$
$$f_3\{w_1,w_3\}$$
$$f_4\{w_2,w_3\}$$

Based on the above, an inverted index for that SSE would be the follow:

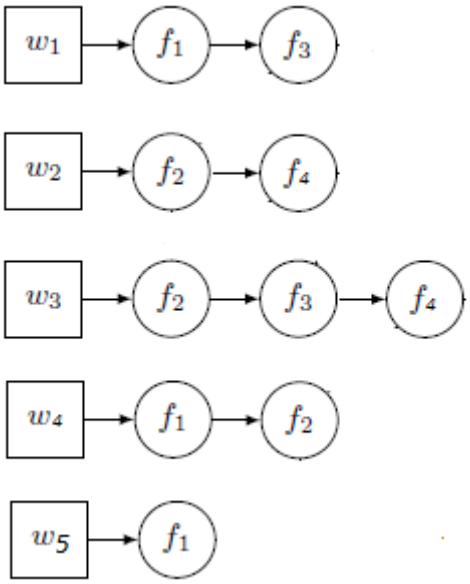


Figure 4.1 - Inverted Indexes

Now, when there is a search query for the keyword w_4, EDB will learn the position of the first tuple that contains that keyword. Which stores the label of f_1 and then it will be revealed the position of the second tuple which will contain the label of the file f_2.

While the inverted index approach yields the most efficient SSE schemes to date, it has at least three important limitations.

- The first is that it is not well-suited to handle dynamic collections and so complex constructions are required.
- In addition, the update operations reveal a non-trivial amount of information.
- The third limitation is that it is inherently sequential, requiring a lot of time even in a parallel model of computation. This happens mainly due to the fact that the majority of SSE schemes require the search algorithm to access a sequence of memory locations, each of which is unpredictable and stored at the previous location in the sequence.

Motivated by advances in multi-core architectures, some researchers presented a new method for constructing sub-linear SSE schemes, which is highly parallelizable and dynamic.

SSE schemes might have a lot of add-ons that researchers are currently focus on. One category, that was already mentioned is Dynamic SSE, which supports adding and removing documents at any point during the life-time of the system. Sometimes there is no offer for a specific operation for initial outsourcing and it is assumed that all data are added incrementally. The vast majority of the proposed schemes have a setup phase where an encrypted index is computed for the specific collection of documents, i.e. each keyword is related to a precomputed set of file identifiers. So, if the index remains unaltered after this phase, then the SSE scheme is called static SSE. If additions and deletions are supported, then it is a dynamic SSE (DSSE) scheme.

Moreover, there are protocols of DSSE that present the design, analysis and implementation of schemes that can support conjunctive search and general Boolean queries on outsourced symmetrically-encrypted data These schemes can scale to very large databases and arbitrarily structured data including free text search. Boolean query is not a single keyword, but it can be a formula like the following one : x AND y NOT z

A second extension is the support of multi-client capabilities. At this scenario there is a Dataowner who execute the SETUP Phase and ADD or DELETE phase. But there are also clients that can be authorized be the Dataowner to make queries at the EBD. Basically Dataowner (D) provides search tokens to clients based on their queries and according to a given authorization policy. Security considers multiple clients acting maliciously and possibly colluding with each other and a semi-trusted server E which acts as "honest-but-curious", but does not collude with clients.

The issue initially with the previous extension was that the Dataowner was learning the keywords of the query. This was a unwanted leak, especially for certain cases. At least, nowadays there are surveys which have investigated that issue and have proposed new solutions. In those solutions the dataowner D outsources its data to a server E, but D is now interested to allow clients to search the database such that clients learn the information D authorizes them to learn but nothing else while E still does not learn about the data or queried values as in the basic SSE setting. Usually in those cases the Dataowner learns only the attribute that the keywords of the query belongs to.

On the other hand, while most constructions have theoretically optimal search times that scale only with the number of documents matching the query, the performance of their implementations on large datasets is less clear. Factors like I/O latency, storage utilization, and the variance of real world dataset distributions degrade the practical performance of theoretically efficient SSE schemes. One critical source of inefficiency in practice often ignored in theory is a complete lack of locality and parallelism: To execute a search, most SSE schemes read sequentially each result from storage at a pseudorandom position, and the only known way to avoid this while maintaining privacy involves padding the server index to a prohibitively large size. Research tries to focus on those issues in order to make DSSE schemes even more efficient in real world environments.

SSE schemes found numerous applications, such as searching one's encrypted files stored at Amazon S3 or Google Drive, without leaking much information to Amazon or Google.

A criticism from some researchers is that SSE constructions are not really searching over data. The underlying issue is that no computation is being performed. Probably this reflects a very uninformed understanding of the real world. Given the amounts of data which are currently produced and have to search over, search has become analogous to sub-linear-time search and therefore to some form of indexed-based search. So, the kind of scale we now have to deal with has fundamentally changed what we usually mean by the term "search".

4.2 Security Definitions

Security definitions are of the most interesting aspects of encrypted search, from a research point of view. The first scheme to explicitly address this question was by Eu-Jin Goh. It had many contributions, but one of the most important ones was simply to point out that SSE schemes were not normal encryption schemes and, therefore, the standard notion of CPA-security was not meaningful for SSE. The problem that he pointed out was essentially that, when an adversary interacts with an SSE scheme, he has access to more than an encryption oracle. As a result, he also has access to a search oracle. Goh's point was that this had to be captured in the security definition otherwise it was meaningless.

Moreover, he proposed the first security definition for SSE, in order to address this. The definition that he proposed guaranteed that given an EDB and the encrypted documents, the adversary would learn nothing about the underlying documents beyond the search results even if it had access to a search oracle. Unfortunately, Goh's definition was a game-based definition and it did not provide query. A follow up scheme by Chang and Mitzenmacher proposed a new definition that was simulation-based and that guaranteed query privacy in addition to data privacy.

It should be mentioned here that, simulation-based definitions have some advantages over game-based definitions and are preferable because they are easier to work, with especially when composing various primitives to build larger protocols. Moreover, Reza Curtmola, Juan Garay, Rafail Ostrovsky and Kamara were also noticed that the previous security definitions did not seem to really capture all the security parameters that could affect a SSE scheme. There were primarily two issues:

1) The security definitions were restricting the adversary's power.
2) They did not explicitly capture the fact that the constructions were leaking information, during every phase of the protocols.

Those security definitions will be further analyzed, beginning from adaptivity. The first problem was that in these definitions, the adversary was never given neither the search tokens nor the EDB or the results of its searches. The implication of this was that the adversary could not choose its search oracle queries as a function of the

EDB. Nor as a function of the tokens that he used or previous search results. So, adversary's behavior was being implicitly restricted to making non-adaptive queries to its search oracle. In the real world the adversary, we are trying to protect against, is a server that stores the EDB, that receives tokens from the client and that sees the results of the of the search. So, clearly this was an issue.

By allowing the adversary to also query a search oracle, he must be allowed to query the oracle as a function of the EDB or the tokens and previous search results. When a SSE scheme is secure based on that, then even in the real world, it is secured from an attack where the server send some oracle queries based on the EDB, the tokens or previous search results.

Such a security might seem excessive by non-cryptographers, but it is considered to be mandatory. Usually when a protocol, that uses weak primitives, is hacked by a more sophisticated attacks, it should be redesigned and patched, resulting to an expensive protocol. This has happened in the cases of encryption (CPA- vs. CCA2-security) and key exchange.

In any case, surveys have been written about that stronger definition, where the adversary was allowed to generate its queries as a function of the EDB, the tokens and previous search results. It is called adaptive security. Even though cryptographic community is not aware of an explicit attack on a concrete SSE construction that takes advantage of adaptivity, this fact should not matter anymore because nowadays it is known how to construct adaptively-secure SSE schemes that are as efficient as non-adaptively-secure ones. Moreover, another important reason to consider adaptive security is for situations where SSE schemes are used as building blocks in larger protocols. In such situations, the primitive can be used in unorthodox ways, which open up more new oracles that one may not have considered.

Another important issue that was overlooked in previous work is leakage. It is a fact that non-ORAM solutions leak some information. Everyone was basically aware that SSE revealed the search results, which are the identifiers of the documents that contained the keyword (labelf). The problem was that the definitions did not capture any of this leakages. To address it researchers decided to treat leakage in SSE more formally and to capture it very explicitly in their security definitions. Leakage will be explained for every protocol that will be presented in that thesis.

Initially it was considered that leakage has two types.

- The access pattern
- The search pattern.

The access pattern is basically the search results . It is the identifiers of the documents that contain the keyword of the query. The search pattern is whether a search query is repeated. For the SSE, there are two more kinds of leakages:

- Setup leakage, which is revealed just by the EDB
- Search leakage, which is revealed by a combination of the EDB and a token.

Comparing a solution based on deterministic encryption or on property-preserving encryption which have a high degree of setup leakage, SSE-based solutions are better because their setup leakage is usually minimal. Furthermore, the query leakage is controlled by the client, since queries can only be executed with knowledge of the secret key.

Finally the last two security notions that should be mentioned are:

- Forward Privacy
- Backward Privacy

Forward privacy exists when a new keyword and file identifier pair is added and the server does not learn anything about this pair. In backward privacy, queries cannot leak the file identifiers of deleted documents. Based on the fact that search query token are deterministic, the adversary (Server with memory) can learn if that keyword was existing in a document that have been deleted. Designing an efficient DSSE scheme that possesses both these security properties has been shown to be a very difficult task. During the last years, some schemes do exist that fulfill the requirements of Forward Privacy, but none for Backward Privacy.

Summing up, although there is not a good way to understand and analyze the leakage of SSE schemes, for now, the best solution is to describe it precisely. What should be expected from an SSE security definition is a guarantee that the adversary cannot learn anything about the data and the queries beyond the explicitly allowed leakage, even if the adversary can make adaptive queries to a search oracle. A scheme that is secure against adaptive chosen-keyword attacks (CKA2) guarantees security even when the client's queries are based on the encrypted index and the results of previous queries. Whereas with CKA1 could not guarantee this.

After having discussed the security definitions and understood the principles of DSSE schemes, in the next chapters four DSSE proposals are presented.

Chapter 5

First analysis of a DSSE proposal "Dynamic Searchable Symmetric Encryption"

5.1 Introduction

The first proposal that will be discussed is the " Dynamic Searchable Symmetric Encryption" of Kamara, Papamanthou and Roeder. It was chosen for presentation because of its, which is outlined as follows:

1. They presented a formal security definition for dynamic SSE. In particular, their definition captures a strong notion of security for SSE, which is adaptive security against chosen-keyword attacks (CKA2).
2. They construct the first SSE scheme that is dynamic, CKA2-secure and also achieves optimal search time. Unlike previously known schemes their construction is secure in the random oracle model.
3. They described the first implementation and evaluation of an SSE scheme based on the inverted index approach that seems to be very efficient.

The overview of their index-based SSE proposal is that the encryption algorithm takes as input an index δ and a sequence of n files $\mathbf{f}=(f_1,.....,f_n)$ and outputs an encrypted index γ and a sequence of n ciphertexts $\mathbf{c}=(c_1,.....,c_n)$. All constructions can encrypt the files **f** using any symmetric encryption scheme. To search for a keyword w, the client generates a search token τ_w and given γ and c, the server can find the identifiers I_w of the files that contain the keyword w. From these identifiers it can recover from the cloud storage the appropriate ciphertexts c_w.

If the scheme is used as dynamic SSE it should be allowed addition and removal of files. Both of these operations are handled using tokens. To add a file f, the client generates an add token τ_a and given τ_a and γ , the provider can update the encrypted index γ. Similarly, to delete a file f, the client generates a delete token τ_d, which the provider uses to update γ. It is assumed that each file has a unique identifier $id(f_i)$ or it could also be denoted as $label_f$. The files do not have to be text files but can be any type of data, as long as there exists an efficient algorithm that maps each document to a file of keywords from W.

5.2 Algorithm

A dynamic index-based SSE scheme is a tuple of nine polynomial-time algorithms SSE = (Gen, Enc, SrchToken, AddToken, DelToken, Search, Add, Del, Dec) such that:

- $K \leftarrow Gen(1^k)$: is a probabilistic algorithm that takes as input a security parameter k and outputs a secret key K.
- $(\gamma,\mathbf{c}) \leftarrow Enc(K,\mathbf{f})$: is a probabilistic algorithm that takes as input a secret key K and a sequence of files **f**. It outputs an encrypted index γ, and a sequence of ciphertexts **c**.
- $\tau_s \leftarrow SrchToken(K,w)$: is an algorithm that takes as input a secret key K and a keyword w. It outputs a search token τ_s.
- $(\tau_a,c_f) \leftarrow AddToken(K,f)$: is an algorithm that takes as input a secret key K and a file f. It outputs an add token τ_a and a ciphertext c_f .
- $\tau_d \leftarrow DelToken(K,f)$: is an algorithm that takes as input a secret key K and a file f. It outputs a delete token τ_d.
- $I_w := Search(\gamma,\mathbf{c},\tau_s)$: is a deterministic algorithm that takes as input an encrypted index γ , a sequence of ciphertexts **c** and a search token τ_s. It outputs a sequence of identifiers I_w, that belongs to **c**.
- $(\gamma',c'):=Add(\gamma,\mathbf{c},\tau_\alpha,c)$: is a deterministic algorithm that takes as input an encrypted index γ, a sequence of ciphertexts **c**, an add token τ_a and a ciphertext c. It outputs a new encrypted index γ' and sequence of ciphertexts **c**'.
- $(\gamma',c'):=Del(\gamma,\mathbf{c},\tau_d)$: is a deterministic algorithm that takes as input an encrypted index γ, a sequence of ciphertexts **c**, and a delete token τ_d. It outputs a new encrypted index γ' and new sequence of ciphertexts **c**'.
- f := Dec(K,c): is a deterministic algorithm that takes as input a secret key K and a ciphertext c and outputs a file f.

Having already described the algorithms, the construction of them will be presented with an example.

An array is referred to as the search array and a dictionary Ts is referred to as the search table. To encrypt a collection of files f , the scheme constructs for each keyword w ∈ W a list L_w. Each list L_w is composed of $\#f_w$ nodes (N_1,.....,$N_{\#fw}$) that are stored at random locations in the search array A_s. The node N_i is defined as N_i = {id, addrs(N_{i+1})}, where id is the unique file identifier of a file that contains w and addrs(N) denotes the location of node N in As.

For each keyword w, a pointer to the head of L_w is then inserted into the search table Ts under search key $F_{K1}(w)$, where K_1 is the key to the PRF F. Each list is then encrypted using SKE under a key generated as $G_{K2}(w)$, where K_2 is the key to the PRF G. To search for a keyword w, it suffices for the client to send the values $F_{K1}(w)$ and $G_{K2}(w)$. The server can then use $F_{K1}(w)$ with T_s to recover the pointer to the head of L_w, and use $G_{K2}(w)$ to decrypt the list and recover the identifiers of the files that contain w. The total search time for the server is linear in the number of f_w, which is optimal.

In order for the scheme to become dynamic there were some limitations to overcome. The difficulty is that the addition, deletion or modification of a file requires the server to add, delete or modify nodes in the encrypted lists stored in As. This is difficult for the server to do since:

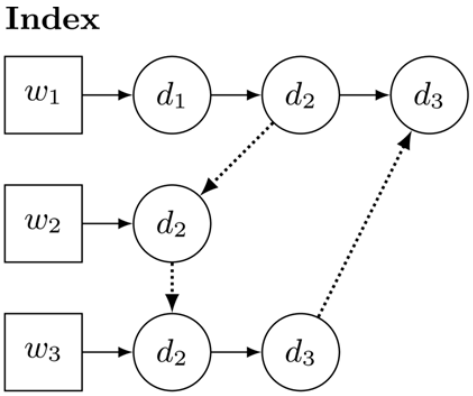
1) Upon deletion of a file f, it does not know where in As the nodes corresponding to f are stored.
2) Upon insertion or deletion of a node from a list, it cannot modify the pointer of the previous node since it is encrypted.
3) Upon addition of a node, it does not know which locations in As are free.

They addressed these limitations as follows:
1) For the file deletion issue, they added an extra encrypted data structure A_d called the deletion array, that the server will query, with a token provided by the client, in order to recover pointers to the nodes that correspond to the file being deleted. More precisely, the deletion array stores for each file f a list L_f of nodes that point to the nodes in As that should be deleted if file f is ever removed. So every node in the search array has a corresponding node in the deletion array and every node in the deletion array points to a node in the search array.

2) For the pointer modification issue, they decided to encrypt the pointers stored in a node with a homomorphic encryption scheme. By providing the server with an encryption of an appropriate value, it can then modify the pointer without ever having to decrypt the node.

3) For the memory management, in order to keep track of which locations in As are free they added a "free list" that the server uses to add new nodes.

With the Figures and the example that follows, you will have a better understanding of the scheme.



Main Index M

$f_{k_c}(w_1) \longrightarrow (4 \mid\mid 1) \oplus f_{k_b}(w_1)$

$f_{k_c}(w_2) \longrightarrow (0 \mid\mid 2) \oplus f_{k_b}(w_2)$

$f_{k_c}(w_3) \longrightarrow (5 \mid\mid 0) \oplus f_{k_b}(w_3)$

$f_{k_c}(\mathsf{free}) \longrightarrow 6 \oplus f_{k_b}(\mathsf{free})$

Deletion Index I

$f_{k_c}(d_1) \longrightarrow 1 \oplus f_{k_b}(d_1)$

$f_{k_c}(d_2) \longrightarrow 5 \oplus f_{k_b}(d_2)$

$f_{k_c}(d_3) \longrightarrow 4 \oplus f_{k_b}(d_3)$

Figure 5.1 - All indexes of the DSSE scheme

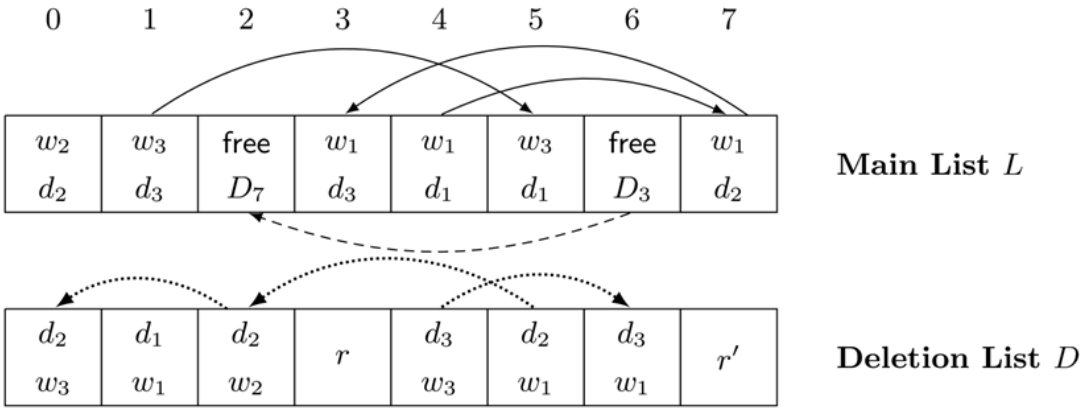


Figure 5.2 - All Lists of the DSSE scheme

The above figures shows the data structures of our fully-dynamic SSE scheme for a specific index. The index is built on three documents, namely f_1,f_2,f_3 over three keywords, namely w_1,w_2,w_3. All the documents contain keyword w_1, keyword w_2 is only contained in document f_2 and w_3 is contained in documents f_2 and f_3. The respective search table Ts, the deletion table T_d, the search array As and the deletion array A_d are also shown in that Figure. Note that in a real DSSE index, there would be padding to hide the number of file-word pairs. They omit padding for simplicity in this example. The reason that they have the search Array As, is to store in random places the tuples of every inverted index of any keyword. With this approach they hide the number of files per keyword and reduce the leakage of their scheme.

Searching is the simplest operation in their scheme. Supposing the client wishes to search for all the documents that contain keyword w_1. He prepares the search token, which among others contains $F_{K1}(w_1)$ and $G_{K2}(w_1)$. The first value $F_{K1}(w_1)$ will enable the server to locate the entry corresponding to keyword w_1 in the search table Ts. This value would be $x = (4 \| 1) \oplus G_{K2}(w_1)$. The server now uses the second value $G_{K2}(w_1)$ to compute $x \oplus G_{K2}(w_1)$. This will allow the server to locate the right entry (4 in our example) in the search array and begin "unmasking" the locations

storing pointers for the documents containing w_1. This unmasking is performed by means of the third value contained in the search token. In this example the "next" pointer of tuple number 4 in A_s, will point out that the next tuple for w1 is tuple in position 7. After the unmask of tuple 7, the "next" pointer will reveal the position 3 of the search array A_d. Finally, the "next" pointer of that tuple will inform us that this is the last tuple of the inverted index for that keyword, and the search operation will terminate.

For the adding operation, suppose that the client wishes to add a document f_4 containing keywords w_1 and w_2. Note that the search table does not change at all since f_4 is going to be the last entry in the list of keywords w_1 and w_2 and the search table only stores the first entries. However, all the other data structures must be updated in the following way. First the server uses "free" to quickly retrieve the indices of the free positions in the search array As, where the new entries are going to be stored. In our example these positions are 2 and 6. The server stores in these entries the new information (w_1,f_4) and (w_2,f_4).

Now the server needs to connect this new entries to the respective keywords lists by using the add token. It retrieves the indices i=0 and j=3 in the search array As of the elements x and y such that x and y correspond to the last entries of the keyword lists w_1 and w_2. In this way the server homomorphically sets As[0]'s and As[3]'s "next" pointers to point to the newly added nodes, already stored in the search array at positions 2 and 6. Because keyword w1 is placed at position 2, the "next" pointer of the tuple at position 3 ($A_s[3]$) will change in order to point to position 2. Respectively the pointer of As[0], will now point to position 6.

Note that getting access to the free entries in the search array also provides access to the respective free positions of the deletion array Ad. In our example, the indices of the free positions in the deletion array are 3 and 7. The server will store the new entries (f_4,w_1) and (f_4,w_2) at these positions in the deletion array and will also connect them with pointers. Finally, the server will update the deletion table by setting the entry $F_{K1}(f_4)$ to point to position 3 in the deletion array, so that file f4 could be easily retrieved for deletion later.

For the Deleting operation let suppose that the client wants to delete document f_3, which contains the keywords w_1 and w_3. The deletion is a "dual operation" to addition. First the server uses the value $F_{K1}(f_3)$ of the deletion token to locate the right value $4 \oplus G_{K2}(f_3)$ in the deletion table. This will allow the server to get access to the portion of the remaining data structures that need to be updated. Namely it will free the positions 4 and 6 in the deletion array and positions 1 and 3 in the search array. While "freeing" the positions in the search array, it will also homomorphically update the pointers of previous entries in the keyword list w_1 and w_3 to point to the new entries. Note that no such an update of pointers is required for the deletion array, because when deleting a file, all the nodes related to that file are deleted from the T_d.

5.3 Leakage

A limitation of all known SSE constructions is that the tokens they generate are deterministic, in the sense that the same token will always be generated for the same keyword. This means that searches leak statistical information about the user's search pattern. Currently, it is not known how to design efficient SSE schemes with probabilistic trapdoors.

The scheme that was just described has some leakage that must be mentioned for every phase separately. During Setup phase, that scheme reveals some information, which is the setup leakage. More specific those are the number of elements that the search array A_s contains and the total number of keywords and the number of files that have been added at setup phase.

Search takes as input a keyword w and returns a set of file identifiers ($label_f$ / id(f)) to the client. So, during the search operation it is leaked the pairs of (w,f) that fulfill the query. Moreover server can learn through search leakage weather that keyword was searched in the past again or not. How many times it has been searched and when.

Add takes as input an add token that contains the identifier for a file f and adds word information for a set of words associated with this file. Like Search, Add writes tuples (w,id(f)) for each word w associated with the file in the Add Token. Server can learn the number of keywords that this document contains. Additionally, however, the Add operation reveals to the server whether or not f is the only file that contains w. In some cases it might also learn the file f' which was the head of the list before the add operation.

Delete takes as input a delete token that contains an identifier for a file f. This token does not contain any word-specific information. However, in the process of executing the Delete operation, the server uncovers in the index a word-identifier (the search key for Ts) for each word associated with the file. So, like Search and Add, Delete learns tuples (w, id(f)) for each word w associated with f. As each word w is deleted for f, it reveals the location of its neighbors in the search array. So, the leakage in this case consists of the file identifiers for the previous and next nodes in the list for every w that exist in the deleted file.

Chapter 6

Second analysis of a DSSE proposal "Dynamic Searchable Encryption via Blind Storage"

6.1 Introduction

The second scheme presented in this chapter is considered to be simpler than the others proposals. Particularly, it does not require the server to support any operation other than upload and download of data. Thus the server in that scheme can be based solely on a cloud storage service, rather than a cloud computation service as well. So, the important feature of that pseudorandom set construction, compared to linked-list based constructions, of related work in the literature, is that the server need not carry out any decryptions. In linked-list based constructions, each node in the list is progressively revealed. In contrast, that construction allows the server to be "crypto-free" and still have only constant number of rounds of interaction. Indeed, the only operations that need to be supported by the server are uploading and downloading blocks of data, if possible, parallelly.

Most of the previous SSE schemes that have been presented are using a dedicated server, that performed both storage and computation, like the figure bellow.

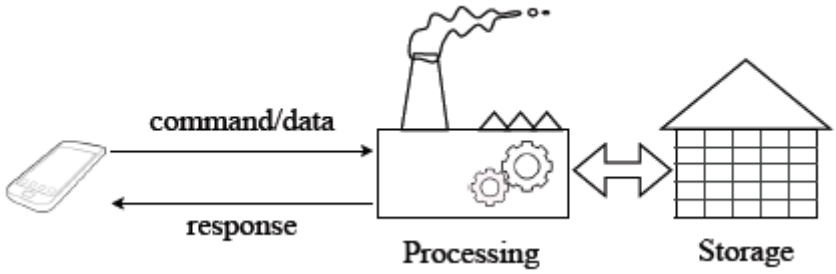


Figure 6.1 - Topology of most DSSE schemes

In such cases, the computation typically involved a sequence of decryptions. To deploy such a scheme, one would need to rely not only on cloud storage services, but also cloud computation services. This presents several limitations. Firstly, this limits the choice of service providers available to a user. Someone could use Amazon EC2 for computation and combine it with Amazon S3 for storage. However, it is not viable to use Dropbox for persistent storage and Amazon EC2 for computation, as this would incur high costs for communication between these two services.

In contrast to, this proposal of Blind Storage, can be easily implemented using Dropbox or other similar services which provide only storage, as can be seen from the Figure 6.2.

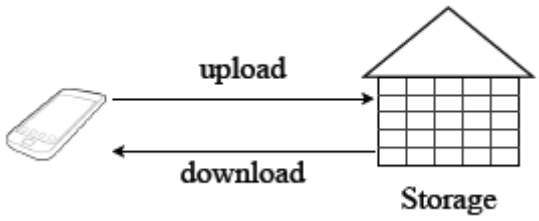


Figure 6.2 - Topology of Blind Storage scheme

The Blind Storage scheme allows a client to store a set of files on a remote server in such a way that the server does not learn how many files are stored, or the lengths of the individual files. Only when a file is retrieved, the server will learn about its existence. A drawback again is that the Storage can notice if the same file being downloaded subsequently, but again the file's name and its content are not revealed.

The Blind Storage scheme also supports adding new files and updating or deleting existing files. Further, though not needed for the Dynamic SSE construction, the Blind Storage scheme can be used so that the actual operation is hidden from the server. As a result the storage cannot understand if the operation is add, delete, write or read.

The main construction is that of a versatile tool called Blind Storage, which is then used to build a SSE scheme. In building the SSE scheme, the search index entries for all the keywords are stored as individual files in the Blind Storage scheme.

The Blind Storage scheme, called SCATTERSTORE, is constructed using a simple, yet powerful technique. Each file is stored as a collection of blocks that are kept in pseudorandom locations. At Figure 6.3 you can see how a file is transformed to a collection of blocks. The server sees only a super-set of the locations where the file's blocks are kept, and not the exact set of locations. The key security property, from the point of view of the server, is that each file is associated with a set of locations independent of the other files in the system. On the other hand, the sets of locations for two files can overlap.

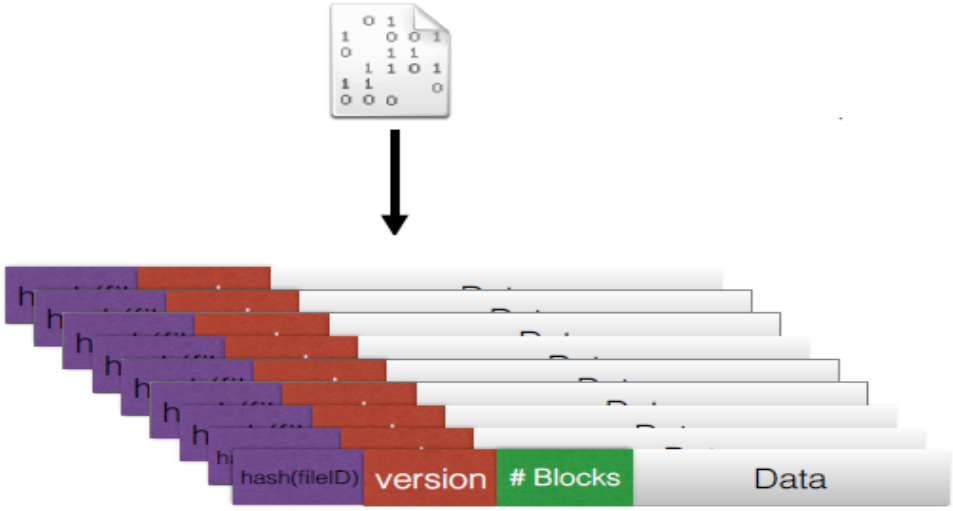


Figure 6.3 - File transforming to a collection of blocks

A probabilistic analysis shows that for appropriate choice of parameters, the probability that any information about files not yet accessed is leaked to the server can be made negligible.

The only cryptographic tools used in our scheme are block ciphers for standard symmetric key encryption, as well as for generating pseudorandom locations where the data blocks are kept and collision resistant hash functions.

Also our SSE scheme could, in principle, involve up to three rounds of communication for retrieving the documents (this happens if the keyword has a large number of matching documents). In contrast, many existing schemes involve only two rounds (one to retrieve encrypted list of documents, and one to retrieve the documents themselves).

6.2 Algorithm

The blind storage system consists of a client and a "dumb" storage server. The server is expected to provide only two operations, download and upload. The data are represented as an array of blocks. The download operation is allowed to specify a list of indices of blocks, to be downloaded. Similarly, the upload operation is allowed to specify a list of data blocks and indices for those blocks.

Simpler scheme

A blind storage system is defined by three polynomial-time algorithms on the client-side: BSTORE-Keygen, BSTORE-Build and BSTORE-Access. Of these, only the BSTORE-Access is an interactive protocol.

BSTORE-Keygen takes security parameter as an input and outputs a key K_{BSTORE}. The K_{BSTORE}, which the client is required to retain throughout the lifetime of the system, is required to be independent of the data to be stored.

BSTORE-Build takes as input (K_{BSTORE}, d_0, $\{id_i, data_i\}^t_{i=1}$, where K_{BSTORE} is a key, d_0 is an upper bound on the total number of data blocks to be stored in the system and (id_i, $data_i$) are the id and data of the files, that the system will be initialized with. Finally, it outputs an array of blocks D to be uploaded to the server.

BSTORE-Access takes as input a key K_{BSTORE}, a file id "id", an operation specifier op ϵ {read, write, update, delete}, and optionally data "data". Then it interacts with the server and returns a status message and optionally file data. For the update operation, BSTORE-Access allows more flexibility. First it requires only id as input, and outputs the current size of the file with that ID. Then it accepts as input (an upper bound on) what the size of the file will be after update. Then it outputs the current file data. Only then, it will require the new data with which the file will be updated.

To store a file f of n blocks, a pseudorandom subset S_f of not n blocks is required to be chosen. For collision and security issues usually 2n blocks are required. This subset of 2n blocks will be chosen independent of the other files in the system. This subset is what the server sees when the client accesses this file. Within this set a subset $\widehat{S}_f \subseteq S_f$ of n blocks is chosen, where the actual data are stored. The set $\widehat{S}_f$ is of course, selected depending on the other blocks used by other files, to avoid collisions. However, since the contents of the blocks are kept encrypted, the server does not learn anything about $\widehat{S}_f$, except of its size.

Full Construction

In the simpler scheme described above, the client maintained a data-structure mapping a file-identifier id_f to a descriptor of the pseudorandom set S_f . This is not desirable if the system would store a large number of small files. In such cases the size of this data structure is comparable to that of the entire collection of files. The client should store only a constant number of cryptographic keys.

In order to define a pseudorandom set S_f , two pieces of information are needed, a seed and the size of the set. The seed itself can be obtained by applying a full-domain PRF to the file-identifier. So, if the client knew the size of S_f as well, there will be no need to store this map at all. This can be achieved by using a two-level access to a file, as follows.

For each file, the first block consists of a header that stores the size of the file. The size of the file is the number of blocks that are needed to store the file. To retrieve a file with id_f , the client assumes that the file is "small" and retrieves a pseudorandom set S^0_f with the smallest possible number of blocks, i.e. κ. After recovering the first block of the file from the blocks in S^0_f, the client computes the actual size of S_f. In case, it is larger than κ, then client retrieves the rest of S_f from the server.

As you can see from the figure 6.4, every block has a header and a Data-field. Header field is required in order for the client to understand if that blocks refers to the file that he is searching for. In cases of updates, the header also store the version. An exception to that format is only the first block of a file, which has a bigger header. It also stores the number of blocks that this file has.

Figure 6.4 - Format of blocks that consist a file

Figure 6.5 - Format of the first block of every file

Dynamic SSE

Finally, this pseudorandom set construction easily supports a dynamic blind-storage scheme. The update operation such as creating or deleting a file are considered as special cases of the update operation. To update a file, the client retrieves the encrypted blocks corresponding to the file's pseudorandom set S_f, and decrypts them. Then he updates the subset of blocks $\widehat{S}_f$ where the file's blocks are present and re-encrypts all of the downloaded blocks. Finally uploads them back to the server.

A dynamic searchable symmetric encryption scheme consists of five probabilistic polynomial time procedures which are run by the client. SSE-keygen, SSE-indexgen, SSE-search, SSE-add and SSE-remove. These procedures interact with, a "dumb" server which provides download and upload facilities to access blocks in an array and also with a simple file-system in order to lookup documents by identifiers.

SSE-keygen: Takes the security parameter as input, and outputs a key K_{SSE}. All of the following procedures take K_{SSE} as an input.

SSE-indexgen: Takes as input the collection of all the documents, a dictionary of all the keywords and for each keyword, an index file which lists the document IDs in which that keyword is present. It interacts with the server to create a representation of this data on the server side.

SSE-search: Takes as input a keyword w, interacts with the server, and returns all the documents containing w.

SSE-add: Takes as input a new document, interacts with the server, and incorporates it into the document collection.

SSE-remove: Takes as input a document ID, interacts with the server, and if a document with that ID is present in the server, removes it from the document collection.

In order for the dynamic scheme to be functional, for each keyword, they use two index files. One is listing the original documents that include that keyword, and another listing the newly added documents. The first index file is stored with the server using a blind storage scheme, where as the second can be stored in a "clear storage" system.

Searching for keywords now involves retrieving both these index files. Adding documents involves updating only the second kind of index files. Also, removing a newly added document involves updating only the second kind of index files, which is straightforward. But, while removing an original document, it should be ensured that, the information for keywords in it, which have not been searched, remains secret. This is achieved by a lazy deletion strategy. The index file of a keyword, from the original set of documents is not updated until that keyword is searched for. At that point, if the client learns that a document listed in that index has been deleted, the index is updated accordingly.

6.3 Leakage

The information revealed to the server is strictly lesser than in all prior Dynamic SSE schemes. Moreover it satisfies a fully adaptive security definition, allowing for the possibility that the search queries can be adversarial influenced, based on the information revealed to the server by prior searches.

Security is in the standard model, rather than the heuristic Random Oracle Model. This is achieved because it relies only on the security of block ciphers and collision resistant hash functions. The number of documents in the system and their lengths can be kept secret, revealing the existence of a document only when it is accessed by the client.

So, the Blind Storage scheme lets the client keep all information, including the number and size, about files secret from the server who stores them, until they are accessed.

Chapter 7

Third analysis of a DSSE proposal "An ORAM based forward privacy preserving Dynamic Symmetric Searchable Encryption Scheme"

7.1 Introduction

Until now, it is obvious that ORAM schemes provide security, but they are not efficient. This was the reason that SSE schemes seemed to be the reasonable choice for searchable encryption. The proposal presented in this chapter is a SSE protocol, proposed by Rizomiliotis, that makes a limited use of ORAM algorithms in order to achieve forward privacy and to minimize the overhead that ORAMs introduce. "An ORAM based forward privacy preserving Dynamic Symmetric Searchable Encryption Scheme" was one of the first proposals that achieved that level of security and also improved the search and update complexity.

In this survey, it is introduced an efficient DSSE scheme that achieves forward privacy with very small leakage. The scheme uses double linked lists to store the set of file identifiers per keyword and an ORAM structure in order to manage keyword related information. Because using just ORAM is not efficient, this proposal shows how to make a limited use of an ORAM algorithm in order to achieve forward privacy security and at the same time, to minimize the overhead that an ORAM construction introduces. More precisely, the scheme has the following characteristics:

- As far as leakage concerns, it leaks the access and the search patterns, as every SSE scheme do.
- Furthermore there is no backward privacy, because it leaks the identifiers of the deleted pairs.
- However, it is only the second DSSE that offers forward privacy.
- Efficiency. The search complexity is $O(\max(\log^2(|W|)/c, |I_w|))$, where $|W|$ is the number of keywords, $|I_w|$ the number of files that contain the keyword w and c depends on the ORAM scheme that is used.

The definition of the SSE is similar to previous works. The only difference is that they define update at the level of a single (w, id) pair. The majority of all other proposals consider additions/deletions of files, which can be seen as, a lot of pairs together.

That scheme, uses several data structures. More precisely, there are two double indexed lists per keyword, a hash table T which stores simultaneously these lists and an ORAM structure which stores all the keyword related information. They

also use an IND- CPA secure secret-key encryption scheme SKE, a pseudorandom function $H : \{0,1\}^{\lambda} \times \{0,1\}^{*} \leftarrow \{0,1\}^{\lambda}$ and a random oracle $G : \{0,1\}^{\lambda} \times \{0,1\}^{*} \leftarrow \{0,1\}^{\lambda}$, where λ is the security parameter. In order to encrypt the files **F**, a secret-key encryption scheme is used, which is a tuple of three algorithms SKE = (Gen, Enc, Dec). The ciphertexts are outsourced and managed using the corresponding file identifier ($label_f$).

As mentioned before, it is based on the inverted index approach, for the keywords. For each keyword w ϵ W, they maintain two double indexed lists of file identifiers, $L_w^{(o)}$ and $L_w^{(n)}$. The list $L_w^{(n)}$ contains the subset of the file identifiers I_w that have been added after the last search for the keyword w. Moreover at this list, they store all the pairs at Setup procedure. The list $L_w^{(o)}$ contains all the remaining file identifiers, which are the file identifiers that have been leaked because of a search query for w and that have not been deleted.

As an overview, when a pair has been searched, and so has been leaked to server, it is treated differently than other pairs who have not been searched yet. When a pair is leaked, then efficiency must be the first priority for these pairs and not security.

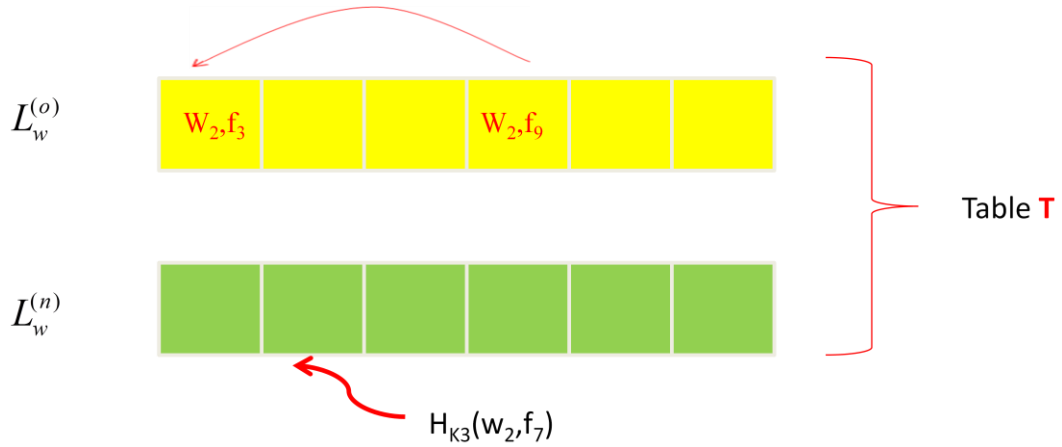


Figure 7.1 - Example of Table T structure

When the set I_w is updated, one of the two lists is updated, depending on the file identifier's history. This means that, when a previously leaked identifier is deleted from I_w, then the corresponding node is removed from the list $L_w^{(o)}$, otherwise it is

deleted from the list $L_w^{(n)}$. Similarly, additions are performed into the corresponding list. When a pair is added to hash table T, first it is checked whether this tuple has been removed in the past and in such a case whether that tuple has been searched in the past or not. So, if the new pair was never searched in the past, they will add it at $L_w^{(n)}$. Otherwise, if that pair has been searched and then deleted, now that it will be added again, it will be placed at $L_w^{(o)}$.

Since double linked lists are used, each node of the list contains three fields: a link to the previous node, a link to the next node and the data field. In our case, the pair (w, id), which is the keyword-file identifier pair, consists the node's data field:

$$[\text{link}_{\text{prev}} \mid (w, id) \mid \text{link}_{\text{next}}]$$

When $\text{link}_{\text{next}} = '\perp'$ the node is the last of the list (tail), while when $\text{link}_{\text{prev}} = '\perp'$, it is the first (head).

Four types of keys are used in the scheme, those are the following k_1, k_2, k_3 and k_w. k_1, k_2, k_3 are stored at the client, and they remain the same for the lifetime of the scheme. k_1, is the ORAM key and it is used by the client in order to communicate with ORAM with encryption. k_2 is used to encrypt the outsourced files F with the IND-CPA encryption algorithm. k_3 is used to produce the search keys of the hash table T by keying the pseudorandom function H. Finally, each keyword $w \in W$ has its own secret key k_w. When a keyword w is searched by the client then the key k_w for that keyword is changed. A new one K_w' is created by the client and stored back at ORAM.

7.2 Algorithm

The list nodes are stored in a hash table T. The pair (w, id) is inserted into the table under the search key H_{k3} (w, id), where k_3 is the λ-bit secret key to the pseudo-random function H. In other words, the link to a node, is a location in T and equals the output of H_{k3}(w, id) for a pair (w, id). The list's $L_w^{(o)}$ nodes are stored unencrypted, while the $L_w^{(n)}$ list nodes are stored partly encrypted. Only the data field is kept encrypted, by masking with G_{kw}(r), where r is a randomly sampled string. Thus, each node of the $L_w^{(n)}$ list has the form:

$$[link_{prev} \,|\, (< w, id > \oplus G_{k_w}(r), r) \,|\, link_{next}]$$

where the secret key k_w is different per keyword. In the following figure it is explained the format of both lists. The yellow tuples are the pointers. The blue one are the leaked and unencrypted datafield of list $L_w^{(o)}$. The red tuples are the encrypted datafield of the list $L_w^{(n)}$.

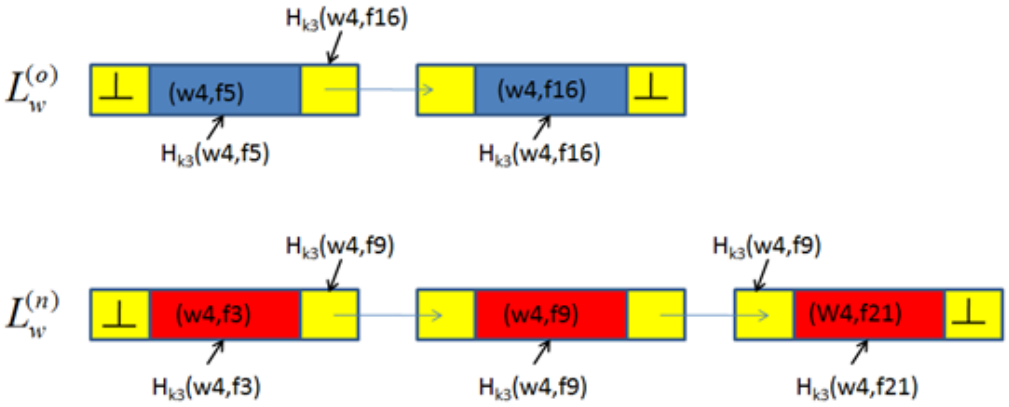


Figure 7.2 - Example of Double Linked Lists for keyword w_4

The position of every tuple in T is found based on the hash function: $H_{k3}(w_i,id_j)$. So the yellow parts of every node, which are the pointers to the previous and next node contains the result of these hash functions.

The ORAM structure is maintained in order to provide forward privacy, by storing information for every keyword. Per keyword, the ORAM stores the followings:

- The head of the list $L_w^{(n)}$
- The head of the list $L_w^{(o)}$
- The keyword's secret key k_w.

The ORAM is searched using a keyword w and the ORAM's secret key k_1. The structure has the format that you can see below:

$$DataO_w = [k_w \mid head_w^{(o)} \mid head_w^{(n)}]$$

The ORAM supports two operations, ReadOram and WriteOram. Using the secret key k_1, the first operation is used for reading the keywords data $DataO_w$ and the second for updating this data:

- (O', $DataO_w$) ← ReadOram(O, k_1, w): Using the key k_1 the ORAM O is queried using the keyword w as the search key. The new ORAM state O' and the keyword's data $DataO_w$ are returned.

- O' ←WriteOram(O, k_1, w, $DataO_w$): Using the key k_1, the ORAM is queried using the keyword w as the search key. The keyword's data are replace by $DataO_w$ and the new ORAM state O' is returned.

As far as double List operations are concerned, they are defined four operations for the management of those lists:

- ReadList (T, head, k_w): The list is read from the hash table T using the "head" as the location of the first node. It returns a set I_w of the file identifiers stored at the data filed of the list's nodes. When the node's data field is encrypted, k_w which was retrieved from the ORAM is used as the key for decryption of the datafield. After the decryption, the nodes are stored back into the hash table T unencrypted, because now those entries belongs to $L_w^{(o)}$ list.

- AppendList (T, $head_1$, $head_2$): The list $L_w^{(n)}$ with the first node stored at the location $head_2$ of the hash table T is appended at the end of the list $L_w^{(o)}$ with first node location $head_1$. This procedure is followed after every search for a keyword w.

- AddNode (T, head, pos, nd): The node nd is added as the head of a list. The node is stored at the pos-th location of the hash table T. The old first node of the list is stored at the location head. Because when a new pair is added at an inverted index, it is always added at the head of the list, it should be modified So, the linkprev of the old head is modified to point to the location "pos", which is the new head position in T.

- DelNode (T, pos): Deletes the node that is stored at the pos-th location of the hash table T. In order this to be done, the server needs to connect the previous with the next node. In order to learn if the deleted node was the head of the list and to store the new head's location, the fields linkprev and linknext of the deleted node are returned as output.

Dynamic

As an overview, the operations of the DSSE scheme will be described more precisely.

Setup operation. The client chooses three random λ-bit strings k_1 for the ORAM structure, k_2 for the files encryption and k_3 for keying the pseudorandom function H that connects the pairs with the hash table entries. Moreover, it initiates the ORAM structure for O(|W|) elements and the hash table for O(N) pairs. All nodes during this phase are added at $L_w^{(n)}$, because they are not leaked yet.

For the Search operation, initially the client needs to retrieve from ORAM the information that he needs. By using the secret key k_1 and the keyword w he reads from the ORAM O all $data_w$ related to the specific keyword. $Data_w$ contains the location of the head of the list $L_w^{(o)}$, the location of the head of the list $L_w^{(n)}$ and the current secret key k_w of the keyword w. Because after every search the list $L_w^{(n)}$ is

appended at the end of $L_w^{(o)}$, the location of the head is set to '$\perp$' and the keywords secret key is reset to a new value k_w'. After that phase, client send to the server a search token that consists of the information he retrieved from ORAM. The server reads the lists $L_w^{(o)}$ and then decrypts, by using k_w and read the list $L_w^{(n)}$. From both lists, server compiles a set I_w which contains all the file identifiers that fulfill the query for keyword w. I_w is sent back to the client as the answer. Then, the list $L_w^{(n)}$ is appended at the end of $L_w^{(o)}$.

For the UpdatePair operation. The client computes and sends to the server the position of the pair (w, id) at the hash table T. The position is calculated by H_{k3}(w,id). The server checks this position. If it has never been used or if it was used but it was never decrypted, then the list that must be modified is $L_w^{(n)}$. In all other cases, the list $L_w^{(o)}$ is modified. In the case of a delete operation, then the server uses the DelNode operation to delete the node and the sends to the client the values linkprev and linknext of the deleted node. If linkprev equals '$\perp$', that means that the deleted node was the head of the list, and the client updates the keyword's w ORAM information with the location of the new list' head. It should be also mentioned here that the content of T is not erased when the operation is "delete". For the AddNode operation, the client retrieves from the ORAM the list's head as well as the keyword's secret key k_w. The client prepares the new node and encrypts only the data field with k_w, in case that the new node will be placed at $L_w^{(n)}$. Then it uses the AddNode to write the node into the hash table and updates the ORAM information, such as the new list's head position.

7.3 Leakage

In terms of security, all SSE schemes leak some information. Although Oblivious RAM solutions are costly, they can be used in order to minimize this leakage. During the last years, ORAM proposals and implementations are getting more practical, but there is still a performance gap to be filled. On the other hand, in order to make SSE schemes practical, some more leakage should be expected. SSE leakage contains at minimum the search pattern which refers to the hashes of the keywords that are searched and the access pattern, which is the matching document identifiers of a keyword search.

Beside the above two patterns that cannot be hidden at SSE schemes, some extra information is revealed at every phase of the scheme. The Setup operation leaks the number of keywords |W| and the number N of different pairs of keyword-file identity pairs. A search operation for a keyword w leaks the set of file identifiers matching the keyword, that have been added or even deleted in the past. The deleted pairs are leaked and so the scheme does not have backward privacy. Furthermore, the server could learn how many times and the exact time that the same keyword was accessed in the past.

The UpdatePair operation leaks the file identifier and the type of the update operation, such as add or delete. If the pair was searched in the past, then the keyword w is leaked. If the pair has never been searched, then the keyword is not leaked.

During the add operation, an extra leakage exist that was not mentioned in the leakage analysis. Because the pointers are unencrypted when a new tuple is becoming the head of the list, the server must learn which was the location of the old head of the list. As a result, the server could also learn the length of the list, simply by following the "linknext" until he found '$\perp$'. In other words the extra leakage is that the server could learn how many files have also the same keyword, as the keyword which is added. A proposal in order to avoid that leakage is analyzed in the following chapter.

7.4 Proposal to minimize leakage

When a new file is added then the reverted lists for all the keywords that are contained in that file should be updated. The old head of every list is retrieved from the ORAM structure. Then the ORAM information is updated, in order to point out at the new head. Finally the linknext of the new head and the linkprev of the old head are updated.

The issue is that the links are unencrypted. So, the Server can use those links and can find out how many nodes (files) already contain the keyword that correspond to that list. This is an extra leakage that has not be taken under consideration and should be eliminated.

In order to eliminate a second key is needed to be stored at ORAM for every keyword. K'_w will be used in order to encrypt the linknext pointers of the newly added pairs. Like K_w, K'_w will also change after every search operation. By encrypting the pointer to the next node, the Server will not be able to follow the pointers and learn the number of nodes of that list.

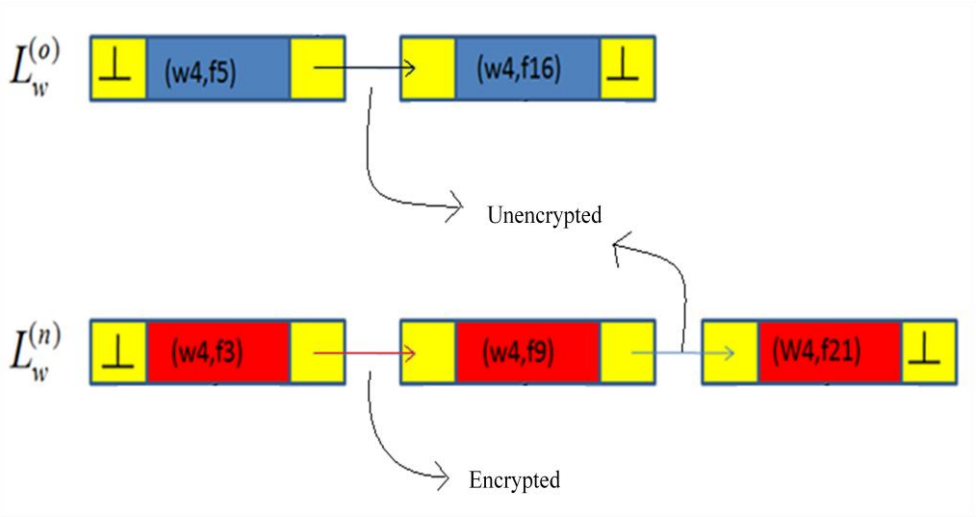


Figure 7.3 - Example of encrypted pointers at $L_w^{(n)}$

During deletion operation the Server is not obligatory to learn the key K'_w because he just have to move the linknext pointer of the deleted node and use it as the linknext pointer of the previous node in the list. The same will be done for the linkprev pointer.

Even if two pairs/nodes are added with encrypted linknext pointer, the Server will have no issue because both of them will be encrypted with the same key K'_w. This encryption is only valid for the additions at list $L_w^{(n)}$. The list $L_w^{(o)}$ is totally unencrypted and all the information are leaked.

Chapter 8

Forth analysis of a DSSE proposal "ORAM based forward privacy preserving Dynamic Symmetric Searchable Encryption Scheme"

8.1 Introduction

At this survey, the author presents two efficient DSSE schemes that leak a limited amount of information. Both of them make a limited use of ORAM algorithms in order to achieve forward privacy. It is one of the first proposals that could achieve forward privacy while still be efficient and simple.

The contribution of this proposal, is not only the forward privacy or the small leakage that it has. An advantage that is has is that it relies only on hash functions for the search operation, which makes it very fast comparing to other DSSE schemes that also need a lot of decryption operations during the search phase. Moreover, it make use of simple structures such as dictionaries to store the inverted index for keyword and the index for file. The locality that some of this structures have helps a lot to make the scheme even more efficient.

The main difference of the two proposals (simple and the extended one) of that research, is the leakage at setup phase. Both of them leak the access and the search pattern after each search operation for a keyword. Furthermore during add or delete phase, both schemes offer forward privacy and the only information leaked is the number of keywords per added file. Each file that it is added, even if it was previously deleted from the DSSE, it is treated as a completely new file with new file identifier. At setup phase, if padding is not used, then it is leaked the Finally, number of keywords and files. However the difference is that the simple DSSE leaks also the number of keywords per file. The 'extended' DSSE hides this information, by using two more structures.

As far as efficiency is concerned, the search complexity of both schemes is $O(\max(\log2(|W|)/\gamma, |I_w|))$, where $|W|$ is the number of keywords, $|I_w|$ is the number of files that contain the keyword w and γ depends on the ORAM scheme that is used. Furthermore, even at the simple scheme the Search, Add and Delete operations can be performed with parallelism.

As a general idea of this proposal, the information is divided into two categories, leaked and not leaked. The leaked data can be treated differently by mean of security, in order to have a gain for locality. So the leaked data are stored in contiguous area of memory positions and dictionary reads are replaced by array reads.

A drawback of most previous works on DSSE was that they did not support the storage of the index for file at the DSSE. So, it was not obvious how the deletions of a file was achieved, while not knowing the exact keywords that this file contained. That scheme took that under consideration and so it also maintains the index for file.

Simple Scheme

In order to support search queries, the inverted index for keyword must be stored in a structure. The scheme is based on the following observation: at any time t the set I_w of the file identifiers related to a keyword w can be divided into two subsets $I_w^{(1)}$ and $I_w^{(0)}$. The subset $I_w^{(0)}$ contains the file identifiers ($label_f$) that have been leaked because they were used in a previous search. They can be stored efficiently, without any privacy concern, by using locality. The structure for those file identifiers is S0. The other subset $I_w^{(1)}$ contains the file identifiers that have not been used in a search reply. The data structure for those file identifiers is called S1 and is formatted in a way that protects their privacy. Note that this set includes also the file identifiers that were leaked, deleted and, then, added again.

The inverted index of any keyword is stored to S0 and S1. In many cases there would be some entries of the inverted index for a keyword to S0 and some unleaked entries to S1. Of course, after each search operation the corresponding S1 entries are appended to structure S0, because at the end of the search operation, they would be leaked .

On the other hand addition or deletion of files requires the maintenance of the file index. A third data structure S2 is used to store the sequence of keywords W_i per file $f_i \in F$. The simple scheme has five structures. S0, S1 and S2 are using dictionaries. The other two are ORAM structures where it is stored the client's state. $ORAM_w$ is used to store keyword related information while $ORAM_f$ is used for the client's file state and is necessary for achieving forward privacy. The Dataowner can read and write data from the ORAM structures using the secret keys K_W and K_F.

Besides those structure the scheme also use some more cryptographic tools such as an IND-CPA secure secret-key encryption scheme for the encryption of files F and a random oracle H.

As it was mentioned before each file has a unique file identifier ($label_f$), which is randomly selected at the addition of the file. Each keyword has a unique identifier w and a temporary secret key K_w. Every time the keyword w is searched, a new key value K'_w is selected and stored at $ORAM_w$ for that keyword. All labels P_w used in S1 are computed by applying the random oracle to the secret key K_w and a counter c.

The basic idea of the proposal is that for each keyword/file identifier pair (w, $lable_f$) there are two dictionary entries. One in S2 for the file index and one either in S0 or in S1, depending whether that pair has been leaked or not.

In case the pair is unleaked , then there is an entry in S1 :

$$S_1(P_w) = label_f \,|\, c$$

with label P_w and an entry in S2 :

$$S_2(label_f) = (P_w \,|\, '1')$$

with label the $label_f$. The '1' indicates that the pair has not been read yet. Moreover, P_w is calculated by :

$$P_w = H(K_w \,|\, c)$$

Also, c is the value of the counter used to compute P_w and at the same time indicates that it is the c-th entry in S2 with label the $label_f$.

On the other hand, if the pair has been leaked, then in S0 there is an entry with label the keyword w :

$$S_0(w) = (label_f \,|\, c)$$

and is S2 :

$$S_2(label_f) = (w \,|\, i \,|\, '0')$$

where '0' indicates that the pair has been leaked and it is stored in S0 with label w. Value i , means that it is the i-th item with this label at structure S0. It is used for sorting purposes in order for S0 to have locality.

After having explain the format of the dictionaries, it should also be mentioned that the $ORAM_w$ structure's data is a pair (K_w, c) per keyword w. The K_w is the current secret key used for labeling the entries in S1 related to w. The counter c indicates how many such entries have been added since the last search for the keyword w. On the other hand, $ORAM_f$ entries store only the current identifier $label_f$ of a file f. $Label_f$ is a random choice and it cannot be retrieved with any other way than storing it.

The following figure will help to better understand the structures that are used in this scheme.

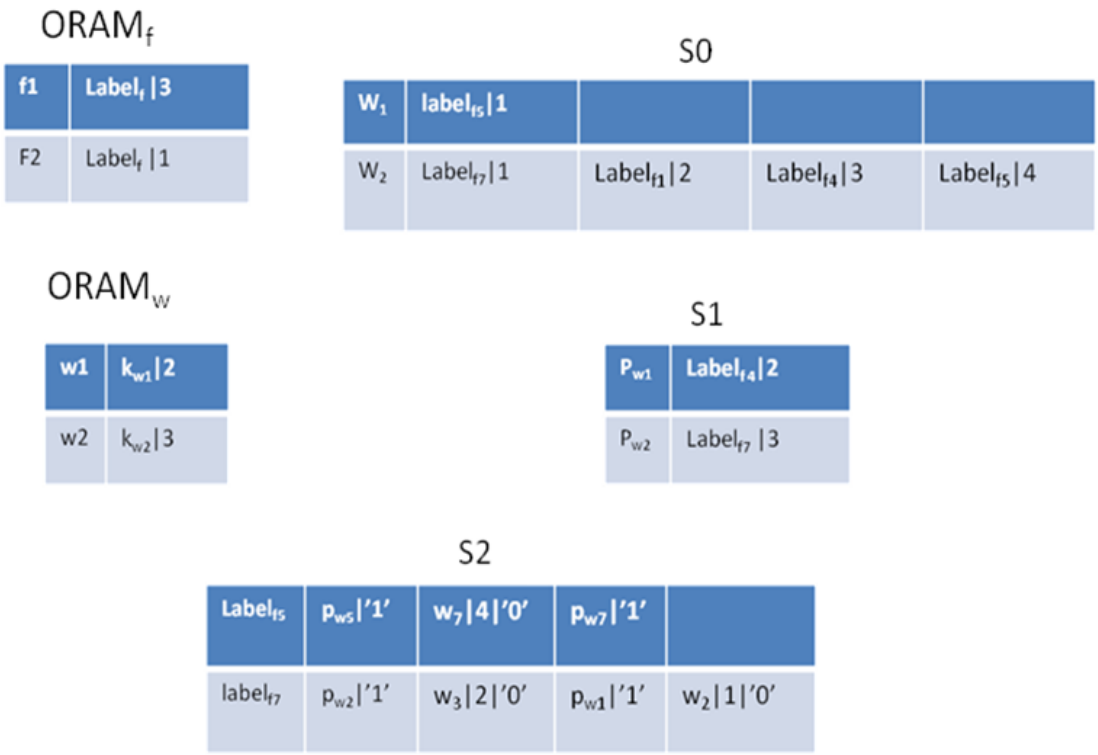


Figure 8.1 - All structures of the simple DSSE scheme

$ORAM_f$

It Stores the random labelf of every file and the number of keywords that the file contains.

$ORAM_w$

It stores the current key of every keyword and the number of unleaked files that contain this keyword.

S1

The first value is the $label_f$ of the file of the corresponding pair. The second value points out the column of the S2 structure, which will be leaked, due to the search that returned that $label_f$.

S2

The final value is "1" if the pair is unleaked or "0" if it is leaked. In case the pair is unleaked then the first value is the label at structure S1 for the corresponding pair. Otherwise the first value is the keyword of the pair and the middle value is the column of S0 structure that should be deleted if that pair should be removed.

S0

The first value is the $label_f$ of the file of that pair. The second value is used in order to sort again the row in case of deletion of a cell.

8.2 Algorithm

The scheme is a Dynamic SSE, so it must support four operations (Setup / Search / Add / Delete), which will be described in this chapter and also present the algorithms for them.

Setup operation.

The Dataowner (D) chooses three random λ-bit strings K_F, K_W for the two ORAM structures and K for the files encryption. Then, initiates the two ORAM structures $ORAM_w$ and $ORAM_f$ which are of size |W| and |F| respectively. For each keyword $w \in W$ it is generated a random key K_w and it is stored at $ORAM_w$ with label the specific keyword w.

The next step would be to fill in S1 the inverted index for keyword and in S2 the file index. For this step D creates the $label_f$ for each file and calculate the values of the P_w, which will be used as labels at structure S1.Because at the setup phase no search will have been done yet, the structure S0 is empty, as there is no leaked information. As a final step he encrypts every file with K and outsource it at Cloud Storage. Moreover he outsource S1 and S2 at Cloud Server

1. $K \xleftarrow{\$} SKE.Gen(1^{\lambda})$, $K_F \xleftarrow{\$} \{0,1\}^{\lambda}$, $K_W \xleftarrow{\$} \{0,1\}^{\lambda}$

2. Initiate two ORAMs of size |W| and |F|

3. For each $w \in W$

$K_w \xleftarrow{\$} \{0,1\}^{\lambda}$

4. For each $f \in F$

f'=SKE.Enc(K,f)

$label_f \xleftarrow{\$} \{0,1\}^{\lambda}$

add ($label_f$,f') to list C

$ORAM_f$.write(f,$label_f$)

i=0

For each $w \in W$

i++

P=H(K_w|i)

S1.insert(P,($label_f$|i)))

S2.insert($label_f$,(P|'1')))

c(w)=i

For each $w \in W$

c=c(w)

$ORAM_w$.write(w,(K_w,c))

5. Output C,S1,S2

Search operation

The search for a keyword w returns the set of file identifiers I_w that contains that keyword. The search should check both S0 and S1 for that keyword. The first step is very efficient as it has to do with the retrieval of the file identifiers that have been used in a previous search. These identifiers are stored with the same label w in S0. This step is parallizable with high locality.

In order to retrieve the file identifiers from S1, the server should create and then search for all the labels P, that correspond to the keyword w. In order this to be done the client retrieves from ORAM the information about that keyword, K_w and c, and sends them at the server, who will create the labels P and search them at S1. The retrieved identifiers from S1 are then stored in S0 using the label w and the corresponding entries in S2 are updated. At S2, the entry (P|'1') will change to (w|j|'0') and now the entry at S2 will be leaked and reviled. Finally, the keyword's secret key K_w is updated with a new randomly selected value and the counter c is set to zero.

The client side

1. $(K_w,c) \leftarrow ORAM_w read(w)$
2. $K'_w \xleftarrow{\$} \{0,1\}^{\lambda}$, c'=0
3. $ORAM_w$.write(w,(K'$_w$,c'))
4. Output w, Kw, c

The server side

1. $I_w \leftarrow$ S0.get(w)
2. For i=1:c do
   - $P \leftarrow H(K_w|i)$
   - block $\leftarrow$ S1.remove(P)
   - Iw $\leftarrow$ Iw$\cup$block.l
   - j $\leftarrow$ S0.insert(w,block)
   - S2.update(block.l,block.j,(w|j|'0'))
3. Output I_w

Add file operation

When a file f is added, the server must update S1 with all the new pairs of (w,f) and the S2 with the corresponding sequence of keywords w which should be stored in S2 using a completely new $label_f$. The update of the structure S1 is more complicated because client should retrieve from $ORAM_w$ all the information for every keyword that exist in the new file and send them to server. Server then will compute a new label P for every pair (w,f_{new}) and store it at S1. For the computation, server will use the K_w and c of each file.

The client side

1. f'=SKE.Enc(K,f)
2. $label_f \xleftarrow{\$} \{0,1\}^{\lambda}$
3. For each $w \in W$
   - $(K_w,c)=ORAM_w.read(w)$
   - c++
   - $ORAM_w.write(w,(K_w,c))$
   - $P=H(K_w|c)$
   - add P to list L
4. $ORAM_f.write(f,label_f)$
5. Output L, $label_f$, f'

The server side

for i=1:|L| do
- P=L[i]
- $S1.insert(P,(label_f|i))$
- $S2.insert(label_f,(P|'1'))$

Delete file operation

In order to delete a file f, initially its $label_f$ is needed and so it is retrieved from $ORAM_F$. Using $label_f$ as the label, the server searches at structure S2 and retrieves all the needed information in order to delete the appropriate entries from S0 and S1. The last value of every cell in S2 which is 0 or 1, informs as about the structure, S0 or S1, that this pair is stored to.

If it is 1, then the first value of the cell is the lable of S1 that should be deleted. In case it is 0, the first value is the label for S0 structure and the second one is the j-th value of that label.

Client side

1. $label_f \leftarrow ORAM_f$.read(f)

2.Send $label_f$ at server

3.Send Delete request at cloud storage for $lable_f$

Server side

1.$I_p \leftarrow$ S2.removeALL($label_f$)

\\I_p={first-value|second-value|last-value},second-value exists only at leaked cells

2. for every retrieved I_p do:

```
If last-value=='1' then
        S1.remove(first-value)
else
        S0.remove(first-value,second-value)
```

3.Output "OK"

8.3 Extended Scheme

At this scheme, because the newly added files are treated the same way as in the simple version, the Search and Update leakages are exactly the same as in the simple case. However, there is a difference at the Setup phase, where two new structures are used. They are named $S1$ and $S2$. $S1$, is used to store the inverted index for keywords and $S2$ is used for the file index storage. Both of them are dictionaries with the entries encrypted and a different $S1$ label per entry.

There is also an addition at Secret keys. Besides the three keys of the simple scheme, another λ-bit secret key K is needed. This key is used to compute two keys per keyword and two per file for the initial set of files.

1. $K_w^{(1)} = H(K|1|w)$
2. $K_w^{(2)} = H(K|2|w)$
3. $K_f^{(1)} = H(K|1|f)$
4. $K_f^{(2)} = H(K|2|f)$

For each pair (w, labelf) of the initial set of files, two entries are stored, one into $S1$ and one into $S2$. More precisely, the labels $P_w = H(K_w^1|c)$ and $P_f = H(K_f^1|c')$ are computed, where c, c' are counters on the number of pairs with the same keyword and the same file identifier labelf , respectively.

A pair is stored encrypted into $S1$ with a label P_w as:

$$S_1(P_w) = ((label_f | P_f) \oplus H(K_w^{(2)} | r), r)$$

where r is a random string of length λ. Similarly, an entry of $S2$ is encrypted and stored with label $P_w^{'}$ as:

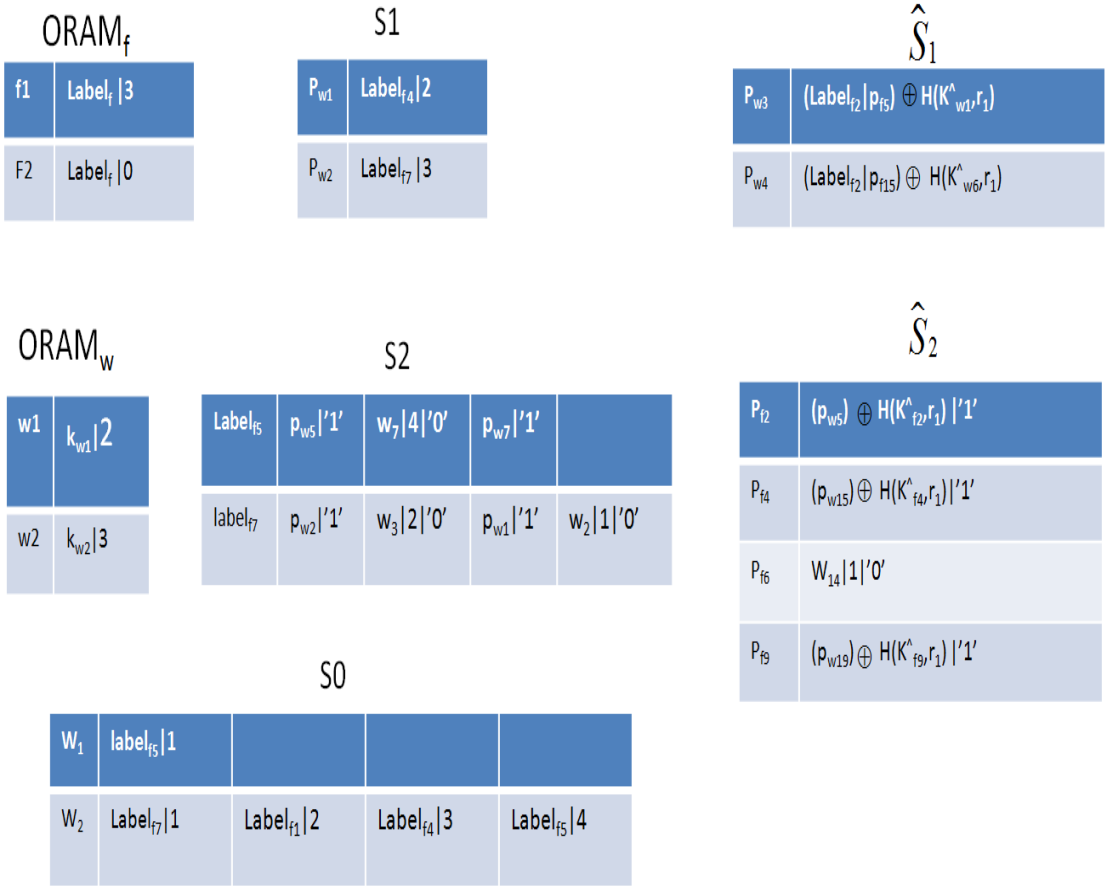
$$S_2(P_w^{'}) = ((P_w \,|\, '1') \oplus H(K_f^{(2)} \,|\, r'), r')$$

where r' is a random string of length λ and '1' means that this pair has not been leaked yet and it is stored in $S1$ with label P_w.

However, when the keyword w is searched, as in the simple scheme, the entry of $S2$ is modified and it is stored again unencrypted with label $label_f$ as:

$$S_2(label_f) = (w \,|\, i \,|\, '0')$$

where '0' indicates that the pair has been read and it is i-th item stored in S0 with label w.



8.2 All structures for the advance DSSE scheme

S_1

The payload of this structure is encrypted until the moment of the first search for that label. The first value if the corresponding labelf of the pair and the second value is the label of that pair at the structure S_2.

S_2

The final value is "1" if the pair is unleaked or "0" if it is leaked. In case the pair is unleaked then the first value is the label at structure S_1 for the corresponding pair. Otherwise the first value is the keyword of the pair and the middle value is the column of S0 structure that should be deleted if that pair should be removed.

Now, only the three operations that differ from the simple scheme will be further explained. The add operation remains exactly the same.

Setup operation

The difference, in comparison to simple scheme is that for every file of the setup phase, the Dataowner needs to compute the values $K_f^{(1)}$ and $K_f^{(2)}$. The first one will be used with the hash function in order to produce the label P_f, of each pair, for the structure $S2$, whereas the second one will be used in order to encrypt the datafield of every entry at $S2$. The structure $S2$ will not have columns anymore and its label won't be the $label_f$ but a P_f. The length of $S2$ will also reveal the number of pairs (keyword, files) at the setup phase.

1. $K \xleftarrow{\$} SKE.Gen(1^{\lambda})$, $K \xleftarrow{\$} \{0,1\}^{\lambda}$

2. $K_F \xleftarrow{\$} \{0,1\}^{\lambda}$, $K_W \xleftarrow{\$} \{0,1\}^{\lambda}$

3. Initiate two ORAMs of size |W| and |F|

4. For each $w \in W$

$K_w \xleftarrow{\$} \{0,1\}^{\lambda}$

5. For each $f \in F$

f'=SKE.Enc(K,f)

$label_f \xleftarrow{\$} \{0,1\}^{\lambda}$

add ($label_f$,f') to list C

$ORAM_f$.write(f,$label_f$)

$K_f^{(1)} = H(K|1|f), K_f^{(2)} = H(K|2|f)$

i=0

For each $w \in W$

i++

$K_w^{(1)} = H(K|1|w)$, $K_w^{(2)} = H(K|2|w)$

$P_f = H(K_f^1|i), P_w = H(K_w^1|i)$

$r_1 \xleftarrow{\$} \{0,1\}^{\lambda}, r_2 \xleftarrow{\$} \{0,1\}^{\lambda}$

$S_1.insert(P_w, ((label_f | P_f) \oplus H(K_w^{(2)} | r_1), r_1)$

$S_2.insert(P_f, ((P_w | '1') \oplus H(K_f^{(2)} | r_2), r_2)$

c(w)=i

For each $w \in W$

c=c(w)

$ORAM_w$.write(w,(K_w,c))

6. Reshuffle C

7. Output C,S1,S2, $S1, S2$

Search operation

The search operation of the advance scheme has one more step comparing to the two steps of the simple one. So, the search for keyword w is performed in both structures S0 and S1 as before. In the third step, $K_w^{(1)}$ is used in order to compute the labels in S_1 ($P_w = H(K_w^1 | c)$) . When all the datafields S_1 are retrieved, they are decrypted by using the key $K_w^{(2)}$ ($K_w^{(2)} = H(K | 2 | w)$). Like in the simple scheme, the entries from both S1 and $S1$ are appended at structure S0, unencrypted.

The client side

1. (K_w,c)<--$ORAM_w$.read(w)

2. $K_w^{'} \xleftarrow{\$} \{0,1\}^{\lambda}$, c'=0

3.$ORAM_w$.write(w,(K'$_w$,c'))

4. $K_w^{(1)} = H(K | 1 | w)$, $K_w^{(2)} = H(K | 2 | w)$

5.Output w, Kw, c, $K_w^{(1)}$, $K_w^{(2)}$

The server side

1. $I_w \leftarrow$ S0.get(w)

2. If $I_w = \varnothing$, then

For c=1 until remove returns $\perp$ do

$P_w \leftarrow H(K_w^1 | i)$

$block \leftarrow S_1.remove(p_w)$

$(data.l | data.l^{'}) \leftarrow (block.\lambda | block.\lambda^{'}) \oplus H(K_w^{(2)} | block.r)$

$I_w \leftarrow I_w \cup$data.l

$j \leftarrow S0.insert(w, data)$

$S_2.update(data.l^{'}, (w | j | '0'))$

3. For i=1:c do

$P \leftarrow H(K_w|i)$

$block \leftarrow S1.remove(P)$

$I_w \leftarrow I_w \cup block.l$

$j \leftarrow S0.insert(w,block)$

S2.update(block.l,block.j,(w|j|'0'))

4.Output I_w

Delete operation

Delete operation musr take under consideration the fact that the file might has been added at a second time or during the setup phase. If it has been added at setup phase, then the server should create and check for that label at $S2$. Whereas, if it had been added with the add operation, server should create the labels and search at structure S2.

The most efficient approach is, the client to retrieve from $ORAM_f$ the file identifier $label_f$ and then to compute the two keys $K_f^{(1)}$ and $K_f^{(2)}$, which will send to the server. The Server then retrieve from S2 all the information related to that $label_f$, more basically the labels P_w for the corresponding entries in S1 or the exact position of the tuple at S0. As a final step, the server also remove the corresponding entries in S0 and S1

In case no match was found at S2 with label the $label_f$, then the server supposes that the file was there from the setup phase. So, the server then computes the related labels used in $S2$ and decrypts their datafields by using the key $K_f^{(2)}$. All found entries are removed from $S2$, and based on their unencrypted information, the server also remove the corresponding entries in S0 and $S1$.

Client side

1. $label_f \leftarrow ORAM_f.read(f)$

2. $K_f^{(1)} = H(K|1|f)$, $K_f^{(2)} = H(K|2|f)$

3.Output $label_f$, $K_f^{(1)}$, $K_f^{(2)}$

4.Send Delete request at cloud storage for $lable_f$

Server side

1. $flag \leftarrow$ 'new'

2.$I_p \leftarrow S2.removeAll(label_f)$

\\I_p={firtst-value|second-value|last-value},second-value exists only at leaked cells

3. If $I_p == \varnothing$

flag='init'

For i=1:c until remove returns $\perp$

$P_f = H(K_f^1 | i)$

$block \leftarrow S_1.remove(P_f)$

$I_p \leftarrow I_p \cup block.l \oplus H(K_f^{(2)} | block.r)$

4. For $p \in O_p$

If last-value=='0' then

S0.remove(firtst-value,second-value)

else if flag=='new'

S1.remove(first-value)

else

S_1 .remove(first-value)

5.Output "OK"

8.4 Leakage

The author used the standard simulation model to define the scheme's security. The scheme is secure in the semi-honest model, where the server follows the protocol, but is curious. The Server is allowed to learn some information and leakage functions are used to define this knowledge. Three leakage functions exist, namely L_{setup}, L_{search} and L_{update}.

The definition of leakage for that scheme captures forward privacy, because the leaked set I_w contains only documents that were added in the past, but no future file additions. On the other hand, I_w can contain deleted files and as a result the definition of leakage does not satisfy backward privacy.

More analytically, during the Setup operation it is leaked the number of files |F|, the number of keywords |W| that will be inserted at cloud server and the size of each file. Moreover at simple scheme server also learns the number of keywords per file.

A search operation for a keyword w leaks the keyword w and the set of file identifiers I_w , who match that keyword that have been added or even deleted in the past, which is the access pattern at time t $ACCP_t(w)$). Moreover a curious server could also learn the time the same keyword was accessed in the past, which is the search pattern SEAPt(w)).

During the add operation, it is only leaked the $label_f$ of the file that is added, the file size len(f) and the number of keywords that it contains. The last is leaked from the number of labels that the server will add at structure S1. On the other hand, at delete operation, which is also part of the update leakage, again the $label_f$ is revealed. Finally, for the update operation, the server can understand if the operation is adding or deleting a file.

Chapter 9

Best Practices for Dynamic Searchable Encryption

1. For efficient search operation, the tuples that are used must be placed in lexicographical order.
2. In order to achieve forward privacy the head of the keyword list should be stored in an ORAM structure.
3. For forward privacy, the encryption key after every search should be different, so that new pairs are encrypted with new keys.
4. For efficiency, it is a good practice to store extra information on the head tuple of the list.
5. Revocation lists for deleted files should be avoided, because they introduce extra overhead for also reading the revocation list.
6. Padding techniques are inefficient for large scale databases.
7. Bottleneck for the search operation is the disk that should be read and moreover to encrypt or decrypt data from random places of those disks. Using RAM instead of HDD is a solution for storing indexes or lists.
8. parallel searching capability should only be inserted if there is no other drawback from it. Otherwise, due to the amount of simultaneously searched keywords the CPU threads will probably be used for different queries and not for speeding up a single search.
9. Boolean search capability is crucial in order for the DSSE scheme to be practical for using it in the real world scenarios.
10. In order for a scheme to be practical it should also support Multi-Client environment.

Chapter 10

Boolean Queries

10.1 Introduction

Despite the fact that all the previous DSSE schemes are efficient and they offer good privacy, they offer limited capabilities as far as the search operation is concerned. A client can only specify a single keyword, which he wants to search for, and then he receives all of the documents containing that keyword. In real world scenarios, like remotely-stored email or large databases, a single-keyword search will often return a large number of documents that the client must then download and filter himself in order to find the relevant results.

This limitation should be overcome, so that the client will have the ability to search with conjunctive queries or even with Boolean queries on the Encrypted database. This means that, given a set of keywords the cloud server should find all documents that contain all these keywords.

An illustrative example is in the case of a conjunction of two highly-frequent keywords, but whose intersection returns a small number of documents. When searching for name=Maria AND gender=Male, the amount of returned results will be minimum, despite the fact that individually, both terms of the query are highly frequent.

A solution that was initially proposed was the server to search for all the keywords of the query separately and then to combine the results in order to return the answer at the Client. This often results in inefficient searches and significant leakage. For the first disadvantage, we can just imagine that some keywords might have as answer a huge number of documents, perhaps half the database size. From the leakage point of view, all file identifiers for every keyword are revealed at server, although they don't satisfy the conjunctive query. In addition, this proposal could not extend to achieve Boolean queries.

Ideally, the search should be run with complexity proportional to the number of matches of the least frequent term in the conjunction, which is the standard of plaintext information retrieval algorithms. This can be achieved, but in order to be efficient, some forms of access-pattern leakage should be allowed. At least this leakage profile would be much better than the firstly proposed solution.

The scheme that achieved the above was introduced be Cash et. al at "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries" and was named OXT. The central idea of this protocol design is a “virtual” secure two-party protocol in which the server holds encrypted pointers to documents, the client holds a list of keywords, and the output of the protocol is the set of encrypted pointers that point to documents containing all the client’s keywords. The client is then able to decrypt these pointers and obtain the matching (encrypted) documents but the server cannot carry this decryption nor can it learn the keywords in the client’s query.

While the protocol is interactive, the level of performance targeted by that solutions required to avoid multiple rounds of interaction. In order to avoid them, the authors proposed to pre-compute parts of the protocol messages and to store them encrypted at the server. During search operation, the client sends information to the server that allows to unlock these pre-computed messages without further interaction. For their implementation they used DH-type operations over any Diffie-Hellman group.

Moreover, the complexity of the search protocols is independent of the number of documents in the database. Complexity depends only on the number of documents which match the least frequent keyword in the conjunction. Of course, a crucial factor is the way that the least frequent keyword is estimated.

With the following example, which is given as an overview of the scheme, it would be easier to understand the basic idea of the scheme. Supposing there is a conjunctive query q, where:

q= w1 AND w2 AND w3

For the estimated most infrequent keyword the server run the search protocol and retrieves the files I_w that contains it. In this example the least frequent is w_2 and the files that contain it are for example the: f_1, f_5, f_8, f_9. The keyword w_2 will be referred as s-term keyword, whereas w_1 and w_3 as x-term keywords.

The second phase of the search protocol would be to find out if these four files also contain both the keywords w_1 and w_3. For that reason, they add to their scheme an extra structure named XSET, which stores an encrypted value for each pair (w,f) that exist in the scheme. These encrypted values are named xtag and depends on both f and w.

$$xtag = xtrap^{xind}$$

where

$$xtrap = g^{F_p(K_x, w)} \quad \rightarrow \text{ depends only on w}$$

$$xind = F_p(K_I, label_f) \rightarrow \text{ depends only on f}$$

and Fp is a PRF with range Z_p^* and keys either K_I or K_x.

So, the server after the first step of the search operation knows the $label_f$, for every file at I_w, for keyword w_2 and so can compute the four xind values. Based on the two x-term keywords w_1 and w_3, server can also compute the two xtrap values. By combining them he has 8 xtag values and now the server can search on XSET structure in order to find which of these eight values exist inside the XSET. If an xtag is found in XSET, that means that the corresponding pair also exist in the EDB.

If all the xtags that was generated by the xind of a file were found in the XSET, then it is assumed that the file fulfill the requirements of the query.

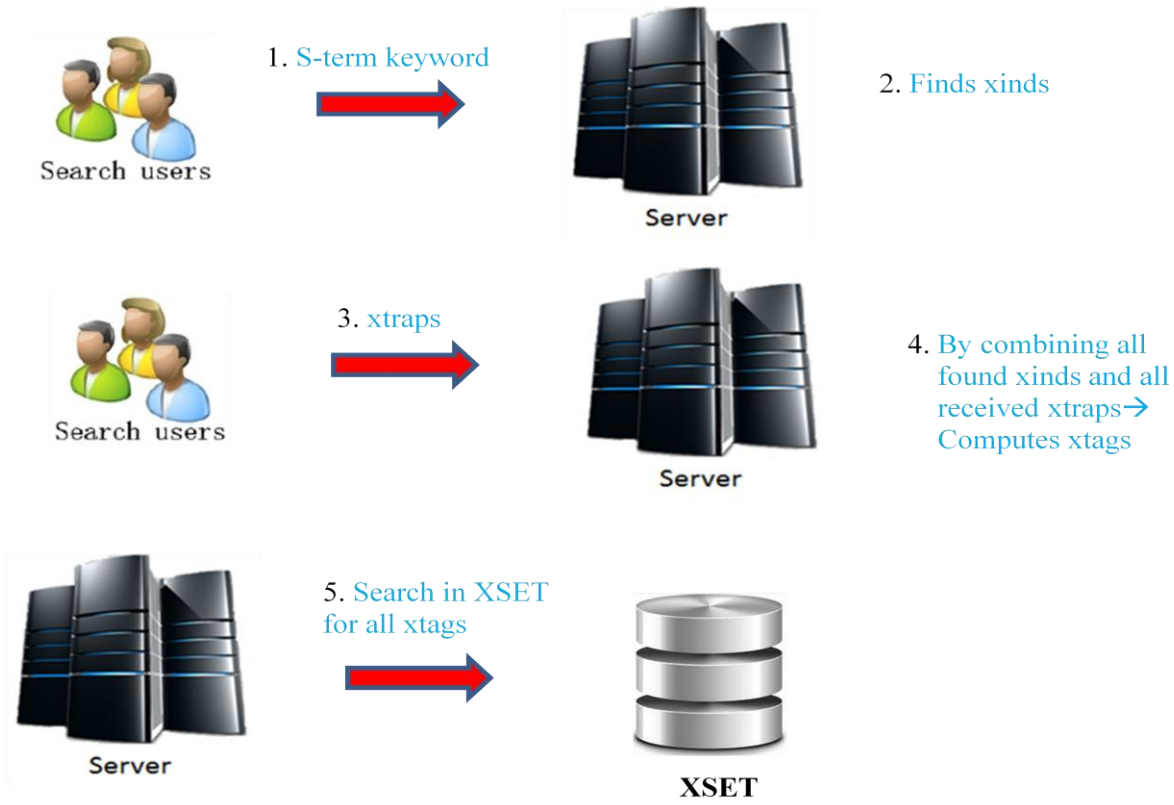


Figure 10.1 - Example of xtag computation

Security issue.

In order for the server to compute the xtags values, client sends the strap values for the x-term keywords and he also uses the xind values that match the s-term keyword. The security issue of this approach is that, nothing can stop a curious server from combining the new strap values with old xinds, that the Server has stored from previous search operations, in order to find out if the combined pairs also exist in the Encrypted Database. This leakage was not tolerable.

Solution

Every pair at the EDB will store a pre-computed blinded value named Y_c. This value will blind the xind of the corresponding file of the pair, that is stored at that position. The blinded factor would be the Z_c

$$Yc=xind\ Z_c^{-1}.$$

During the search phase, the client will also send the corresponding factors Z_c, in order for the server to be able to unblind the values of Yc, that will retrieve from the search of the s-term keyword. Moreover, with this approach there is no need for the server to knows the keys K_x and K_I of the PRF that produces the xind values.

The blinding factor Z_c is calculated as follows:

$$Zc=PRF(w,c)$$

where w is the keyword and c is a counter which increments by one for each different file associated with that keyword w.

Finally, it should be mentioned that when a Boolean query is negations (NOT) then the server also computes the value xtag, but he returns the files only if the xtag is not found in the XSET structure.

Cash et. al. Implemented their scheme that supports Boolean queries based on their DSSE proposal. Their DSSE Proposal is based on the inverted index approach, like the majority of DSSE schemes. One of the major goal of this thesis was to try to implement the addition of Boolean queries on the DSSE scheme that was presented in the previous chapter " ORAM based forward privacy preserving Dynamic Symmetric

Searchable Encryption Scheme ". This proposal as mentioned before, has a completely different approach and it uses dictionaries in order to reduce leakage and improve efficiency.

The following figure will help to better understand the structures that are used in this scheme.

ORAM$_f$

It Stores the random label$_f$ of every file. Moreover, if the last value is "1" it means that the file had been added at setup phase. Only then there is a middle value for the number of keywords that the file contains.

ORAM$_w$

It stores the current key of every keyword and the number of unleaked files that contain this keyword. The last value is the number of files that ever had or have that keyword. Even in case of deletion that counter is not reduced. It is used for the computation of Z_c.

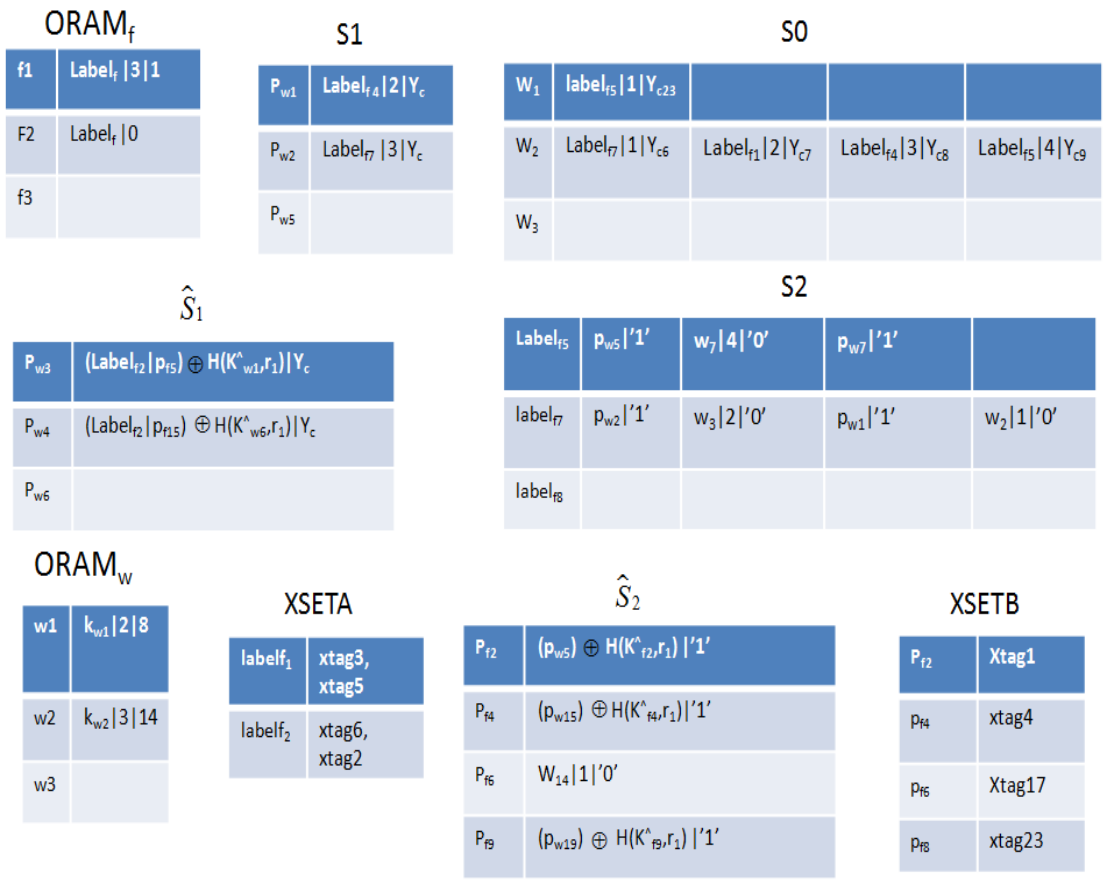


Figure 10.2 - All structures of the DSSE scheme

S1

The first value is the $label_f$ of the file of the corresponding pair. The second value points out the column of the S2 structure, which will be leaked, due to the search that returned that $label_f$. The third value is the blinded value of the xind of that label.

S2

The final value is "1" if the pair is unleaked or "0" if it is leaked. In case the pair is unleaked then the first value is the label at structure S1 for the corresponding pair. Otherwise the first value is the keyword of the pair and the middle value is the column of S0 structure that should be deleted if that pair should be removed.

S0

The first value is the $label_f$ of the file of that pair. The second value is used in order to sort again the row in case of deletion of a cell. Again all cells have as last value the corresponding value of Y_c.

S_1

The payload of this structure is encrypted until the moment of the first search for that label. The first value if the corresponding labelf of the pair and the second value is the label of that pair at the structure S_2. The last value is unencrypted and it is the Y_c.

S_2

The final value is "1" if the pair is unleaked or "0" if it is leaked. In case the pair is unleaked then the first value is the label at structure S_1 for the corresponding pair. Otherwise the first value is the keyword of the pair and the middle value is the column of S0 structure that should be deleted if that pair should be removed.

XSETA

It stores the xtag values inserted with the addition operation.

XSETB

It stores the xtag values inserted with the setup operation. The label that is used for that structure is the same as in S_2, so that in case of deletion to be able to delete the correct xtag from the XSETB structure.

At the next chapter, there are presented the algorithms for each phase of the Dynamic Searchable Encryption proposal, which are based on the scheme of Rizomiliotis for single keyword search.

10.2 Algorithms

In order for a DSSE scheme to support Boolean queries, it should also support at delete phase the deletion of the xtags that correspond at the deleted pairs of the file. In order to achieve that, we split the XSET structure to two different structures. The first one is called XSETA and it stores the xtags of the files that were added with the addition operation. The second one is called XSETB and it stores the xtags of the files that were added at setup phase. So in that way all the xtags of all the pairs of the setup phase are stored with the same label as the corresponding pairs are stored at S_2.

At Setup phase, the 3 for steps are for key generation and fourth, fifth are for ORAM structure initiation. At step 6, every file of the setup phase is encrypted and calculate the parameters $K_{fj}^{(1)}$, $K_{fj}^{(2)}$ which will be later user for the calculation of the labels of S_2 and the encryption of the payload stored at S_2. xind_j will be used for the calculation of the xtags of that file.

Continuously at step 6 for every keyword of every file, there are retrieved from the ORAM all the needed information. Counter c reveals the number of unleaked files that have this keyword, whereas m is the total number of files, even if they are deleted now, that ever had that keyword. With these counters, the Dataowner computes the keys $K_{wi}^{(1)}$, $K_{wi}^{(2)}$ which will be used at computing the labels of structure S_1 and the encryption of the payload stored at S_1.

Finally S_1 and S_2 are filled with the information. Y_c is the blinded factor in order for the server to store the value of the xind encrypted. The key for the decryption is Z_c which is sent by the client during the search phase. Z_c is computed using a PRF , the keyword w_i and a counter m, so that it won't be possible that two pair (w,f) with the same w will ever have the same value of Z_c. This is the reason that the value of m at ORAM_w, is never reduced even when a pair is deleted.

Setup Phase

1. $K \leftarrow SKE.Gen(1^{\lambda})$
2. $K \leftarrow \{0,1\}^{\lambda}$, $K_I \xleftarrow{\$} \{0,1\}^{\lambda}$
3. $K_w \leftarrow \{0,1\}^{\lambda}$, $K_f \leftarrow \{0,1\}^{\lambda}$
4. Initiate $ORAM_w$ of size |W|
5. Initiate $ORAM_f$ of size |F|
6. j=0

   For each $f \in F$ do

   j++

   f_j'=SKE.Enc(K_f,f_j)

   $label_{fj} \leftarrow \{0,1\}^{\lambda}$

   add ($label_{fj}$,f_j' to list L1)

   $K_{fj}^{(1)} = H(K|1|f_j)$

   $K_{fj}^{(2)} = H(K|2|f_j)$

   $xind_j = F_p(K_I, label_{fj})$

   i=0

   for each $w \in W_f$ do

   i++

   $K_w \leftarrow \{0,1\}^{\lambda}$

   c,m $\leftarrow$ ORAM.READ(w_i)

   m++, c=++

   $ORAM_w$.write(w_i&K_w,c,m)

   $K_{wi}^{(1)} = H(K|1|w_i)$

   $K_{wi}^{(2)} = H(K|2|w_i)$

   $P_{fj} = H(K_{fj}^{(1)}|i)$

   $P_{wi} = H(K_{wi}^{(1)}|i)$

$r_1 \leftarrow \{0,1\}^{\lambda}$, $r_2 \leftarrow \{0,1\}^{\lambda}$

Z_c=PRF(w_i,m) $\in Z_p$

Y_c=xindZ_c^{-1}

$$S_1.insert(P_{wi},((label_{fj} \mid P_{fj}) \oplus H(K_{wi}^{(2)} \mid r_1), r_1) \mid Y_c$$

$$S_2.insert(P_{fj}, P_{wi} \oplus H(K_{fj}^{(2)} \mid r_2), r_2) \mid '1'$$

$$xtag = g^{F_p(K_x, w_i)xind}$$

XSETB.insert(p_{fj},xtag)

ORAM$_f$.write (f,label$_f$|i|1)

7. Reshuffle L1 and store it at STORAGE.

8. Output XSETB , S_1 and S_2 at Server

Add Phase

The add operation have many similarities with the setup phase. The major difference is that it only involves one file. After the encryption of the file and the creation of its label$_f$, the value xind is calculated. Again for every keyword of the file the same approach is followed, as in setup phase, with the exception that now the inverted index is stored at S1 and file index is stored at S2

Στον Client (Dataowner)

f'=SKE.Enc(K,f)

$label_f \xleftarrow{\$} \{0,1\}^{\lambda}$

add ($label_{fj}$,f_j') to list L1

xind=$F_p(K_I,label_f)$

i=0

for $w \in W$ do

i++

(K_w,c,m)=ORAM.READ(w)

c++ , m++

ORAM.WRITE (w,&K_w,c,m)

p_{wi}=H(K_w,c)

add p_{wi} to list L2

Z_c=PRF(w_i,m)

Y_c=xindZ_c^{-1}

add Y_c to list L3

$xtag_i = g^{F_p(K_x,w_i)xind}$

Add $xtag_i$ to List L4

$ORAM_f$.WRITE (f, $label_f$|'0')

Output L2,L3,L4,$label_f$ at Server

Reshuffle L1 and store it at STORAGE

Στον SERVER

for j=1:|L2| do

p_w=L2[j]

Y_c=L3[j]

S1.insert (p_w , $label_f$ | j | Y_c)

S2.insert ($label_f$, p_w | '1')

XSETA.insert ($label_f$, $xtag_i$)

Search Phase

Search phase is an interactive phase. Initially, the client choose the s-term keyword from his query and retrieves from the $ORAM_w$ the information about that keyword. He then calculates the values $K_w^{(1)} = H(K|1|w)$, $K_w^{(2)} = H(K|2|w)$ and sends them at server along with the s-term keyword, the temporary key for that keyword and the number of the rest keywords of his Boolean query (t).

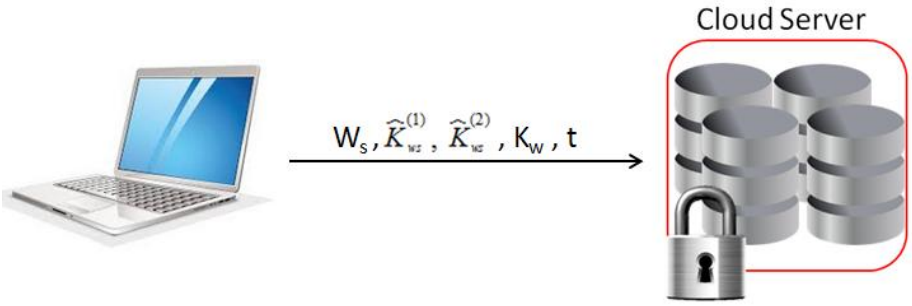


Figure 10.3 - Information sent during the first step of the query

With these values the server firstly retrieve the leaked information for the files that contain that keyword from S0. The next step is to search for the unleaked files at S_1, by computing the labels that are needed for accessing the structure S_1. The first portion of the payload from S_1 must be decrypted with the use of the key $K_w^{(2)}$. These

information is then considered as "leaked" and should be removed from S_1 and stored at S0. The same procedure is followed for the searching at S1 structure.

All the retrieved file identifiers are stored at I_w and their respective values of Y_c are stored at List L8. Server creates an empty table(x,y) where x=|L8| and y is the number of x-term keywords. The cells of this table will be stored with the tokens that the client will send at the next phase.

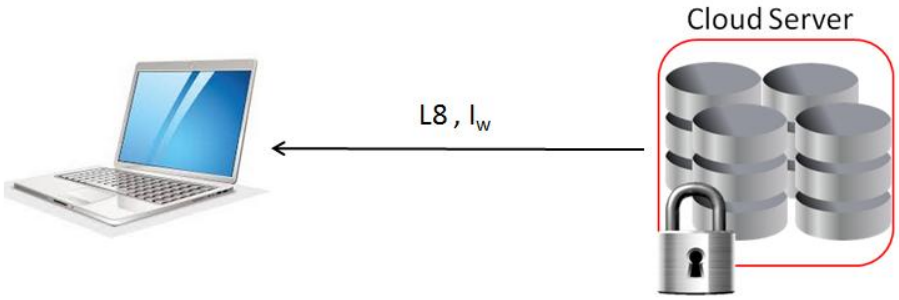


Figure 10.4 - Information sent during the second step of the query

Client receives the I_w and the list L8 that contains the values of Y_c. Now he can computes the xind values based on the $label_f$ of the I_w and then the corresponding value of Z_c for every file retrieved for keyword w_s. Afterwards, for every x-term keyword of every file of I_w client computes the values token(x,y) that sends at server with list L9.

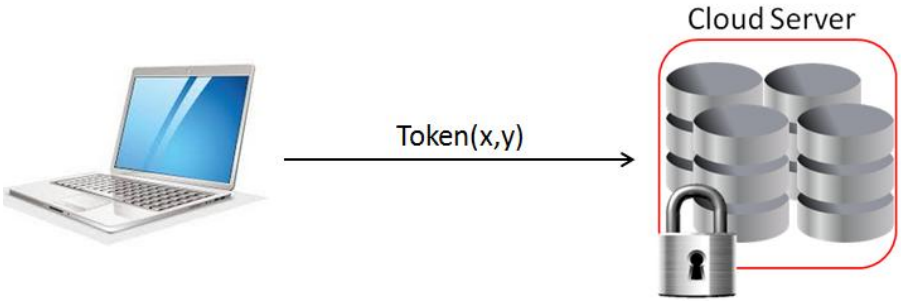


Figure 10.5 - Information sent during the third step of the query

Finally the server fill the table(x,y) with the tokens(x,y) that received from client. Then by searching the values of all xtags at Structures XSETA and XSTEB server decides if any of the rows of the table fulfill the Boolean query.

Dataowner-Client side

1. Choose the s-term keyword (w_s) from the query, the number of the rest w is "t"

2.$(K_w,c,m) \leftarrow ORAM_w.read(w_s)$

$K_w' \leftarrow \{0,1\}^{\lambda}$, c'=0

$ORAM_w.write(w , K_w'|c'|m)$

3. $K_{ws}^{(1)} = H(K|1|w_s)$

$K_{ws}^{(2)} = H(K|2|w_s)$

4. Output: w_s, $K_{ws}^{(1)}$, $K_{ws}^{(2)}$,K_w , t

Server Side

1. For i=1 until found "nothing" do

$(label_{fi},Y_{ci}) \leftarrow S0.get(w)$

$I_w \leftarrow I_w$ U $label_{fi}$

Add Y_{ci} to L8

2. n=0

for n until p_{wn} not found at S_1 do

n++

$p_{wn} \leftarrow H(K_w^{(1)},n)$

$block.\lambda \mid block.\lambda'' \leftarrow S_1.remove(p_{wn})$

Y_c=block.λ"

Add Yc to L8

$a|b \leftarrow block.\lambda \oplus H(K_w^{(2)}|block.r)$

$label_{fn}$=a

$I_w \leftarrow I_w$ U $label_{fn}$

j $\leftarrow$ S0.insert (w, $label_{fn}$ | j | Y_c)

p_f=b

$S_2.update(P_{fn}, w_n \mid j \mid '0')$

3. n=0

for n until p_{wn} not found at S1 do

n++

$p_{wn} \leftarrow H(K_w,n)$

a|b|c $\leftarrow$S1.remove(p_w)

Y_c = c

Add Yc to L8

$label_f$=a

$I_w \leftarrow I_w$ U label

j $\leftarrow$ S0.insert (w_n , $label_{fn}$ | j | Y_c)

S2.update ($label_f$/b , w_n | j | '0')

4. x=|L8|

y=t

Create an empty table(x,y)

5. Sent to Client : L8 , I_w

Dataowner-Client side

1. for x=1:$|I_w|$ do

Y_{cx}=L8[x]

$label_{fx}$=I_w[x]

xind=$F_p(K_I, label_{fx})$

Z_{cx}=xind Y_{cx}^{-1}

for y=1:t do

$$xtag_i = g^{F_p(K_x, w_i)xind}$$

$$token(x, y) = (xtrap_y)^{Z_{cx}}$$

add token(x,y) to L9

2. Client sends to Server L9

Server Side

1. Server Stores tokens(x,y) at table(x,y)

2. I_{wfinal}=empty

3. For x=1:|L5| do

for y=1:t do

xtag=$(token(x,y))^{Ycx}$

if xtag € XSETA($label_x$)

THEN table(x,y)==TRUE

elseif xteg € XSETB

THEN table(x,y)==TRUE

else table(x,y)==FALSE

4. For x=1:|L5| do

if boolean function is TRUE

then I_{wfinal}= I_{wfinal} U I_{wx}

Server outputs I_{wfinal}

Delete Phase

The first step of the deletion operation is to read from the $ORAM_f$ all the related information about the deleted file. This information and especially counter z will reveal if the file had been added during the setup phase or with the addition operation. In case z=1 then the file was inserted at setup phase and so the deleted values would be from structures S_1, S_2 or S0 for the leaked pairs. If on the other hand z=0, then Server understand that the file had been added with addition and so the values of the pairs will be removed from S1, S2 or S0.

Value x at $ORAM_f$ exist only for files from the setup phase. It is used in order to calculate the labels P_f for the S_2. Finally in both structures S2 and S_2, if counter c=0, then server understands that the pair is already leaked and so server deletes the appropriate cell of the S0 structure. After every deletion of a cell at S0, S0 is auto-restructured in order not to have empty cells for locality and efficiency.

Client (Dataowner) Side

1. $y|x|z \leftarrow ORAM_f.READ(f)$

2. $ORAM_f.DELETE(f)$

3. $label_f=y$

4. $K_f^{(1)} = H(K|1|f)$

   $K_f^{(2)} = H(K|2|f)$

5. Dataowner sends to Server: $K_f^{(1)}$, $K_f^{(2)}$, x, y, z

6. $Storage.Delete(label_f)$

Server Side

if z==1 then

for i=1:x do

$P_f = H(K_f^1 \mid i)$

$a \mid b \mid c \leftarrow S_2.read(P_f)$

$S_2.delete(P_f)$

XSETB.delete(P_f)

if c==0 do

S0.remove(a,b)

max ← count.S0(a)

a | b''' | c''' ← READ.S0(a,max)

Remove.S0(a,max)

Write.S0(a/b , a | b | c''')

else do

$a' \leftarrow a \oplus H(K_f^{(2)} \mid block.r)$

S_1 .remove(a')

if z==0 then

a | b | c ← S2.READ($label_f$)

XSETA.DELETE(labelf)

S2.DELETE($label_f$)

if c==0 do

S0.remove(a,b)

max ← count.S0(a)

a | b''' | c''' ← READ.S0(a,max)

Remove.S0(a,max)

Write.S0(a/b , a | b | c''')

else do

S1.remove(a)

10.3 Toy Example

In this chapter, a toy example will be presented in order to explain the design philosophy of this proposal.

Setup Phase

At Setup phase, there exist four files with four keywords. The initial keyword sequence is as follows:

- f_1 with keywords w_1,w_2,w_3
- f_2 with keywords w_4,w_1
- f_3 with keywords w_2,w_1,w_4
- f_4 with keywords w_3

For every phase a figure will present the state of each structure at the end of the phase. Red color indicates that there were a change at that row of the table.

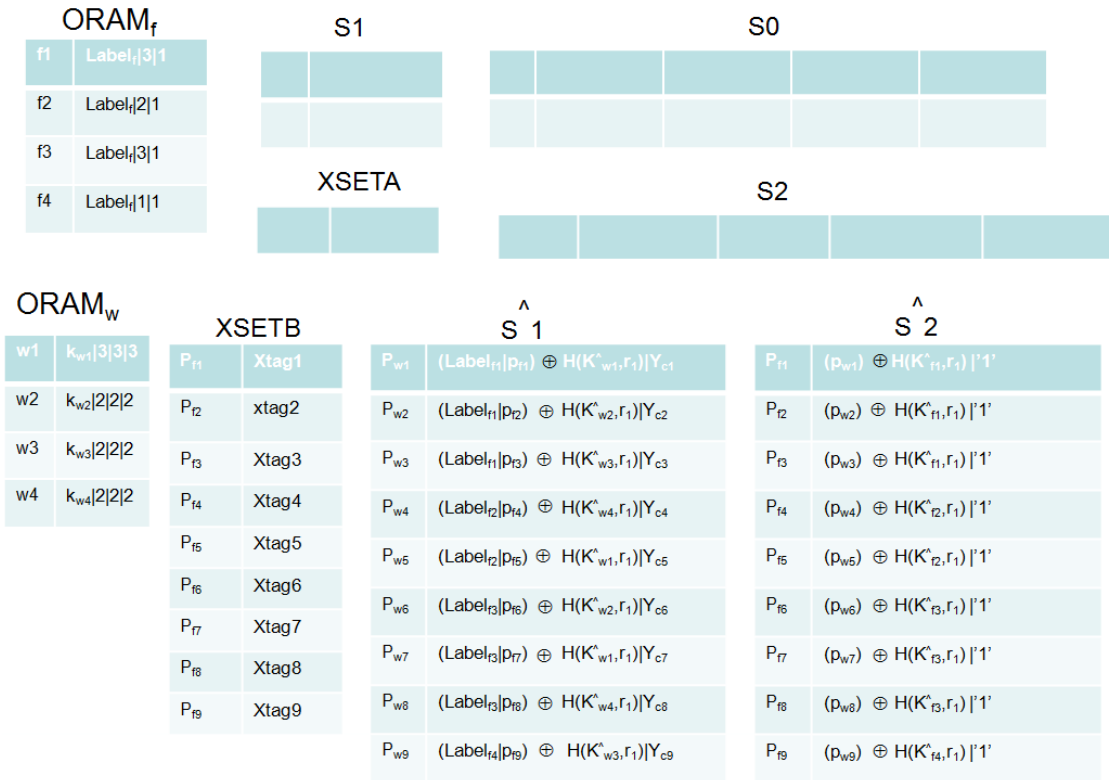


Figure 10.6 - State of structures after Setup phase.

Search Phase

At this phase the query of the search would be the:

w_1 AND w_2 AND w_4

w_2 keyword will be the chosen s-term keyword

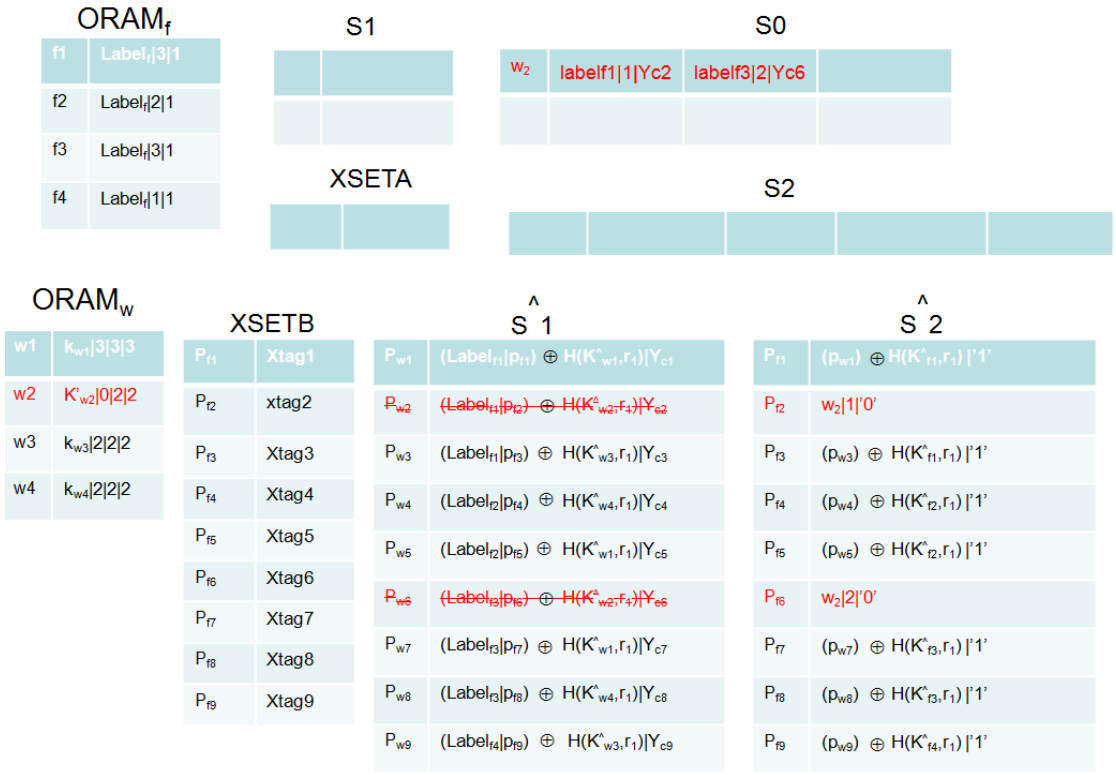


Figure 10.7 - State of structures after Search phase.

All pairs that were leaked from S_1 are moved at structure S0. Furthermore, all corresponding tuples of structure S_2 are leaked.

Add Phase

At this phase a new file (f_5) will be added at the scheme. f_5 contains the following keywords: w_5, w_2 and w_4

Figure 10.8 - State of structures after Add phase.

As it was expected, the new pairs were added at structure S1 and S2 which are dedicated for the pairs of the add operation.

Search Phase

At the second search operation of this toy example the Boolean query is the following:

$$(w_3 \text{ AND } w_5 \text{ AND } w_2) \text{ OR } w_4$$

Because of the OR, two s-terms keyword are needed w_4 and s-term w_3, so the first figure will be the result of the search operation for keyword w_4 and the second figure will be for keyword w_3. xtag values will be sent from Client to Server only for the I_w of keyword w_3.

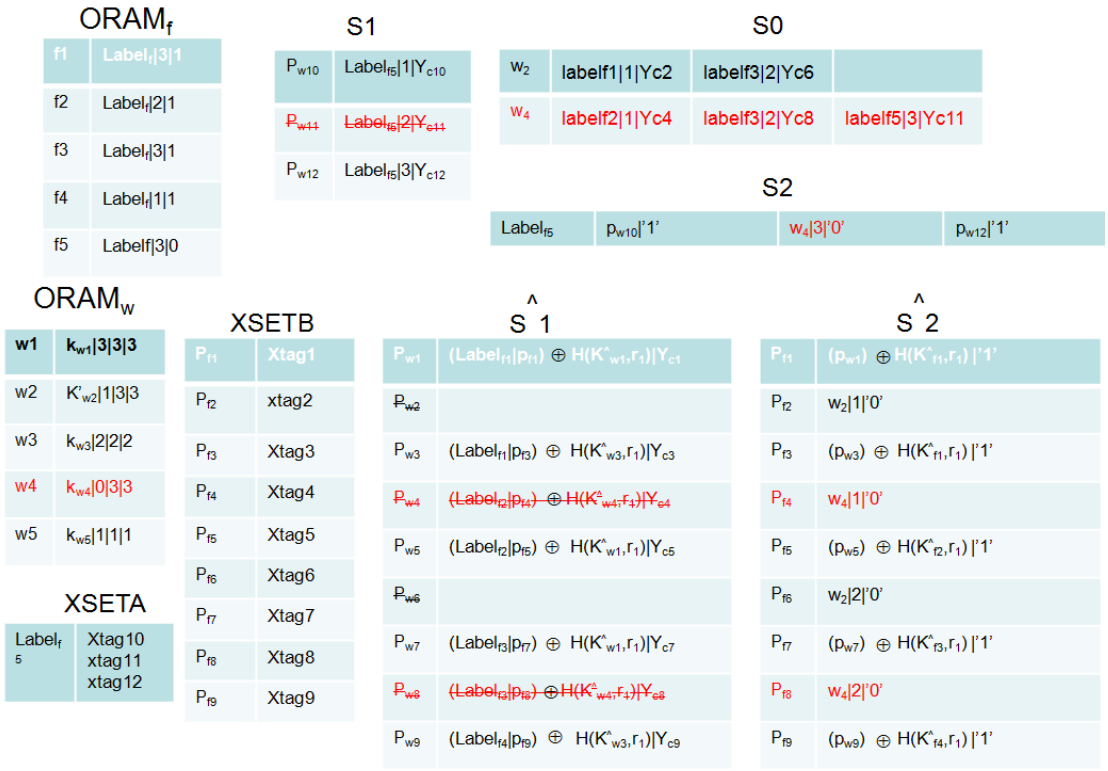


Figure 10.9 - State of structures after Search for keyword w_4.

Figure 10.10 - State of structures after Search for keyword w_3.

Delete Phase

At this phase the deleted file would be the f_3. Based on retrieved information from $ORAM_f$, the server will know that it contains three files.

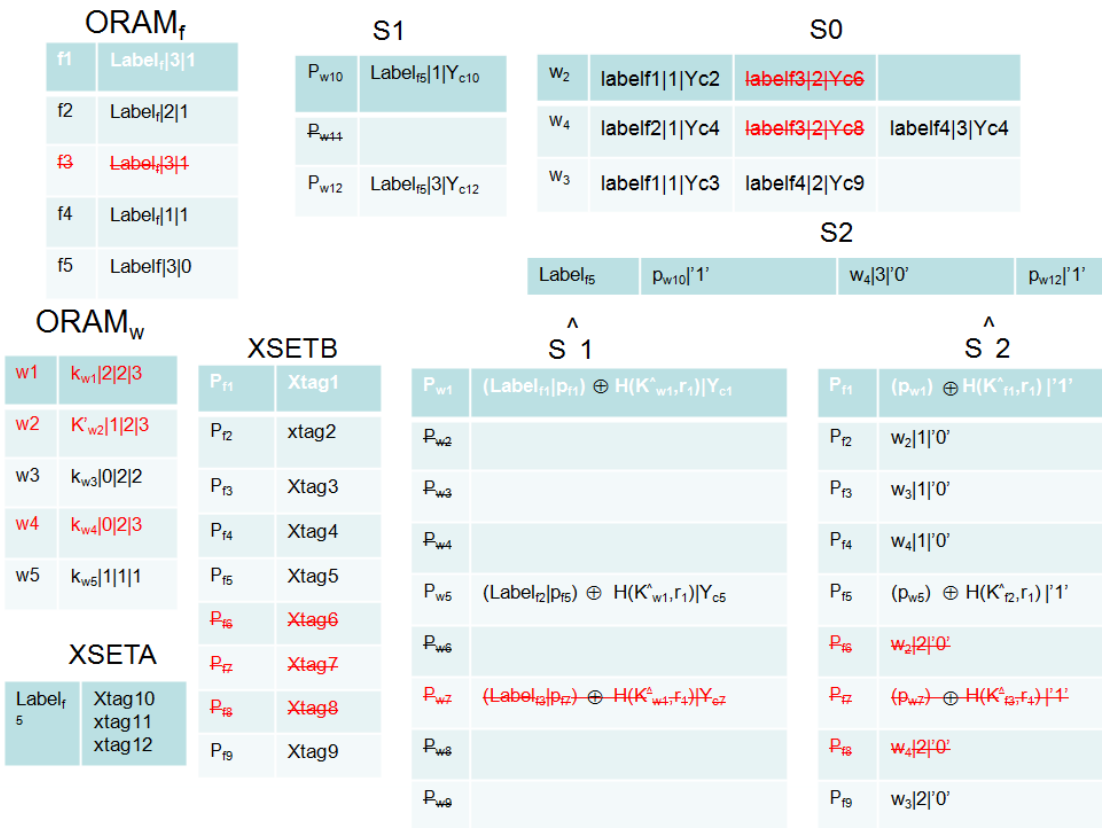


Figure 10.11 - State of structures after Delete of file f_3.

10.4 Leakage

Setup

- Number of Keywords*files

Add

- Operation add or delete
- File Identifier ($label_f$)
- Number of keywords per added file

Search

- Search Pattern (hash of keyword s-term , document identifiers that have been added or deleted for s-term keyword)
- Access Pattern 1 (matching document identifiers of the s-term keyword search)
- Access Pattern 2 (only the matching document identifiers of the x-term keywords that fulfill the Boolean query)
- Time that the same keyword was accessed in the past
- Access Pattern at that time in the past
- Boolean Query Expression (AND-OR-NOT)

Delete

- Operation add or delete
- File Identifier ($label_f$)
- Number of keywords per deleted file

Chapter 11

Multi-Client and Parallel Search

11.1 Introduction

In a simplest multi-client scenario, the Dataowner (D) encrypts and stores his documents at the Server (E). Then the Dataowner authorizes the clients to query the Server with Boolean Queries. Only the Dataowner can add new files or delete existed ones. Moreover only the Dataowner can query both ORAM structures. Both procedures have no changes comparing to the single client scenario, because they are executed only by the Dataowner and the Server.

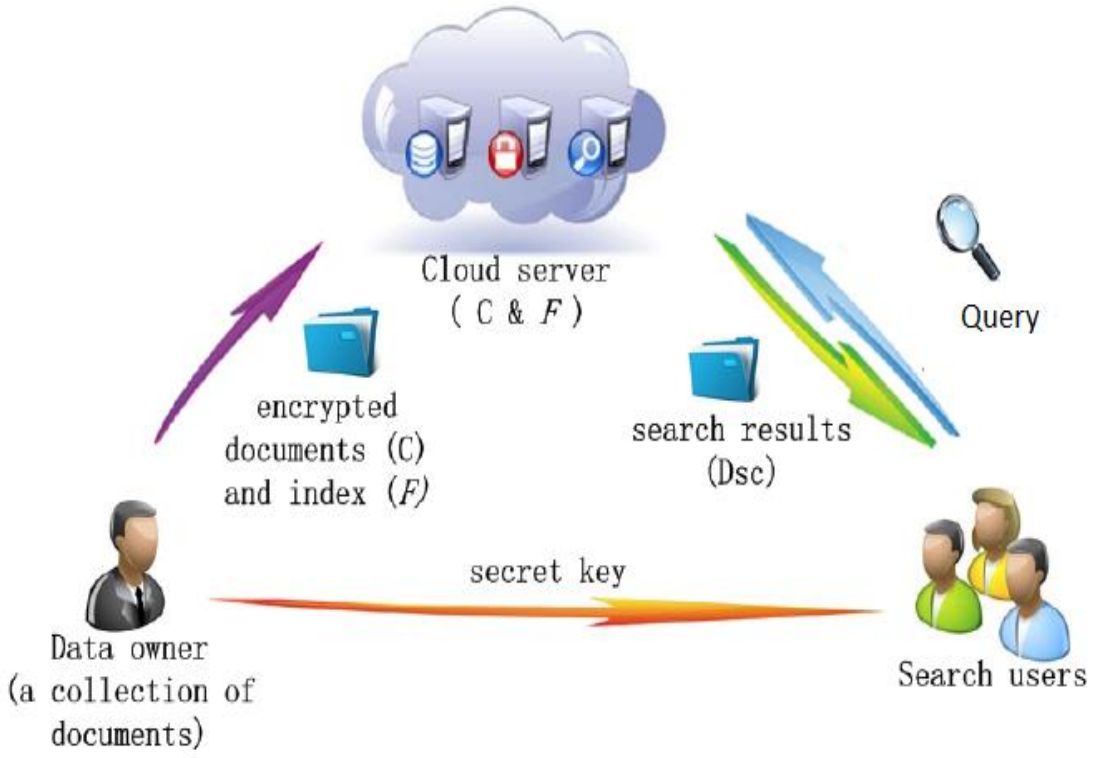


Figure 11.1 - Multi-Client Scenario Topology

During the Search procedure the Dataowner learns the Boolean query from the client and he is the one who will determine the s-term keyword and will also query the $ORAM_w$ structure. A major difference now is that the Dataowner must authorize the query by using a quantity named p_i. He blinds the xtraps values with these quantities and sends them to Server through the Client. The blinded values are called bxtrap.

$$bxtrap=xtrap^{pi}$$

Moreover, the Dataowner will send an encrypted envelope at the Server with key k_e, again through the client. This envelope contains all the blinded factors for every xtrap value.

Using the previous approach the computation time is increased, but it is the only way to authorize a client to do the queries, by not allowing him to have any leakage. Even replay attacks cannot be executed.

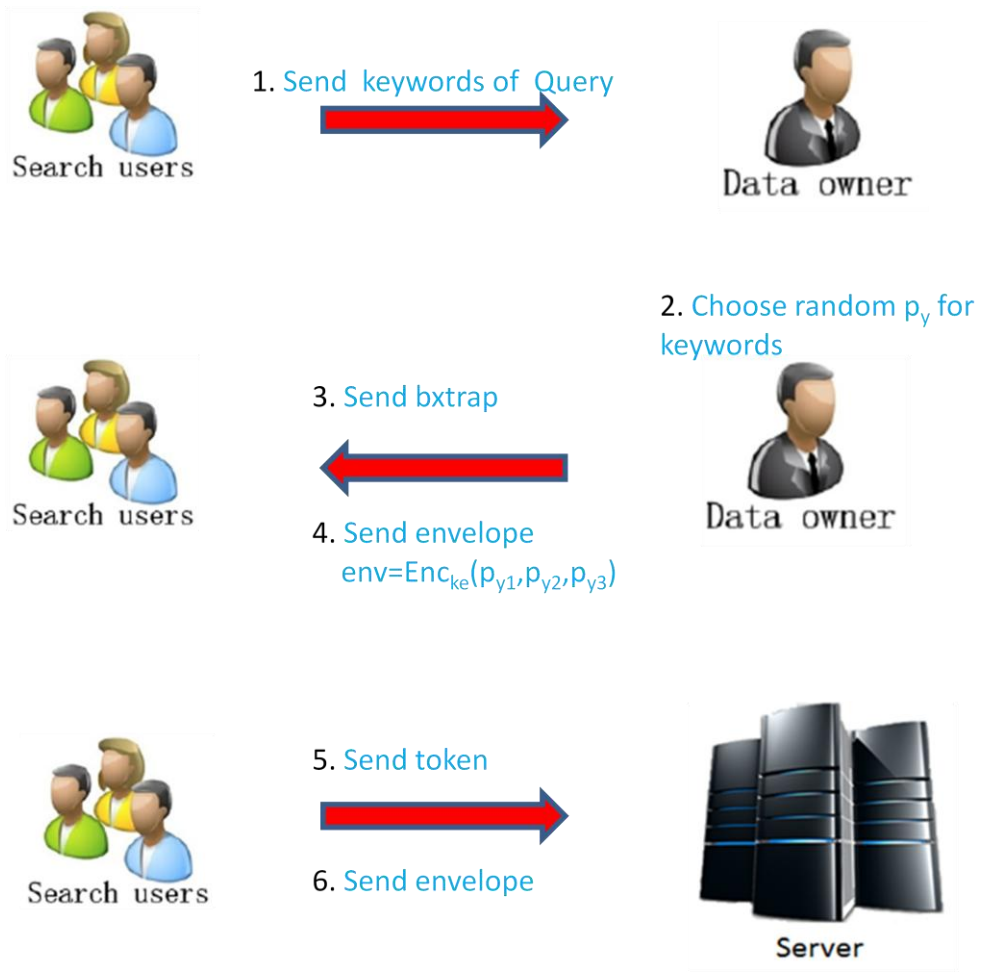


Figure 11.2 - Authorization envelope exchange

In order to make the search procedure even more parallel, the time needed for the delete operation must be sacrificed . This is acceptable because search operations are usually far more than delete operations.

More specifically, three more counters are needed to be stored at $ORAM_w$, which will point out the accurate number of leaked and unleaked files that contains that keyword and whether these pairs exist from the setup phase . If the Server knows

these counters, he would be able to create from the beginning the correct number of labels for searching at S0, S1 or S_1.

The new counter that exist in ORAMw for each keyword are the following:

1. c - The number of unleaked files
2. d - The number of leaked and unleaked files as a total
3. b - The number of unleaked files which exist from the setup phase
4. e - Number of unleaked files from add / No decrease at delete / Zero at Search
5. m - Total number of files ever added (even Setup). Never Decrease

As a result, the number of pairs that Server will find at S0 is d minus c. The number of labels produced for searching at S_1 is equal to b. Finally the number of labels that should be generated for searching at S1 will be equal to c minus b.

Another counter that is needed is the counter e, which is increased for a pair when a new file is added, but it never decreases at delete operations. This counter is only becoming zero when the corresponding keyword is the s-term keyword of a search. By adding that counter there are no more collisions at computation of labels P_w, as there might exist by using the counter c. So, Server will not have to create the labels P_w for S1 structure in serial way, until the client to make him stop. The only drawback is that server will probably create more P_w than needed, but not a lot.

An issue was that in order for these counters to be accurate, every time a file was deleted, the Dataowner should learn the keyword that it contained in order to reduce those counters. In case of deleting a file from S0, this could be done. On the other hand in case of deleting a cell from S1 or S_1, this is not possible with the existing format, because the payload of them can not reveal the keyword.

For the reason above it was mandatory to also include in the payload of S1 and S_1 structures the corresponding keyword encrypted.

11.2 Algorithm

Setup Phase

The only changes of the Setup algorithm, comparing to the previous presented scheme, are:

- The increased number of counters that are stored at structure $ORAM_w$, in order to make the search operation more parallel.
- Blinded Keyword stored at S2 and S_2 structures.

1. $K \leftarrow SKE.Gen(1^{\lambda})$
2. $K \leftarrow \{0,1\}^{\lambda}$, $K_I \xleftarrow{\$} \{0,1\}^{\lambda}$
3. $K_w \leftarrow \{0,1\}^{\lambda}$, $K_f \leftarrow \{0,1\}^{\lambda}$

4. Initiate $ORAM_w$ of size |W|
5. Initiate $ORAM_f$ of size |F|

6. j=0
   For each f ∈ F do
   j++
   f_j'=SKE.Enc(K_f,f_j)
   $label_{fj}$ <--$\{0,1\}^{\lambda}$
   add ($label_{fj}$,f_j' to list L1)

   $K_{fj}^{(1)} = H(K \mid 1 \mid f_j)$

   $K_{fj}^{(2)} = H(K \mid 2 \mid f_j)$

   $xind_j = F_p(K_I, label_{fj})$

i=0

for each $w \in W_f$ do

i++

K_w <--$\{0,1\}^{\lambda}$

c,e,b,d,m <-- ORAM.READ(w_i)

m++ , c=++ , b++ , d++

$ORAM_w$.write(w_i , K_w | c | e | b | d | m)

$$K_{wi}^{(1)} = H(K|1|w_i)$$

$$K_{wi}^{(2)} = H(K|2|w_i)$$

$$P_{fj} = H(K_{fj}^{(1)}|i)$$

$$P_{wi} = H(K_{wi}^{(1)}|i)$$

r_1<--$\{0,1\}^{\lambda}$, r_2<--$\{0,1\}^{\lambda}$

Z_c=PRF(w_i,m) $\in$ Z_p

Y_c=xindZ_c^{-1}

$$S_1.insert(P_{wi},((label_{fj}|P_{fj}) \oplus H(K_{wi}^{(2)}|r_1),r_1)|Y_c$$

$$S_2.insert(P_{fj},(P_{wi}|w_i) \oplus H(K_f^{(2)}|r_2),r_2)|'1'$$

$$xtag = g^{F_p(K_x,w_i)xind}$$

XSETB.insert(p_{fj},xtag)

$ORAM_f$.write (f,$label_f$|i|1)

7. Reshuffle L1 and store it at STORAGE.

8. Output XSETB , S_1 and S_2 at Server

Add phase

The only changes of the Addition algorithm, comparing to the previous presented scheme, are:

- The increased number of counters that are stored at structure $ORAM_w$, in order to make the search operation more parallel.
- Blinded Keyword stored at S2 and S_2 structures.

Στον Client (Dataowner)

f'=SKE.Enc(K,f)

$label_f \xleftarrow{\$} \{0,1\}^{\lambda}$

add ($label_{fj}$,f_j') to list L1

xind=$F_p(K_I,label_f)$

i=0

for $w \in W$ do

i++

(K_w,c,e,d,b,m)=ORAM.READ(w)

c++ , m++ , e++ , d++

ORAM.WRITE (w , K_w | c | e | d |b |m)

$p_{wi}=H(K_w,e)$

add p_{wi} to list L2

$Z_c=PRF(w_i,m)$

$Y_c=xindZ_c^{-1}$

add Y_c to list L3

$B_i = w_i \oplus H(K_f^{(2)} \mid r_2), r_2$

Add Bi to List L4

$xtag_i = g^{F_p(K_x,w_i)xind}$

Add $xtag_i$ to List L5

$ORAM_f$.WRITE (f, $label_f$|'0')

Output L2,L3,L4,L5,$label_f$ at Server

Reshuffle L1 and store it at STORAGE

<u>Στον SERVER</u>

for j=1:|L2| do

p_w=L2[j]

Y_c=L3[j]

B=L4[j]

xtag=L5[j]

S1.insert (p_w , $label_f$|j|Y_c)

S2.insert ($label_f$, p_w| B | '1')

XSETA.insert ($label_f$, $xtag_i$)

Search Phase

The major change of that operation is that client does not send anymore the value xtrap at the Server but the blinded value of xtrap named bxtrap. The blinded factors pi are also send in an encrypted envelope at the server.

Client side

1. Sent the Query at Server

Dataowner Side

1. Choose the s-term keyword (w_s) from the query, the number of the rest w is "t"

2.(K_w,c,e,d,b,m) $\leftarrow$ $ORAM_w$.read(w_s)

$K_w' \leftarrow \{0,1\}^{\lambda}$, c'=0 , b'=0 , e'=0

$ORAM_w$.write(w , K_w' | c' | e' | d | b' m)

3. $K_{ws}^{(1)} = H(K|1|w_s)$

$K_{ws}^{(2)} = H(K|2|w_s)$

4. For y=1:t do

Choose a $p_y \in Z_p$

$xtrap_y \leftarrow g^{Fp(K_x, w_y)}$

$bxtrap_y \leftarrow (xtrap_y)^{p_y}$

Add $bxtrap_y$ to List L10

5. $env = Enc\left(K_e, p_1, p_2, ... p_y\right)$

6. Output: w_s, $K_{ws}^{(1)}$, $K_{ws}^{(2)}$,K_w , t , d, c, e, b, L10

Client Side

Client sends at Server : w_s, $K_{ws}^{(1)}$, $K_{ws}^{(2)}$,K_w , t , d, c, e, b, L10

Server Side

1. q=d-c

2. for i=1:q do

$(label_{fi}, Y_{ci}) \leftarrow$ S0.get(w)

$I_w \leftarrow I_w$ U $label_{fi}$

Add Yc to L8

3. n=0

for n=1:b do

n++

$p_{wn} \leftarrow H\left(K_w^{(1)}, n\right)$

$block.\lambda \,|\, block.\lambda'' \leftarrow S_1.remove(p_{wn})$

Y_c=block.λ"

Add Yc to L8

$a \,|\, b \leftarrow block.\lambda \oplus H(K_w^{(2)} \,|\, block.r)$

$label_{fn}$=a

$I_w \leftarrow I_w$ U $label_{fn}$

j $\leftarrow$ S0.insert (w , $label_{fn}$ | j | Y_c)

p_f=b

$S_2.update(p_{fn}, w_n \mid j \mid' 0')$

4. n=0

for n=1:(e-b) do

n++

p_w=H(K_w|e)

if p_w is found in S1 then

a|b|c $\leftarrow$ S1.remove(p_w)

$Y_c = c$

Add Yc to L8

$label_f$=a

$I_w \leftarrow I_w$ U $label_f$

j $\leftarrow$S0.insert (w_n , $label_{fn}$ | j | Yc)

S2.update ($label_f$/b , w_n | j | '0')

5. x=|L8|

y=t

Create an empty table(x,y)

6. Sent to Client : L8 , I_w

<u>Client side</u>

1. for x=1:|I_w| do

Y_{cx}=L8[x]

$label_{fx}$=I_w[x]

xind=F_p(K_I,$label_{fx}$)

Z_{cx}=xind Y_{cx}^{-1}

for y=1:t do

$$xtag_i = g^{F_p(K_x, w_i)xind}$$

$$token(x, y) = (xtrap_y)^{Z_{cx}}$$

add token(x,y) to L9

2. Client sends to Server L9 and evn=encryption.(p_y) for all y

Server Side

1. Server Stores tokens(x,y) at table(x,y)
2. I_{wfinal}=empty

3. For x=1:|L5| do
    for y=1:t do
        xtag=$(token(x,y))^{Ycx}$
        if xtag € XSETA($label_x$)
            THEN table(x,y)==TRUE
        elseif xteg € XSETB
            THEN table(x,y)==TRUE
        else table(x,y)==FALSE

4. For x=1:|L5| do
    if boolean function is TRUE
        then I_{wfinal}= I_{wfinal} U I_{wx}

Server outputs I_{wfinal}

Delete Phase

A major change of the Delete operation is that at the end of it, the Dataowner should be aware of the pairs that have been deleted in order to reduce the appropriate counters of the $ORAM_w$ structure for every keyword of the file.

Client (Dataowner) Side

1. $y|x|z \leftarrow ORAM_f.READ(f)$
2. $ORAM_f.DELETE(f)$
3. $label_f=y$

4. $K_f^{(1)} = H(K|1|f)$

   $K_f^{(2)} = H(K|2|f)$

5. Dataowner sends to Server: $K_f^{(1)}$, $K_f^{(2)}$, x, y, z

6. $Storage.Delete(label_f)$

Server Side

if z==1 then

for i=1:x do

$P_f = H(K_f^1|i)$

$a|b|c \leftarrow S_2.read(P_f)$

$S_2.delete(P_f)$

$XSETB.delete(p_f)$

if c==0 do

S0.remove(a,b)

Add a to List L6

max ← count.S0(a)

a | b''' | c''' ← READ.S0(a,max)

Remove.S0(a,max)

Write.S0(a/b , a | b | c''')

else do

$a' | b \leftarrow a \oplus H(K_f^{(2)} | block.r)$

Add b to List L7

S_1 .remove(a')

if z==0 then

a | b | c ← S2.READ($label_f$)

XSETA.DELETE(labelf)

S2.DELETE($label_f$)

if c==0 do

S0.remove(a,b)

Add a to List L8

max ← count.S0(a)

a | b''' | c''' ← READ.S0(a,max)

Remove.S0(a,max)

Write.S0(a/b , a | b | c''')

else do

$w_i \leftarrow b \oplus H(K_f^{(2)} | r_2), r_2$

Add w_i to List L9

S1.remove(a)

Server Export L6,L7,L8,L9

Στον Dataowner

For every w€L6 do

(K_w,c,e,b,d,m)<--read.$ORAM_w$(w)

d--

$ORAM_w$.write (w&K_w,c,e,b,d,m)

For every w€L7 do

(K_w,c,e,b,d,m)<--read.$ORAM_w$(w)

c--

d--

b--

$ORAM_w$.write (w&K_w,c,e,b,d,m)

For every w€L8 do

(K_w,c,e,b,d,m)<--read.$ORAM_w$(w)

d--

$ORAM_w$.write (w&K_w,c,e,b,d,m)

For every w€L9 do

(K_w,c,b,d,m)<--read.$ORAM_w$(w)

c--

d--

$ORAM_w$.write (w&K_w,c,e,b,d,m)

11.3 Leakage

Search leakage to Server

- Search Pattern (hash of keyword s-term , document identifiers that have been added or deleted for s-term keyword)
- Access Pattern 1 (matching document identifiers of the s-term keyword search)
- Access Pattern 2 (only the matching document identifiers of the x-term keywords that fulfill the boolean query)
- Time that the same keyword was accessed in the past
- Access Pattern at that time in the past
- Boolean Query Expression (AND-OR-NOT)

Search leakage to Dataowner

- Boolean expression
- s-term and x-term keywords

Search leakage to Client

- Matching document identifiers of the s-term keyword search, even if they don't fulfill the Boolean query expression.

Chapter 12

OSPIR - Dataowner with Limited Knowledge

12.1 Introduction

This chapter is about a special case of the Multi-client scheme, where the Dataowner who authorizes the Boolean query is not aware of the keywords of the query. The Dataowner is only allowed to learn the Attributes that each keyword of the query belongs to. The attribute will be denoted as ω_n or $I(w_n)$, and the set of attribute for a query as $av(\bar{\omega})$. This scheme will be referred to as OSPIR-MC.

The OSPIR - MC scheme can be implemented with two different scenarios.
1) keyword of the query and attribute of the query are two different values.
2) keyword w contains both value and attribute w=(value, attribute)
The second approach will be used for the rest of this thesis.

More specifically, the Dataowner will now learn only the value:

$$av(\bar{\omega}) = (I(w_1), ...I(w_n))$$

and will check if the $av(\bar{\omega})$ satisfies the Policy that the Dataowner has for that specific client that makes the query.

The main differences between the MC protocol and the OSPIR-MC protocol are presented bellow. The OSPIR-MC protocol uses an "oblivious PRF" (OPRF) computation between the Client and the Dataowner. A PRF F(K, w) is called oblivious when there is a two-party protocol in which Client inputs w, Dataowner inputs K, Client learns the value of F(K, w) and Dataowner learns nothing.

An example is the Hashed DH OPRF which is used in the implementation of the OSPIR-MC protocol, defined as $F(K,x) = H(x)^K$. The OPRF protocol consists of Client sending $a = H(x)^r$ for random r in Z_p^*, Dataowner sending back $b = a^K$ and finally Client computing $H(x)^K$ as $b^{1/r}$.

For security it is needed that only queries on authorized attributes can generate valid tokens for search at E. To enforce that Dataowner must use a different key for each possible attribute.

A major change is the replacement of the PRF F_p used in computing xtrap and xtag values with a PRF F_G, which maps w directly onto the group G generated by g. So, xtrap is set as $F_G(K_X,w)$ instead of $g^{F_p(K_X,w)}$ and from now xtag = $(xtrap)^{xind}$ will be computed as $F_G(K_X,w)^{xind}$ instead of $g^{F_p(K_X,w)xind}$.

The first step of the Search operation would be the Client to send at Dataowner the $av(\bar{\omega})$, so the D could authorize or not the Boolean query. Moreover Client will send for the s-term keyword the value a_s and for the rest of the keywords (x-term) the values a_i. $a_i=H(w_i)^{ri}$ where r_i is a random value that is known only by the Client.

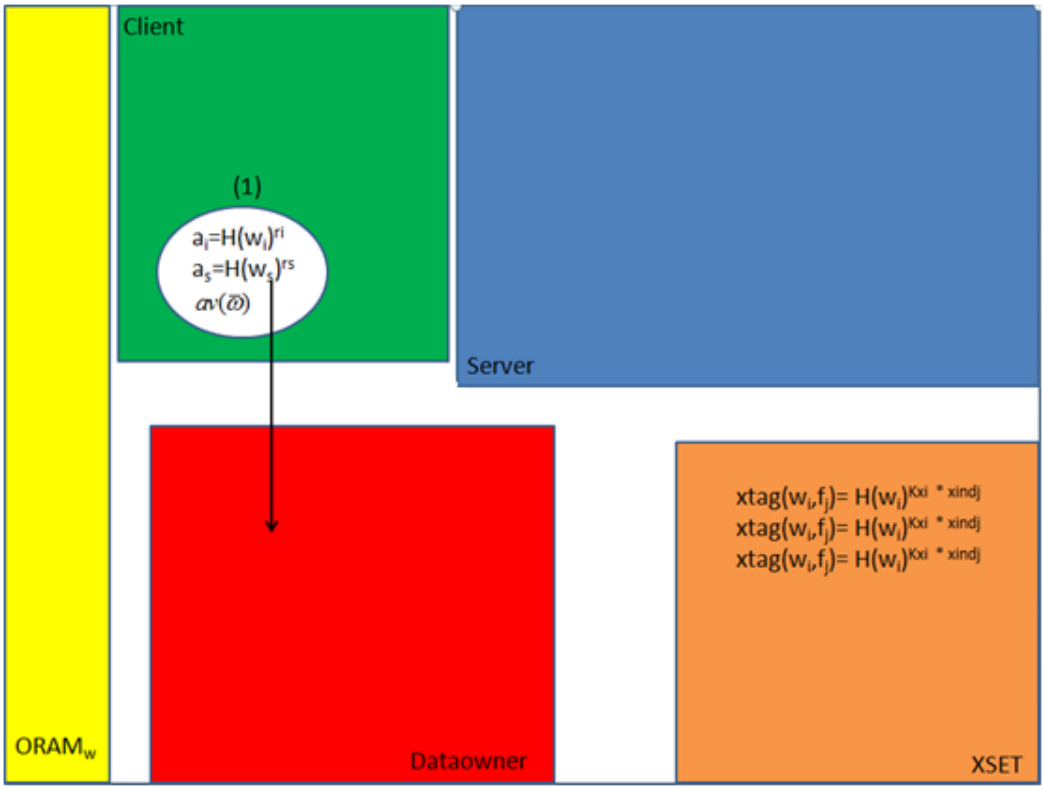


Figure 12.1 - Information sent during the first step of the query

At the second step, the Dataowner will check if the $av(\bar{\omega})$ satisfies the Policy P for that client and if it does, then he will raise the quantity a_s at a product of two values. The key K_{xs} which is the key of the attribute that s-term keyword belongs to, and ρ1 which is a random blinded factor that is known only at Dataowner and the Server. The same blinded procedure if also followed for the rest of the keywords of the Boolean search, in order to produce the values b_i.

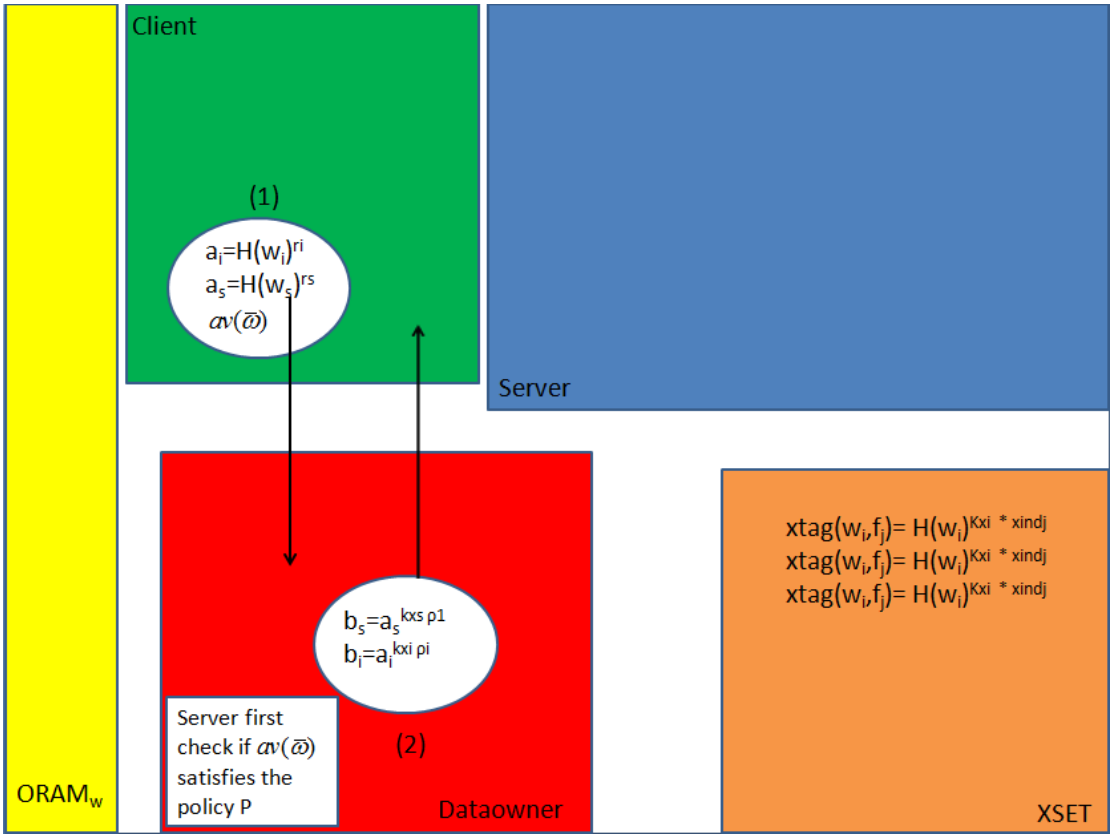


Figure 12.2 - Information sent during the second step of the query

The third step is the query that the client starts to the structure $ORAM_w$ in order to retrieve all the needed information about the s-term keyword.

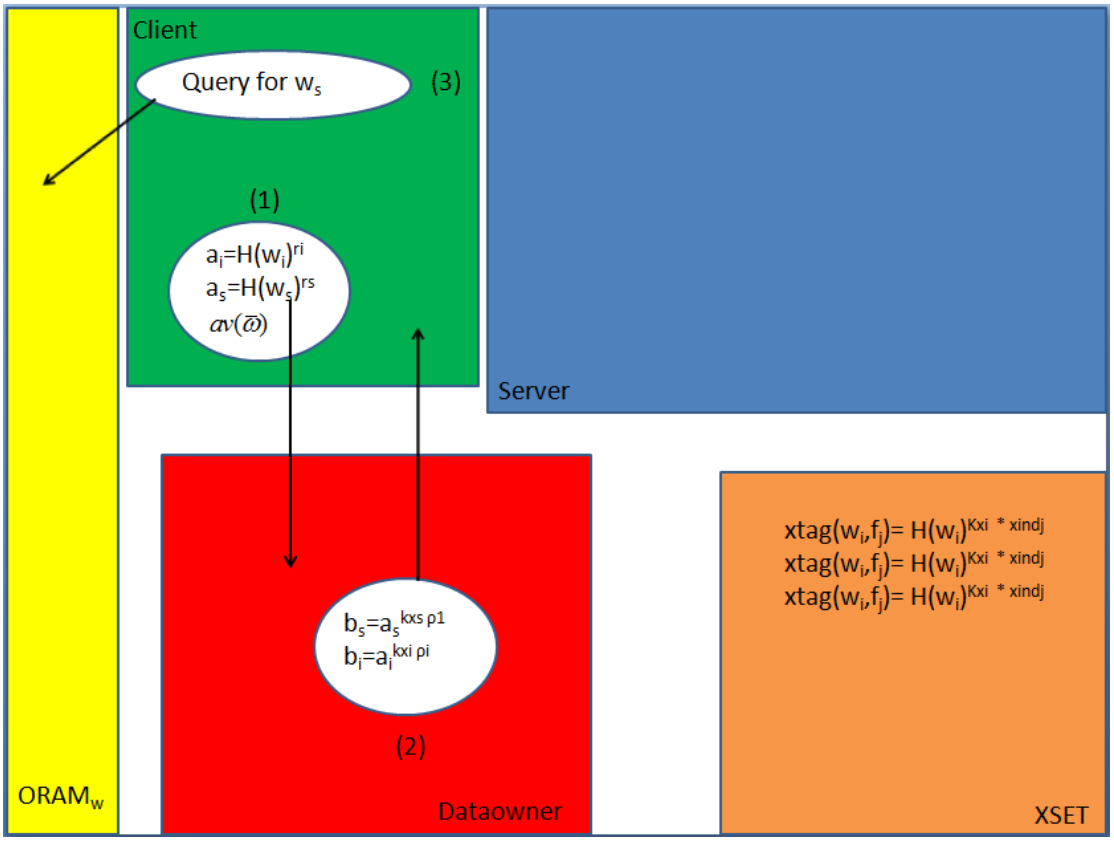


Figure 12.3 - Information sent during the third step of the query

For the fourth step of the search operation the client un-blinds the value of b_s that received from the Dataowner by raising it at $1/\rho_s$. Moreover the Client sends at Server:

- The previous un-blinded value
- The attribute of the s-term keyword
- The s-term keyword itself
- An encrypted envelope that contains all the random values pi that the Dataowner used in order to blind the queries
- Both values $K_{ws}^{(1)}$ and $K_{ws}^{(2)}$ that computed based on the information that retrieved from the $ORAM_w$
- The keyword key K_w

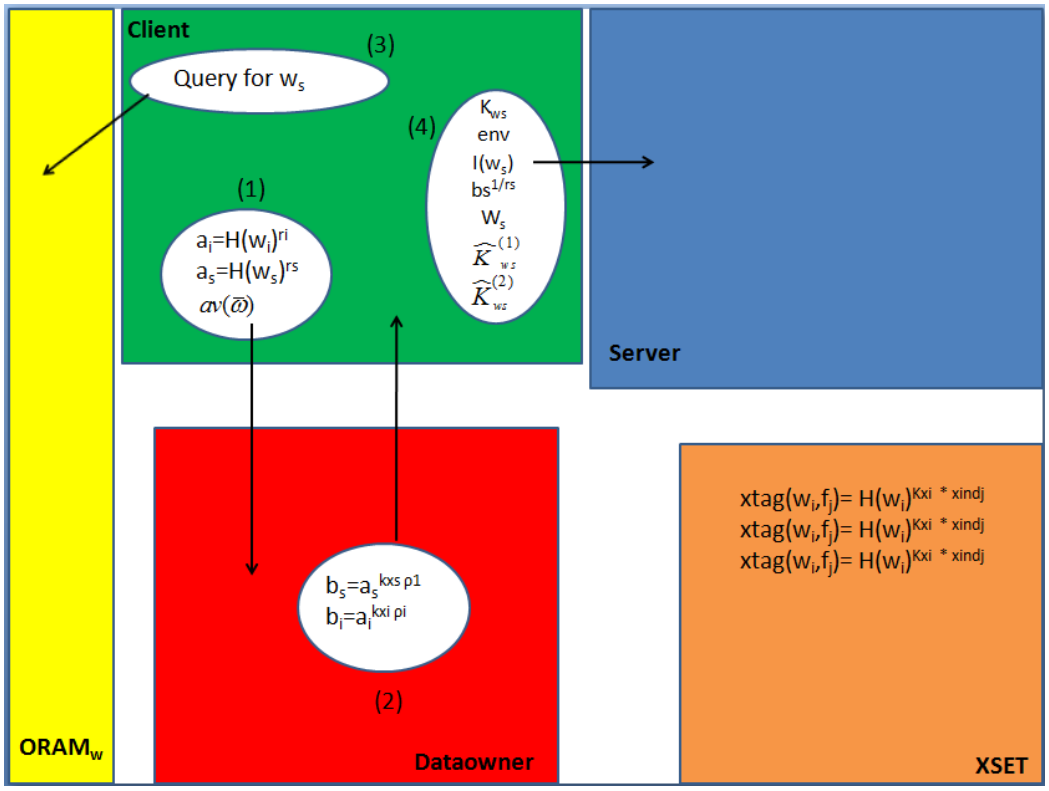


Figure 12.4 - Information sent during the fourth step of the query

Based on received values the Server firstly check if the Client is authorized for that query.

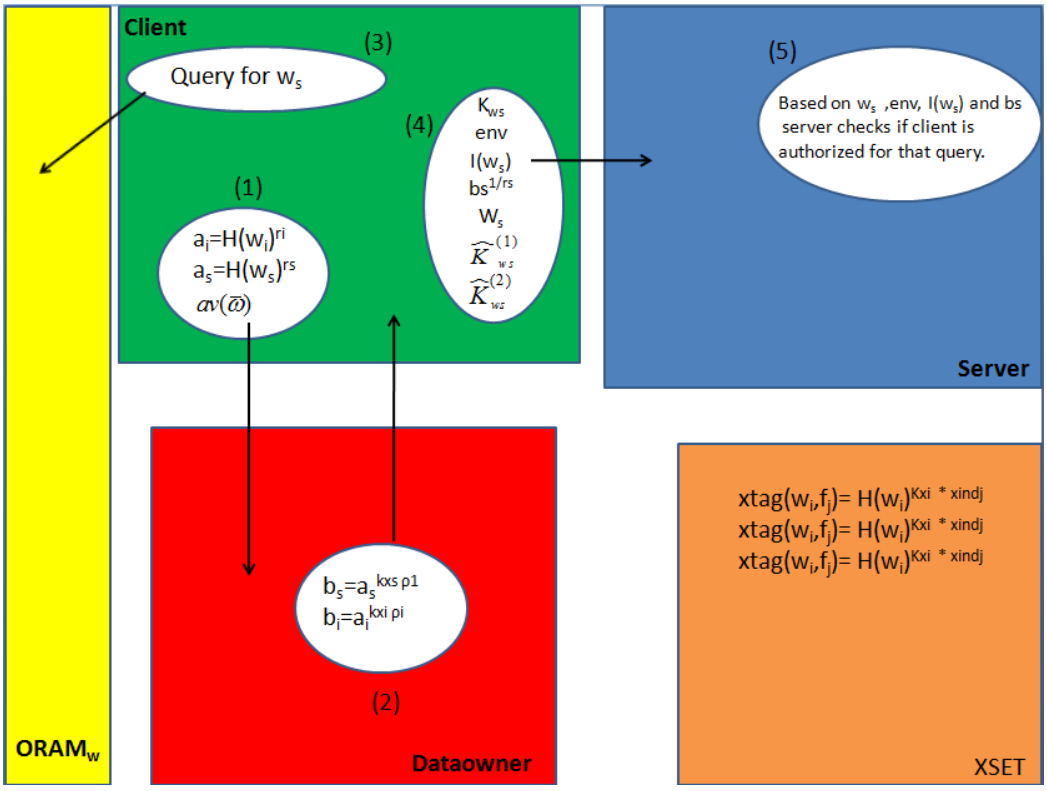


Figure 12.5 - Information sent during the fifth step of the query

As a sixth step, if the Client is authorized, the Server follow the protocol and retrieves all the file identifiers of the s-term keyword along with the corresponding values of Y_c, for every pair that matched the search.

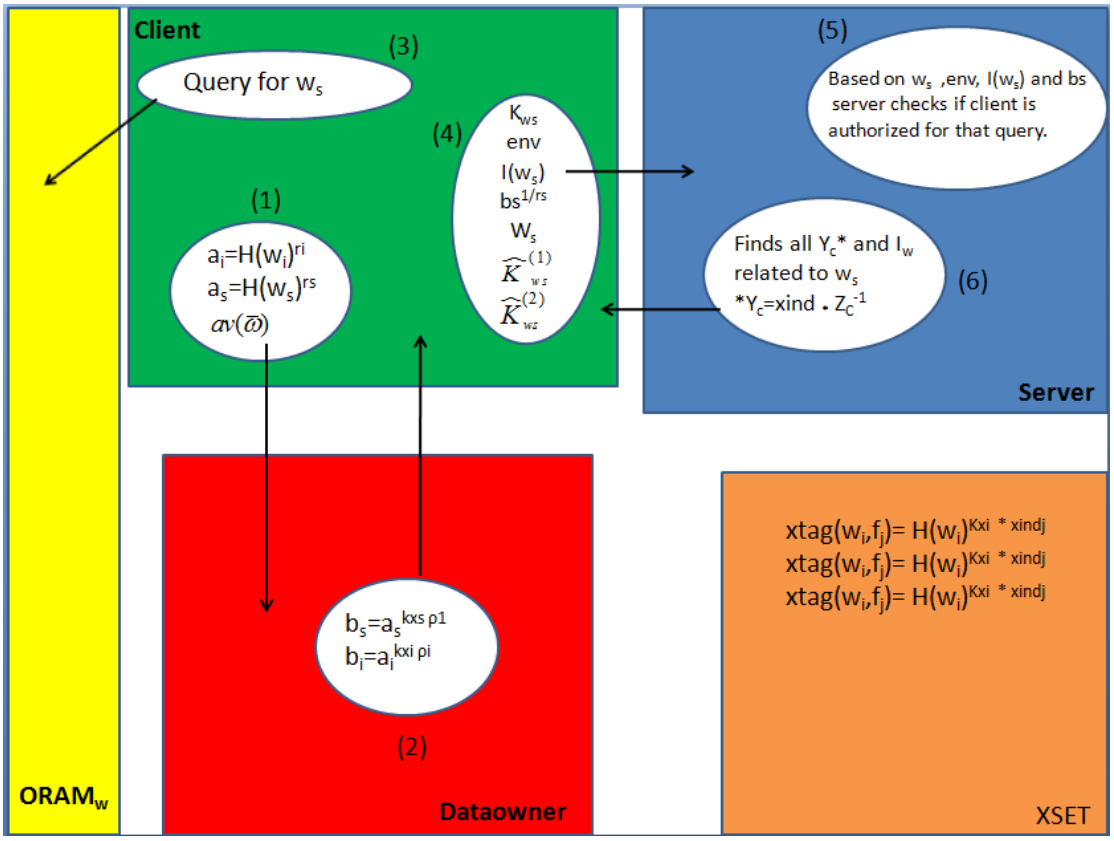


Figure 12.6 - Information sent during the sixth step of the query

Based on the I_w, the client can now calculate the value xind of every file that belongs to I_w and by also using the Y_c values, the Client can calculate all the needed Z_c. Afterwards Client prepares the tokens that should send at Server by un-blinding every bi and raising it at the corresponding Z_c.

If the Client try to cheat and send another query by using a keyword from different attribute, then the search will fail because every xtag value has a specific key based on the attribute of the keyword that is used at that xtag.

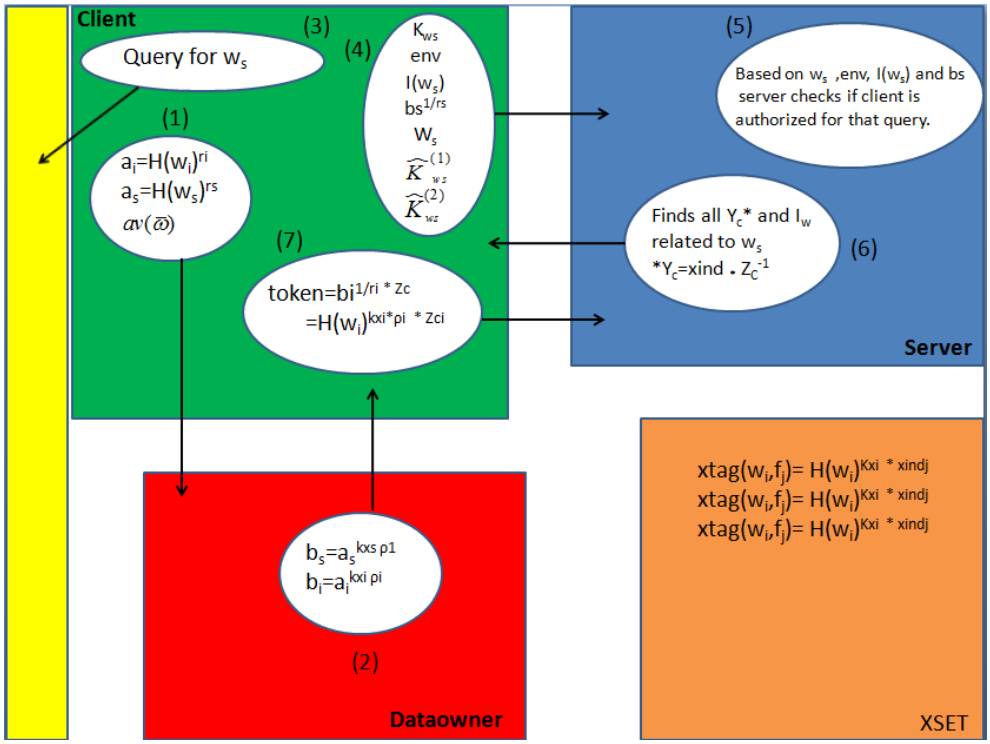


Figure 12.7 - Information sent during the seventh step of the query

Finally the Server, by combining the tokens, the blinding values pi (that receive encrypted by the Dataowner) and the values Y_c, can compute the xtag values and search for their existence at both structures XSETA and XSETB.

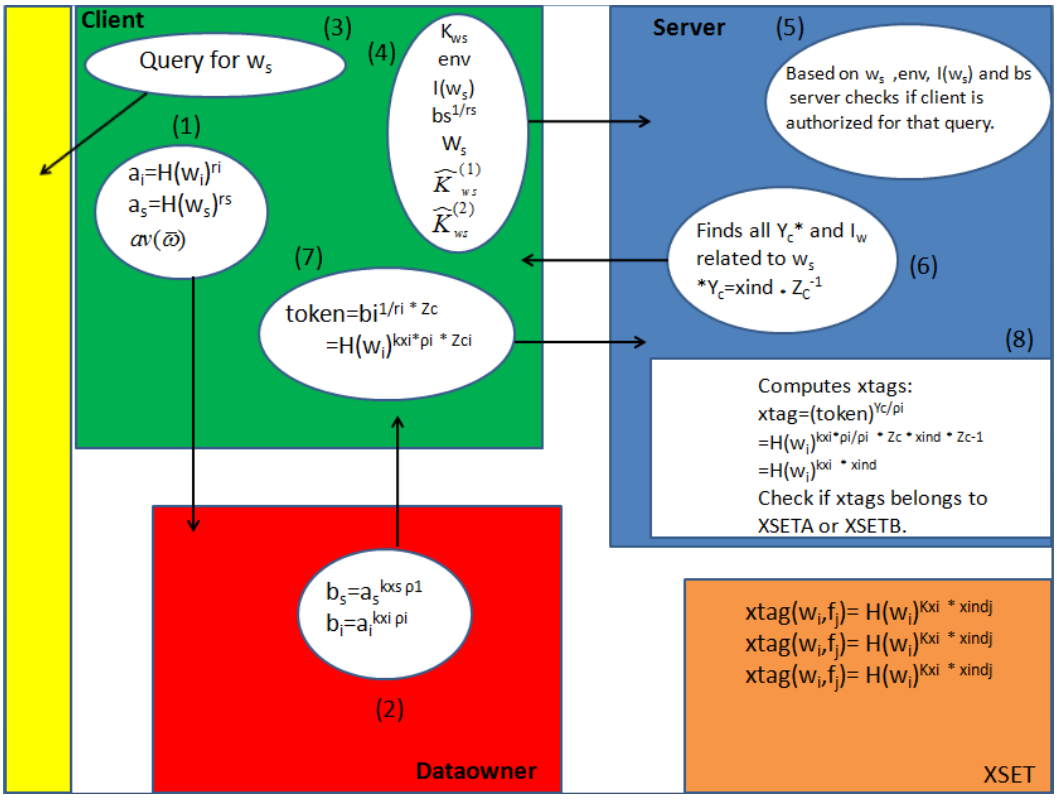


Figure 12.8 - Information sent during the eight step of the query

12.2 Algorithms

OSPIR-MC differs only at the search algorithm, which is presented bellow. The idea on which the search operation is based on, was presented on the previous chapter. Add, Delete and Setup operations remains the same as in simple MC scheme.

Search

Client side

1. Sent the Query at Server
2. $w_{query} = (w_1,....,w_n)$ where w_1 is chosen as s-term:
   $(a_s,r_s) \leftarrow OPRF.C_1(w_1)$
   for each i = 1,.....,n do
      $(a_i,r_i) \leftarrow$ S-OPRF.$C_1(w_i)$
3. Output at Dataowner: $(a_s, a_1,....,a_n)$ and av = $(I(w_1),..., I(w_n))$.

Dataowner Side

1. Data owner D, on input policy P and master key
   $K = (K_S,K_X,K_T ,K_I ,K_P ,K_M)$:
   if av is not in policy set P
      then abort.
      else set av as D's local output.

2. $(b_s,\rho_1) \leftarrow$ S-OPRF.$D(K_x , I_s, a_s)$,

3. for i = 2,.....,n do
      Choose a $p_i \in Z_p$
      $(b_i,\rho_i) \leftarrow$S-OPRF.$D(K_X, I_i, a_i)$.

4. env$\leftarrow$AuthEnc$(K_M, (\rho_1, \rho_2,.....,\rho_n))$

5. Send $(env, av, b_s,b'_s,b_2,.....,b_n)$ at Client.

Client Side

1.$(K_w,c,e,d,b,m) \leftarrow ORAM_w.read(w_s)$

$K_w' \leftarrow \{0,1\}^{\lambda}$, c'=0 , b'=0 , e'=0

$ORAM_w.write(w , K_w' | c' | e' | d | b' | m)$

2. $K_{ws}^{(1)} = H(K|1|w_s)$

$K_{ws}^{(2)} = H(K|2|w_s)$

3. Client sends Server $b_s,w_s,I(w_s),b,e,c,d$

Server Side

1. ρ_s=env[1]
2. $K_x=K_x(I(w_s))$
3. as'= $H(w_s)^{\rho s\ Kx}$
4. if as'=bs then continue
5. q=d-c

6. for i=1:q do

$(label_{fi},Y_{ci}) \leftarrow$ S0.get(w)

$I_w \leftarrow I_w$ U $label_{fi}$

Add Yc to L8

7. n=0

for n=1:b do

n++

$p_{wn} \leftarrow H\left(K_w^{(1)},n\right)$

$block.\lambda \,|\, block.\lambda'' \leftarrow S_1.remove(p_{wn})$

Y_c=block.λ"

Add Yc to L8

$a \mid b \leftarrow block.\lambda \oplus H(K_w^{(2)} \mid block.r)$

$label_{fn}$=a

$I_w \leftarrow I_w$ U $label_{fn}$

j $\leftarrow$ S0.insert (w , $label_{fn}$ | j | Y_c)

p_f=b

$S_2.update(p_{fn}, w_n \mid j \mid' 0')$

8. n=0

for n=1:(e-b) do

n++

p_w=H(K_w|e)

if p_w is found in S1 then

a|b|c $\leftarrow$ S1.remove(p_w)

Y_c = c

Add Yc to L8

$label_f$=a

$I_w \leftarrow I_w$ U $label_f$

j $\leftarrow$S0.insert (w_n , $label_{fn}$ | j | Yc)

S2.update ($label_f$/b , w_n | j | '0')

9. x=|L8|

y=t

Create an empty table(x,y)

10. Sent to Client : L8 , I_w

Client side

1. for x=1:$|I_w|$ do

Y_{cx}=L8[x]

$label_{fx}=I_w[x]$

$xind=F_p(K_I,label_{fx})$

$Z_{cx}=xind\ Y_{cx}^{-1}$

for y=1:t do

$bxtrap_i \leftarrow$ S-OPRF.C2(b_i,r_i)

$bxtrap_i= H(w_i)^{ri\ kx\ pi/ri} = H(w_i)^{kx\ pi}$

$token(x,y)=(bxtrap_y)^{Zcx}$

add token(x,y) to L9

2. Client sends to Server L9 and evn=encryption.(p_y) for all y and tokens(x,y)

Server Side

1. Server Stores tokens(x,y) at table(x,y)

2. I_{wfinal}=empty

3. For x=1:|L5| do

for y=1:t do

$xtag=(token(x,y))^{Ycx}$

if xtag € XSETA($label_x$)

THEN table(x,y)==TRUE

elseif xteg € XSETB

THEN table(x,y)==TRUE

else table(x,y)==FALSE

4. For x=1:|L5| do

if boolean function is TRUE

then $I_{wfinal} = I_{wfinal} \cup I_{wx}$

Server outputs I_{wfinal}

12.3 Leakage

Search leakage to Server

- Search Pattern (hash of keyword s-term , document identifiers that have been added or deleted for s-term keyword)
- Access Pattern 1 (matching document identifiers of the s-term keyword search)
- Access Pattern 2 (only the matching document identifiers of the x-term keywords that fulfill the boolean query)
- Time that the same keyword was accessed in the past
- Access Pattern at that time in the past
- Boolean Query Expression (AND-OR-NOT)

Search leakage to Dataowner

- Boolean expression
- Attributes of s-term and x-term keywords

Search leakage to Client

- Matching document identifiers of the s-term keyword search, even if they don't fulfill the boolean query expression.

Chapter 13

Proposal for improvements

13.1 Oblivious Transfer

An extra leakage that inserted at the scheme presented in chapter 10 is the ability of the Client to query the structure $ORAM_w$. This was mandatory because the Dataowner should not learn the exact keyword of the search but only its attribute. So, the Dataowner could not ask the $ORAM_w$ on behalf of the Client.

A solution in order to avoid the extra leakage would be to avoid the use of the ORAMw structure and replace it with a $Table_w$ structure, which will be hosted by the Dataowner. This approach supposes that Dataowner has the disk space needed for that hosting. Then, by using a principle named Oblivious Transfer, the Client will be able to retrieve the information that he needs from $Table_w$ about the s-term keyword, but on the other hand the Dataowner will not be able to understand which keyword client asked for. The principle of the Oblivious Transfer seems to be impossible and so should be explained with more details in order to clarify it.

Oblivious Transfer implementations can be based on most public-key systems. There are also implementations with two rounds of communication. Oblivious transfer was introduced by Rabin, and 1-out-of-2 OT by Even Goldreich and Lempel. The fundamental idea is that Sender has two values, Y_0 and Y_1. A chooser (Alice) can ask for one of these values from Sender and receive it. Sender never learns which value the chooser received. This is called 1-out-of-2 Oblivious Transfer.

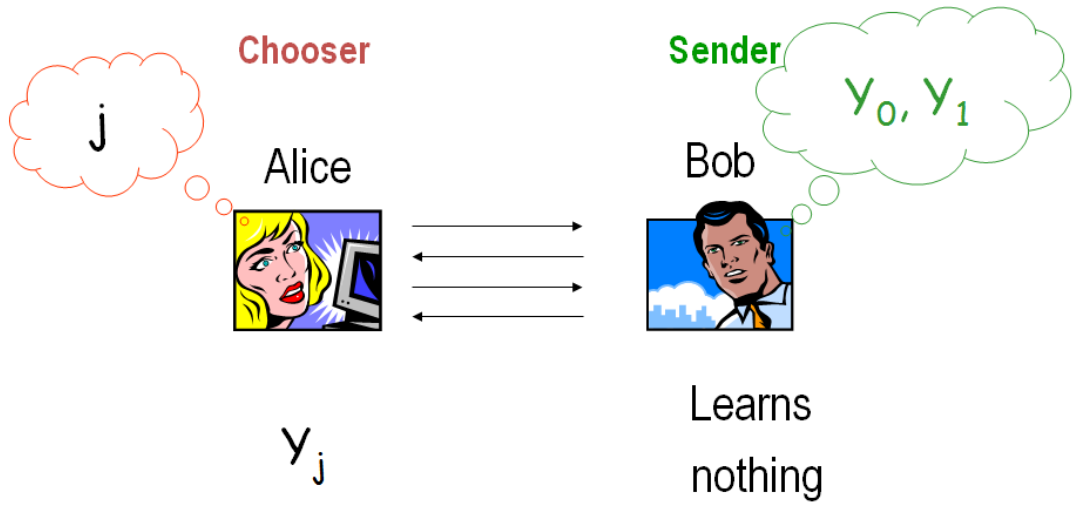


Figure 13.1 - 1-out-of-2 Oblivious Transfer scenario

This scenario can also be generalized for 1-out-of-N Oblivious Transfer, where the chooser can choose between N values and not only 2.

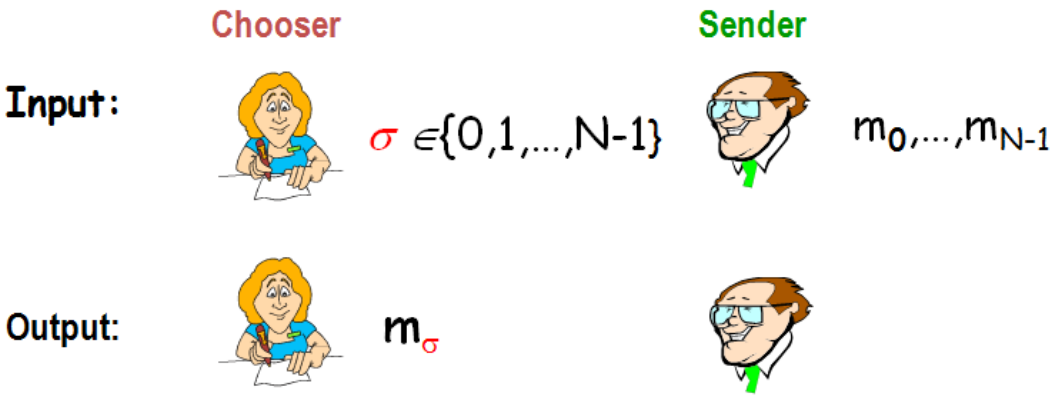


Figure 13.2 - 1-out-of-N Oblivious Transfer scenario

Both parties learn nothing else:

- Indistinguishable to Sender which σ is used
- Chooser learns no other value of $m_0,\ldots,m_{N-1}$

It is interesting to examine the basic implementation of OT, in order to compare its efficiency. In the basic protocol (suggested by Even, Goldreich and Lempel) the chooser sends two public keys to the sender, together with a proof that she knows the private key of only one of them. She should choose to make sure that she knows the private key of PKσ. The sender encrypts each input with the corresponding public key. The chooser can only decrypt m_σ.

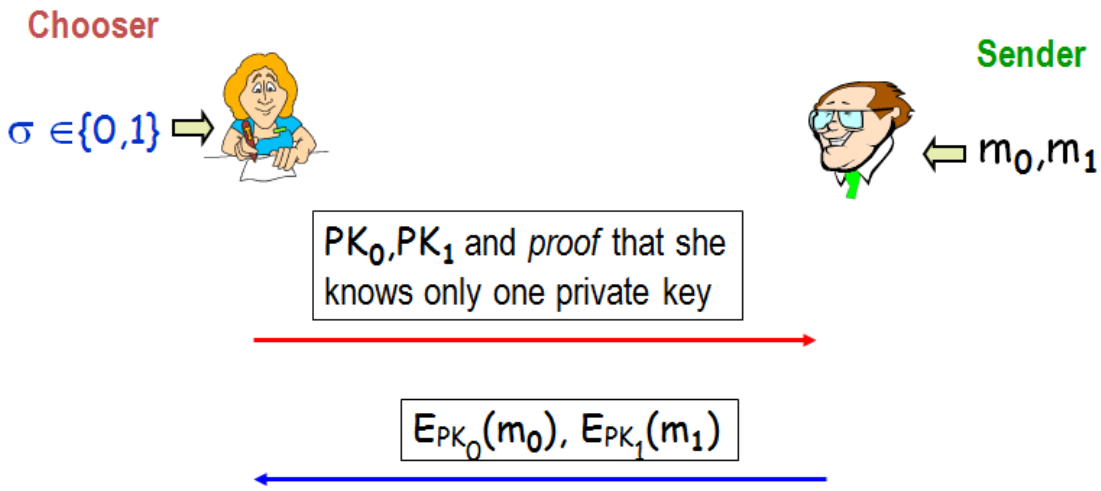


Figure 13.3 - Even, Goldreich and Lempel Proposal

A second approach of the 1-out-of-2 OT is the Bellare-Micali Protocol which is described by the following figure:

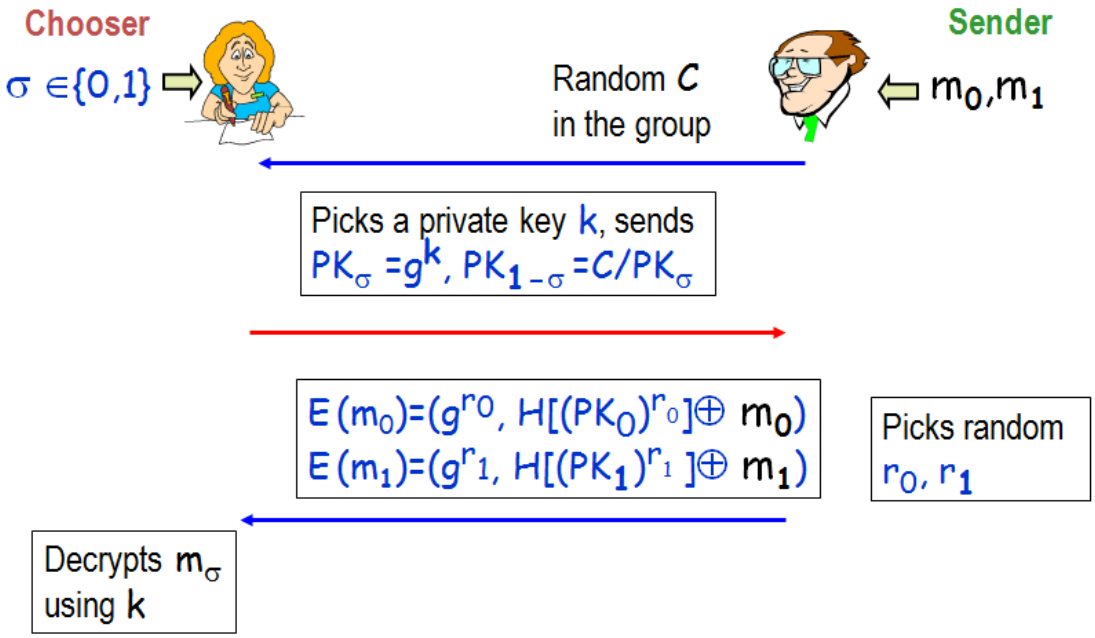


Figure 13.4 - Bellare-Micali Proposal

At this protocol the Sender pick a completely random number C and sends it to Chooser. Chooser picks a private key K and sends at Sender two private keys, so that $PK_{1-\sigma} = C / PK_{\sigma}$. Sender now can be sure that Chooser knows only 1 private key. Sender then selects two random values r_0 and r_1 and sends at Chooser the $E(m_0)$ and $E(m_1)$. Finally the Chooser can decrypt the message that he has chosen (m_σ), by using the private key k.

After having understand the Oblivious Transfer protocol, it should be mentioned that by adding it to our scheme, the computation cost of the scheme will be increased a lot. It is another tradeoff between leakage and efficiency that should be chosen, depending on the needs of the scenario that will use the DSSE scheme.

13.2 Reduce Client-Server Interaction

In the schemes presented in the previous chapters there were two rounds of interaction between the Client and the Server during the search operation. Initially, the Client sends at the Server the s-term keyword along with other needed information. Then the Server returns the results of the Search which are the set of file identifiers (I_w) and list of Y_c of all matched pairs. Finally, the Client based on the I_w and Y_c can calculate the values of Z_c and then the tokens of the query that sends at the Server in order to search them in both XSET structures.

Then main reason for the double interaction is that the Client must calculate somehow the values of Zc and for that he needs the values of Yc.

$$Z_c = \text{xind } Y_c^{-1}$$

On the other hand Zc can also be calculated based on:

$$Z_c = \text{PRF}(w,m))$$

The Client knows from the beginning the x-term keywords, but he cannot use the above formula because with the current algorithm, the Client is not aware of the value m that was used in the calculation of each Z_c. So Client and Server would not be able to agree on which Z_c will be used to unblind each Y_c. It should be mentioned once more, that the counter m is only increased for each file that contains the same keyword and it never decrease in case of deletions. So, there is not a pattern in order to know the correct value of m, because the pair corresponding to a value might have been deleted.

In order to solve that issue, it is needed to store at structures S1 and S_1 the counter m that was used in order to compute the corresponding value of Y_c that is stored on the same label of the structure. In that way, the Client will create from the beginning all the possible values of Z_c for all the m values and will also create all the possible tokens based on the Z_c. On the other hand, the Server will know from the

first step of the search, which m values are still valid and will only use those Z_c values, received from the Client, in order to unblind the correct Y_c values.

Having achieved that, it is no more needed for Client to know the values of Yc from the Server and so the Client even from the first round of interaction can send at Server the w_s and the tokens for the x-term keywords of the search.

This approach increase a little bit the amount of time that is needed for the token creation, because it probably creates more tokens than the correct amount, but in the other hand, it decreases the rounds of communication between Client and Server, which might be vital for communications with high latency.

Most previous schemes that claim to have achieved DSSE schemes with support of Boolean queries, are using revocation lists for the deletion operation and so they don't even mention the way that they use in order to remove the deleted xtag values from the XSET or how they match the correct Z_c value with the corresponding Y_c.

Chapter 14

Conclusion

As a conclusion, it should be mentioned that the structure of this thesis was based on the row of research followed, during the study on the subject of Dynamic Searchable Symmetric Encryption.

Initially, the basic information needed in order to understand the principles of Searchable Encryption was summarized and even more thoroughly the principles of Dynamic Searchable Symmetric Encryption. Two of the surveys on DSSE that were studied were briefly presented as standing out among the rest, by using a different approach compared to the rest of the proposals for that time.

One of the goal of this thesis was to propose a configuration change, for the survey presented in chapter seven, in order to achieve better forward privacy characteristics, when a new file is added. The second goal, was to add more features on the scheme presented in chapter eight. These add-ons, such as capability for Boolean queries and Multi-Client support, made the scheme even more practical for real world scenarios.

These proposals were analyzed and the algorithms for those add-on features were presented. Eventually, a modified scheme was presented, in which the Dataowner is not allowed to learn the keywords of the Boolean queries that Clients send to the Cloud Server. Of course more improvements can be implemented to that scheme and two of them were explained in detail in the last chapter of this thesis.

Chapter 15

References

[1] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Highly scalable searchable symmetric encryption with support for Boolean queries". In Advances in Cryptology, CRYPTO, LNCS, vol. 8042. Springer Berlin Heidelberg, pp. 353-373, 2013.

[2] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawcyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation". In 21th Annual Network and Distributed System Security Symposium NDSS. The Internet Society, February 23-26, 2014.

[3] E. J. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/ 216/.

[4] S. Kamara, C. Papamanthou, and T. Roeder. "Dynamic searchable symmetric encryption". In the ACM Conference on Computer and Communications Security, CCS, pp. 965-976, 2012.

[5] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption". In Financial Cryptography (FC), 2013.

[6] E. Stefanov, C. Papamanthou, and E. Shi, ``Practical dynamic searchable encryption with small leakage," in *Proc. NDSS*, Feb. 2014.

[7] M. Naveed, M. Prabhakaran, and C. A. Gunter, ``Dynamic searchable encryption via blind storage," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 639-654.

[8] P. Rizomyliotis, "An ORAM based forward privacy preserving Dynamic Symmetric Searchable Encryption Scheme", June. 2015.

[9] P. Rizomyliotis, "ORAM based forward privacy preserving Dynamic Searchable Symmetric Encryption Schemes". October. 2015.

[10] 6. F. Hahn, and F. Kerschbaum. "Searchable Encryption with Secure and Efficient Updates". In ACM Conference on Computer and Communications Security, CCS14, Scottsdale, Arizona, USA, 2014.

[11]F. Bao, R. Deng, X. Ding and Y.Yang. "Private Query on Encrypted Data in Multi-User Settings". 2015.

[12] S. Jarecki, C. Jutla, H. Krawczyk, M. C. Rosu, and M. Steiner. "Outsourced symmetric private information retrieval". In ACM CCS 13, Berlin, Germany, Nov. 4–8, 2013. ACM Press.

[13] T. C hou and C. Orlandi. "The Simplest Protocol for Oblivious Transfer". 2015.

[14] Seny Kamara. "Encrypted Search" . Microsoft Research. 2014.

[15] R Curtmola, J. Garay, S. Kamara, R. Ostrovsky. " Searchable symmetric encryption: improved definitions and efficient constructions". Proceedings of the 13th ACM conference on Computer and communications security Pages 79-88. 2006 .