cosmian

Findex

Full Documentation

Version 1.0

Date: 29/06/2022

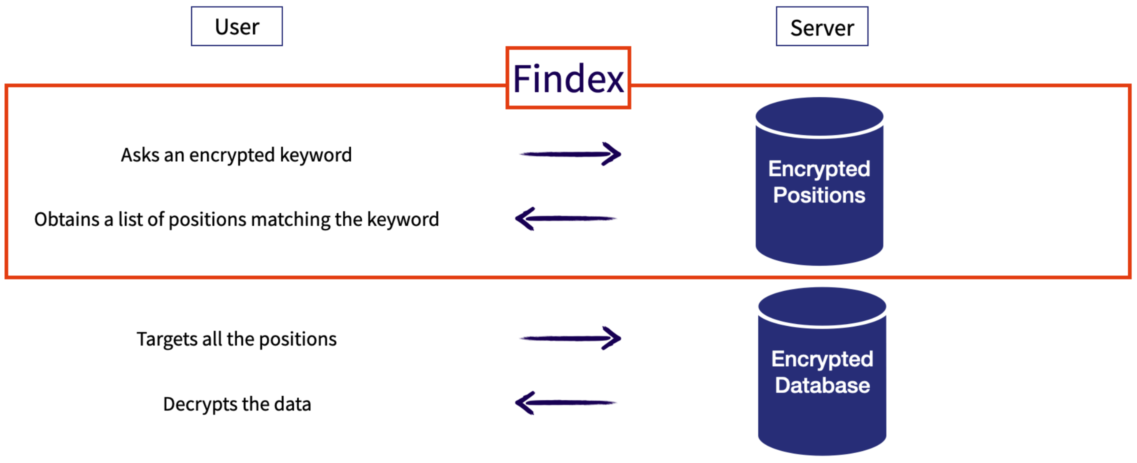
Author: Chloé Hébant, Cryptographer

Table Of Contents

1 Introduction

Findex is a part of Cloudproof Encryption and helps to securely make search queries on outsourced encrypted data.

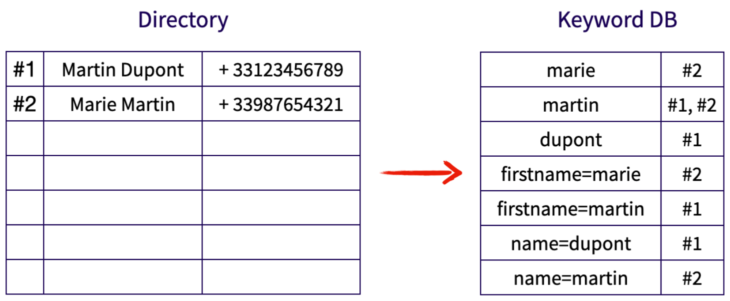
This documentation shows its running and explains the cryptographic details.


User
Server
Findex
Asks an encrypted keyword
Obtains a list of positions matching the keyword
Encrypted Positions
Targets all the positions
Decrypts the data
Encrypted Database

2 Example - The Searchable Directory

To understand the general idea behind Findex, let us assume one wants to outsource a directory while being able to securely make search queries on it.

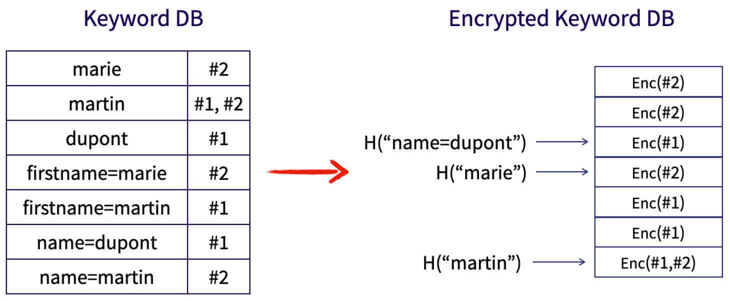
Here the directory is composed of two users. The first step consists of building a Keyword Database:



#1	Martin Dupont	+ 33123456789
#2	Marie Martin	+ 33987654321

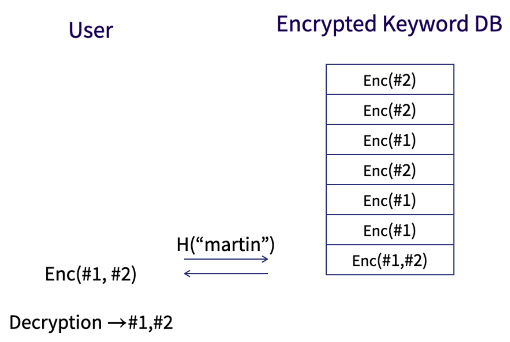
marie	#2
martin	#1, #2
dupont	#1
firstname=marie	#2
firstname=martin	#1
name=dupont	#1
name=martin	#2

Then, the table can be encrypted and sent by an authenticated administrator to a first server. The lines are placed at random to not be able to retrieve the keyword from the position in the encrypted database:

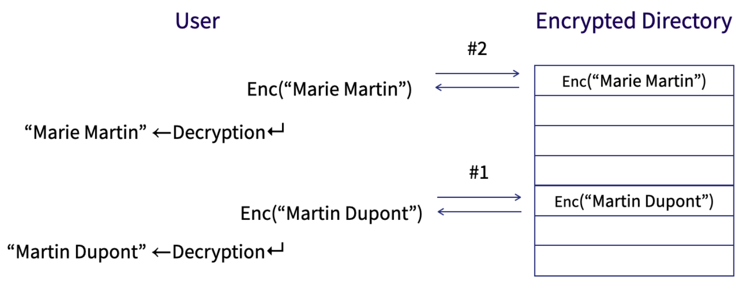


marie	#2
martin	#1, #2
dupont	#1
firstname=marie	#2
firstname=martin	#1
name=dupont	#1
name=martin	#2

Now, the encrypted keyword database exists, a user can build requests. The user hashes the keyword "Martin" and asks for it to the server having the encrypted keyword database:

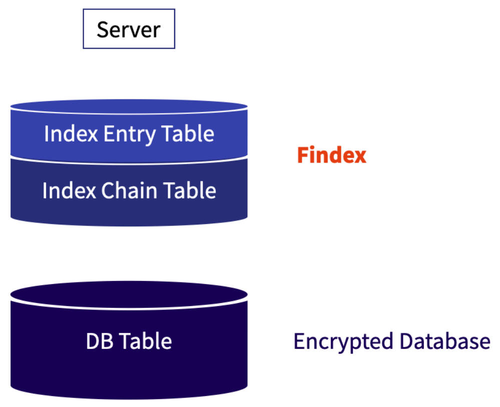


The user receives an encrypted message containing the position of all the matching queries: `#1,#2`. With these two positions, the user can interact with the second server having the encrypted directory:



3 Full Process

3.1 Overview



Findex relies on two server-side tables, *Index Entry Table* and *Index Chain Table*, to solve the following search problem:

> How to *securely* recover the UIDs of DB Table to obtain the matching lines from a given keyword?

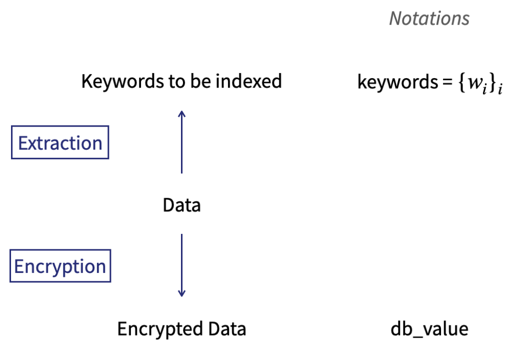
This solution is on top of an encrypted database, for consistency called DB Table, that actually stores the content to be requested.

3.1.1 Symmetric Searchable Encryption

To make efficient search queries on an untrusted cloud server, one needs to use an advanced cryptographic primitive called Symmetric Searchable Encryption (SSE). The security of SSE offers precise guarantees regarding the privacy of the user's data and queries with respect to the host server.

3.1.2 Notations

We assume that each line of DB Table is encrypted but at the time of encryption some keywords $\{w_i\}_i$ have been extracted to be stored with **Findex**. In one line many keywords can be extracted and a particular keyword can be present in several lines.



In this entire document, *key* refers to a cryptographic key and the databases are represented by a list of $(\text{uid}_i, \text{value}_i)$.

Hence,

- $(\text{db\_uid}_i, \text{db\_value}_i) = (\text{uid}_i, \text{value}_i)$ of DB Table,
- $(\text{iet\_uid}_i, \text{iet\_value}_i) = (\text{uid}_i, \text{value}_i)$ of Index Entry Table,
- $(\text{ict\_uid}_i, \text{ict\_value}_i) = (\text{uid}_i, \text{value}_i)$ of Index Chain Table.

3.1.3 Index Tables

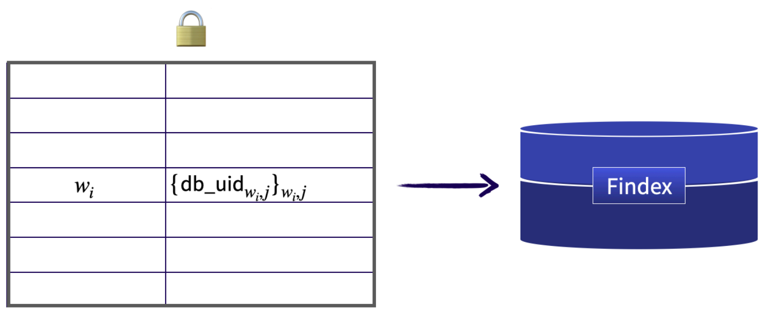


Figure 1: Findex Input

After the extraction, each keyword w_i can be associated to a list $L_{w_i} = \{\text{db\_uid}_{w_i,j}\}_{w_i,j}$ of the db_uids matching the keyword.

By abusing the notation, we will sometimes use i instead of w_i to improve readability and denote $\text{db\_uid}_{i,j} = \text{db\_uid}_{w_i,j}$, $L_i = L_{w_i}$ and $\{L_i\}_i = \{L_{w_i}\}_{w_i}$.

Index Chain Table: *securely* stores all the lists $L_i = \{\text{db\_uid}_{i,j}\}_{i,j}$.

Example: The keyword "Martin" is present in the lines 3, 5 and 10 of a non encrypted directory. These lines correspond to the db_uids: db_uid_a, db_uid_b, and db_uid_c of the DB Table (i.e. the encrypted directory). The Index Chain Table will securely store $\{\text{db\_uid}_a, \text{db\_uid}_b, \text{db\_uid}_c\}$.

Index Entry Table: provides the mandatory values to access the Index Chain Table.

3.1.4 Search Query

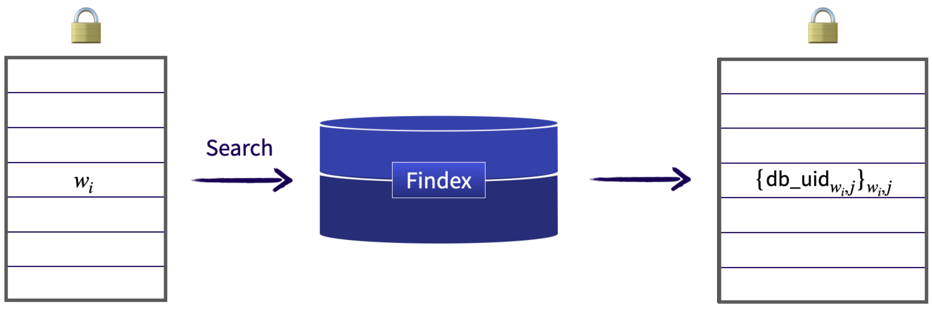


Figure 2: Search Query

Search Query: takes as input a bulk of encrypted keywords $\{w_i\}_i$ and outputs the bulk of the encrypted lists $\{L_i\}_i$.

Findex considers search queries restricted to a single keyword w. To handle queries with several keywords (OR of keywords), several requests are made to the server and possibly a combination can be done on the client's side to deal with ANDs.

3.2 Index Chain Table

Let us see the content of the Index Chain Table (ICT).

We denote by L_{w_i} the list of the UIDs of DB Table matching the keyword w_i and ℓ_{w_i} its length (the number of matching elements). To hide ℓ_{w_i}[1], L_{w_i} will be divided in parts of equal size $L_{w_i} = \{L_{w_i,1}$, $L_{w_i,2}$, …, $L_{w_i,x_{w_i}}\}$ (potentially, the last “block” is not full).

Note: The size t of a block is a common parameter for all the keywords that depends on the possibly searchable keywords and thus dependent of the use of the solution.

Then, all the parts of the list L_{w_i} are encrypted with a symmetric encryption scheme[2] under a key K_{w_i} that is specific to the keyword w_i:

$$\mathsf{Enc}_{\mathsf{Sym}}(K_{w_i}, L_{w_i,1}), \mathsf{Enc}_{\mathsf{Sym}}(K_{w_i}, L_{w_i,2}), \ldots, \mathsf{Enc}_{\mathsf{Sym}}(K_{w_i}, L_{w_i,x_{w_i}})$$

Note: The key K_{w_i} comes from a derivation of the key K^* only known by the Index Authority and a public label T specifying the time period[3]: $K_{w_i} \leftarrow \mathcal{H}(K^*, w_i, \mathsf{T})$.

Before to store these values in Index Chain Table, one needs to create their respective UIDs: ict_uid$_i$. It would be possible to store all of them with random numbers but it would imply to transmit later all these ict_uids to the user, as they correspond to the matching entries of its search request. The number of these UIDs is exactly the number x_{w_i} of blocks, and thus, is strictly smaller than the number of UIDs in the original list L_{w_i}. However, for the same reason than the length ℓ_{w_i} must be hidden, x_{w_i} must be hidden too. The solution would be to repeat the same process by adding a new Index Table, and so on, but it is not practical.

To avoid that, our solution will exploit linked lists to create the UIDs of Index Chain Table:

- from w_i and the key K_{w_i}, one can compute: $\mathcal{H}(K_{w_i}, w_i) \to$ ict_uid$_1$
- then, from ict_uid$_1$ and the key K_{w_i}, one can compute: $\mathcal{H}(K_{w_i}, \text{ict\_uid}_1) \to$ ict_uid$_2$
- then, from ict_uid$_2$ and the key K_{w_i}, one can compute: $\mathcal{H}(K_{w_i}, \text{ict\_uid}_2) \to$ ict_uid$_3$
- and so on, until to have x_{w_i} values of UIDs.

At the end, instead of having x_{w_i} values to transmit to the user, we only have the last value of the linked list to know the stop criterion (and the key K_{w_i}). This will be the goal of the Index Entry Table.

Example: If $x_{w_i} = 3$, the Index Chain Table could be:

UID	Value
…	…
ict_uid$_2$	$\mathsf{Enc}_{\mathsf{Sym}}(K_{w_i}, L_{w_i,2})$

[1] See in Section Security to learn more on the importance to not only hide the keywords and the db_uids but also the length of the result.

[2] See in Appendix for the description of the symmetric encryption scheme

[3] See in Section ReIndexing for an explanation of T.

UID	Value
…	…
ict_uid_1	$\text{Enc}_{\text{Sym}}(K_{w_i}, L_{w_i,1})$
…	…
ict_uid_3	$\text{Enc}_{\text{Sym}}(K_{w_i}, L_{w_i,3})$
…	…

with the ict_uids computed as presented above and in the other lines, the lists of the ict_uids of DB Table matching the other keywords also divided in blocks.

3.2.1 Size

About the size of Index Chain Table,

- the number of lines depends on the number of searchable keywords *and* on the number on necessary lines to store all the positions of the keywords
- in our implementation, a line is composed of:
  - `UID`: 32 bytes
  - `Value`:

	Nonce	AES-GCM encrypted data	MAC
Size (bytes)	12	$32 \times t$	16

3.3 Index Entry Table

The goal of Index Entry Table is to store all the last values of the linked lists for all the possible keywords with the symmetric key used in the chain.

First, for each keyword, the data is symmetricly encrypted under a common secret key K_{uid}. Then, the UIDs are computed from a common secret key K_{value}.

Example: For a keyword w_i, if ict_uid$_3$ is the last value of the linked list and K_{w_i} is the key used in Index Chain Table, $(\text{ict\_uid}_3, K_{w_i}, w_i)$ is encrypted under K_{value}:

$$\text{iet\_value}_i = \text{Enc}_{\text{Sym}}(K_{\text{value}}, (\text{ict\_uid}_3, K_{w_i}, w_i))$$

Then, the iet_uid is computed:

$$\text{iet\_uid}_i = \mathcal{H}(K_{\text{uid}}, w_i, \text{T})$$

Note: The two keys K_{uid} and K_{value} come from a secret key K known by all the authorized entities (i.e. the Index Authority and all the users).

Finally, the Index Entry Table looks like:

UID	Value
…	…
$\mathcal{H}(K_{\text{uid}}, w_i, \text{T})$	$\text{Enc}_{\text{Sym}}(K_{\text{value}}, (\text{ict\_uid}_{x_{w_i}}, K_{w_i}, w_i))$
…	…

3.3.1 Size

About the size of Index Entry Table,

- the number of lines only depends on the number of searchable keywords
- in our implementation, a line is composed of:
  - `UID`: 32 bytes
  - `Value`:

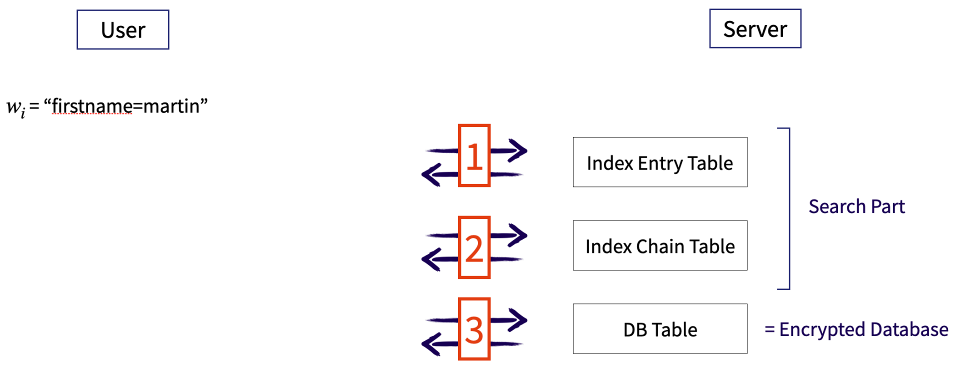
	Nonce	AES-GCM encrypted data	MAC
Size (bytes)	12	`Size_UID` + `Size_Key` + `Size_Keyword`	16

where:

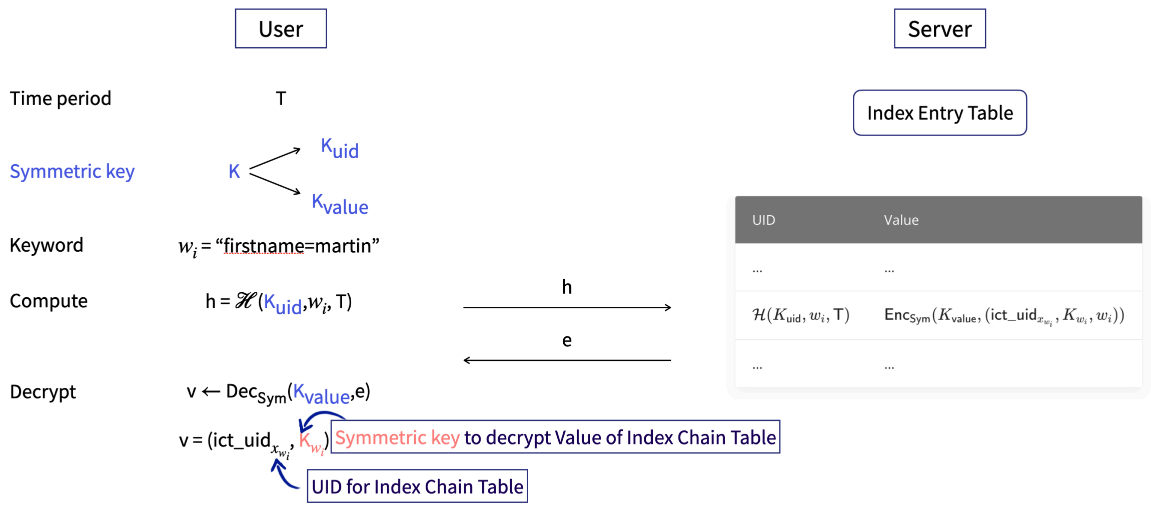
- `Size_UID`: 32 bytes
- `Size_Key`: 32 bytes
- `Size_Keyword`: in bytes

3.4 Search Query Process

We recall the scenario: a user wants to make a search query on an encrypted database hosted in an external (untrusted) server. Now, we will see the interactions between this user and the server. Basically, the search query process travels along the tables in the opposite order as describe before.

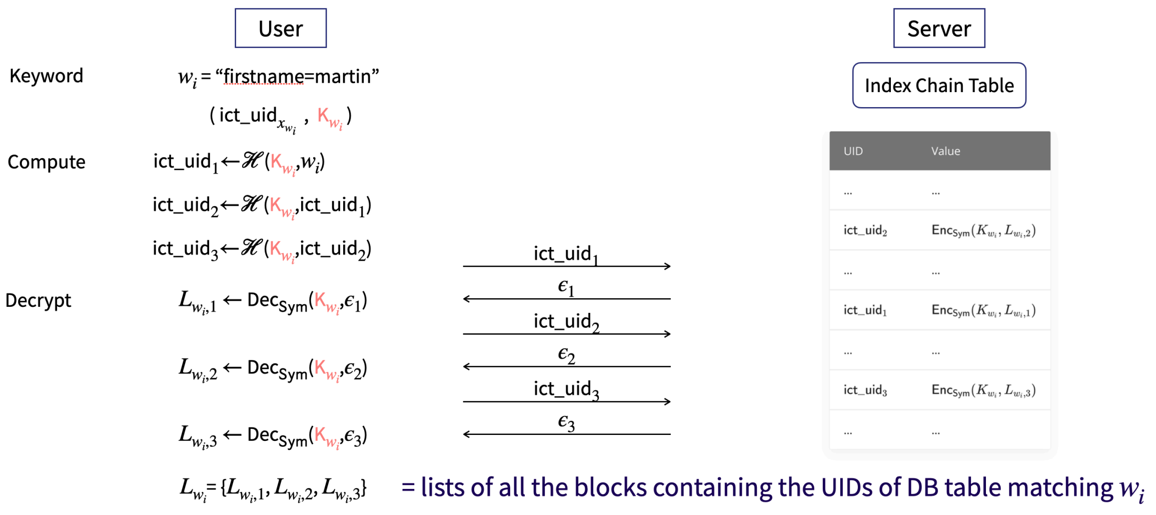


In the figure below, the user queries the Index Entry Table:


User
Server
Time period
T
Symmetric key
K
K_{uid}
K_{value}
Keyword
w_i = "firstname=martin"
Compute
h = $\mathscr{H}(K_{uid}, w_i, T)$
h
e
Decrypt
$v \leftarrow Dec_{Sym}(K_{value}, e)$
$v = (ict\_uid_{x_{w_i}}, K_{w_i})$
Symmetric key to decrypt Value of Index Chain Table
UID for Index Chain Table
Index Entry Table

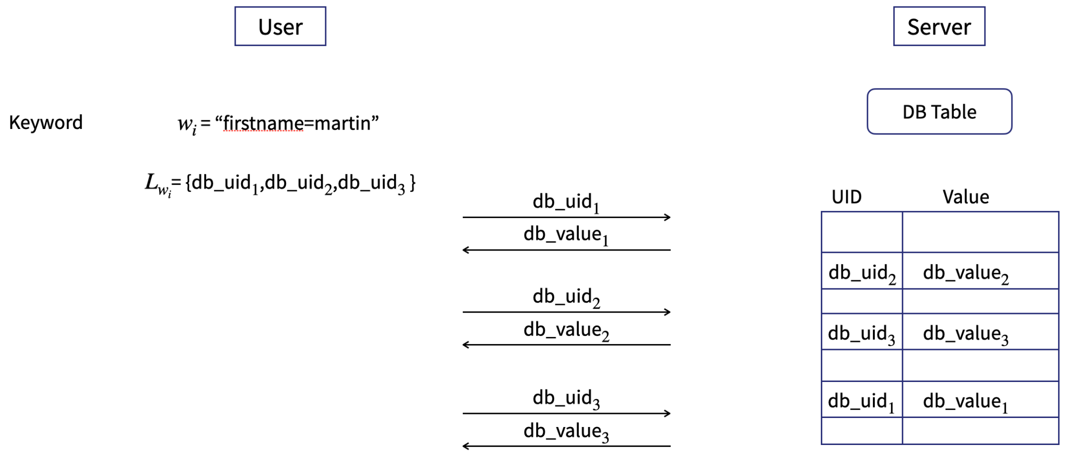
UID	Value
...	...
$\mathcal{H}(K_{\text{uid}}, w_i, \text{T})$	$\text{Enc}_{\text{Sym}}(K_{\text{value}}, (\text{ict\_uid}_{x_{w_i}}, K_{w_i}, w_i))$
...	...

Then, the user computes ict_uid_{i+1} until to reach $\text{ict\_uid}_{x_{w_i}}$ and queries the Index Chain Table:


User
Server
Index Chain Table
Keyword
w_i = "firstname=martin"
(ict_uid$_{x_{w_i}}$, K_{w_i})
Compute
ict_uid$_1$ ← $\mathscr{H}(K_{w_i}, w_i)$
ict_uid$_2$ ← $\mathscr{H}(K_{w_i}$, ict_uid$_1)$
ict_uid$_3$ ← $\mathscr{H}(K_{w_i}$, ict_uid$_2)$
Decrypt
$L_{w_i,1} \leftarrow \text{Dec}_{\text{Sym}}(K_{w_i}, \epsilon_1)$
$L_{w_i,2} \leftarrow \text{Dec}_{\text{Sym}}(K_{w_i}, \epsilon_2)$
$L_{w_i,3} \leftarrow \text{Dec}_{\text{Sym}}(K_{w_i}, \epsilon_3)$
$L_{w_i} = \{L_{w_i,1}, L_{w_i,2}, L_{w_i,3}\}$
= lists of all the blocks containing the UIDs of DB table matching w_i
ict_uid$_1$
ϵ_1
ict_uid$_2$
ϵ_2
ict_uid$_3$
ϵ_3

UID	Value
...	...
ict_uid$_2$	$\text{Enc}_{\text{Sym}}(K_{w_i}, L_{w_i,2})$
...	...
ict_uid$_1$	$\text{Enc}_{\text{Sym}}(K_{w_i}, L_{w_i,1})$
...	...
ict_uid$_3$	$\text{Enc}_{\text{Sym}}(K_{w_i}, L_{w_i,3})$
...	...

Finally, the user queries the DB Table:



UID	Value
db_uid$_2$	db_value$_2$
db_uid$_3$	db_value$_3$
db_uid$_1$	db_value$_1$

and tries to decrypt the results.

4 Update Process

4.1 Overview

Now, one can explain how to update the indexes and search requests. Overall, the changes can be:

- In DB Table:
  - to delete a line
  - to add a new line
  - to modify a line (the content or the right access)
- In Index Tables:
  - to delete already indexed keywords
  - to add new keywords for data already encrypted in DB Table

Remark: The changes are supposed to be made by the Index Authority except for the right access change on a DB line which is supposed to be made by the Right Access Authority.

4.1.1 Impact on the Efficiency

If you want to apply changes in order to improve the efficiency, here are some remarks:

- **Changing the content of DB Table** can increase the size of the index tables (as new keyword may be indexed).
- **Reducing the number of 'searchable' keywords** also reduces the size of the Index Tables but does not change the size of DB Table.
- **Increasing the number of 'searchable' keywords** does not affect the efficiency of a search request: the efficiency of a search request is *independent* in the total number of keywords.

4.2 Change in DB Table

4.2.1 Delete Line

Given the db_uid$_i$ of the line that must be deleted, delete the line db_line$_i$ in the DB Table.

The index tables do not change and thus, continue to refer to deleted contents. They are cleaned during a reindexing phase[4].

[4] See Section ReIndexing

4.2.2 Add Line

The add of a line simply follow the description presented in the full process:

- extraction of the keywords,
- encryption of the line and insertion in DB Table,
- encryption of the chain of keywords with insertions in the relevant index tables.

Here, the three tables change.

4.2.3 Modify Line

In our solution, a modification on the content (or possibly on the right access) will simply be the delete of the old line followed by the add of the new one containing the change.

A change will be:

- a delete of the corresponding encrypted line in DB Table,
- the add of a new line in DB Table containing the change one wanted to apply.

Hence, it requires to know which db_uid must be modified for the delete.

Here, the three tables change.

4.3 Change in Index Tables

In our solution, the choose of the indexed terms is completely free, they can be extracted without any structure as well as coming from the database structure. For example, the first two columns can be indexed and thus be 'searchable' while the other columns are not. Hence, one would be interested in modifying the set of keywords even without changing the content of the encrypted database.

4.3.1 Delete Keyword

It is possible to delete an indexed keyword without changing the DB Table by requesting the keyword and deleting all the corresponding lines in both Index Entry Table and Index Chain Table.

To delete an entire column of the searchable set, one needs to delete all the keywords present in it.

4.3.2 Add Keyword

To add new types of keywords (for example to index a new column of the database):

- For all the new entries (of DB Table), one simply considers all the keywords that must be indexed meaning the old ones and the new ones.
- For the already existing entries (of DB Table), all the concerned lines must be re-added.

4.4 ReIndexing

4.4.1 Why

With the changes presented in the previous sections, the Index Entry Table and Index Chain Table grow indefinitely. After the delete of lines in DB Table, some search requests still answer the deleted lines. Also, the server can link the modified lines to the changes.

The *ReIndexing* reduces the size of the index tables and blurs the information an attacker may have learned.

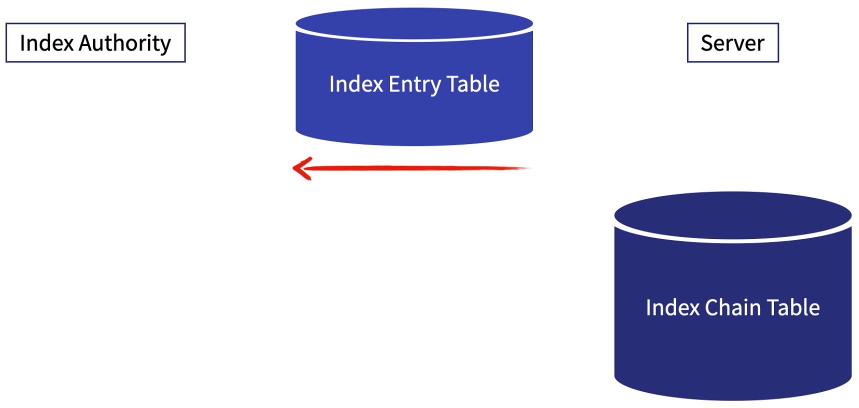
4.4.2 How

A reindexing:

- completely changes the Index Entry Table,
- cleans the search results of a fraction of keywords.

Index Entry Table

The first step of the Index Manager is to download the entire *Index Entry Table* and to delete all the outsourced existing lines:



Notation:

- old_IndexEntryTable $= \{(\text{iet\_uid}_i, \text{iet\_value}_i)\}$ (the downloaded Index Entry Table)
  - iet_uid$_i = \mathcal{H}(K_{\text{uid}}, w_i, \text{T})$
  - iet_value$_i = \text{Enc}_{\text{Sym}}(K_{\text{value}}, (\text{ict\_uid}_i, K_{w_i}, w_i))$ (AES-GCM encryption with nonce$_i$)
- new_IndexEntryTable $= \{(\text{iet\_uid}'_i, \text{iet\_value}'_i)\}$

Then, the Index Manager increments the (public) label $\text{T} \rightarrow \text{T}'$ and, for all the IET lines:

- computes iet_uid$'_i = \mathcal{H}(K_{\text{uid}}, w_i, \text{T}')$,
- changes nonce$_i$ used in the AES-GCM encryption into nonce$'_i$ to create iet_value$'_i$ an AES-GCM encryption with nonce$'_i$.

Hence, iet_value$'_i$ is a randomization of iet_value$_i$.

Cleaning

In a second step, the Index Manager will clean the search part for a fraction of keywords.

For that, the Index Manager:

- randomly chooses a fraction x [5] of the keywords by choosing a fraction of IET lines and decrypting them to recover w_i,
- for each of them,
  - makes the search request (i.e. obtains the list of all the position matching the keywords) and deletes all the corresponding outsourced lines,

[5] $x = [n \cdot (\log n + 0.58)]/t$ is the required number of keywords per reindexing needed to reach the full set of keywords (#`keywords` $= n$) in t reindexing phases. (cf. Coupon collector's problem)

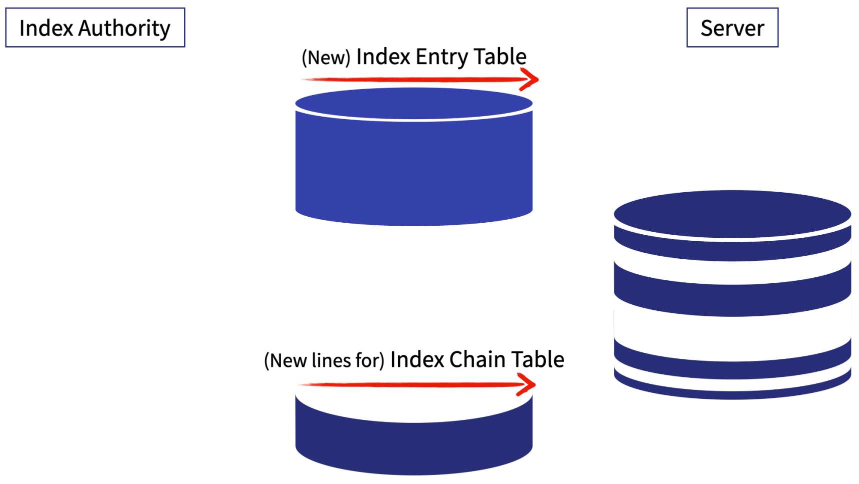
  - gets the result in the DB Tables (i.e. some of the positions will no longer exist),
  - for each non-existing value, deletes the db_uid from the non-encrypted list of position matching the keyword.

For each modified list of results, the Index Manager generates a completely new encryption of the indexes (i.e. create a new linked list of UID with the encryption of the db_uids). This creates a new ephemeral key $K_{w_i} \rightarrow K'_{w_i}$[6] and a new UID value $\text{ict\_uid}_i \rightarrow \text{ict\_uid}'_i$ to be stored in Index Entry Table to be able to reconstruct the linked list during a request.

Remark: We presented the two steps of the reindexing one by one for readability but optimizations are made in the code.

Finally, the Index Manager can push all the computed lines in a *randomized* order:

- inserts the new Index Entry Table,
- insert the new lines of Index Chain Table.



[6] $K'_{w_i} = \mathcal{H}(K^*, w_i, \mathsf{T}')$

5 Security

Findex solves the following search problem:

> How to *securely* recover the UIDs of DB Table to obtain the matching lines from a given keyword?

In this section, we explore the security details of Findex.

5.1 Keys

There are two keys in Findex. One for the authorized users to allow them to make search queries K and one for the Index Manager who can make update queries K^*. In fact, the Index Manager must also be able to make search queries and thus, to know K.

There is no security guarantee if a key is stolen. However, the keys can be changed and distributed to only a subset of authorized users to revoke some of them. In that case, the content of the tables must be reencrypted.

5.2 Server Storage

The server stores two tables: Index Entry Table and Index Chain Table. Each table is composed of several lines containing $(\text{uid}, \text{value})$. The uid looks like a random number as it is the output of a Hash Function $\mathcal{H}$ while value is the symmetric encryption of some elements. Hence, an adversary receiving a copy of Index Entry Table and/or Index Chain Table cannot learn information on the data.

5.3 Client - Server Communication

The requests consist of hashed values and the answers of encrypted messages. Hence, for an adversary, the learnable information remain on the frequencies of the requests. In particular, the server learns if a user asks two times the same keyword.

Example - The Searchable Directory: If `Dupont` is a frequent family name, the number of results matching the keyword will be large, on the contrary the number of matching results of an uncommon family name will be small. If the attacker can see the length of the results, it can exploit the frequency of the keywords to retrieve the search query of an honest user which must be forbidden.

Moreover, if there are not enough requests for different keywords then, the server can "see" the interactions related to a keyword. This is true for the three tables: Index Entry Table, Index Chain Table and DB Table.

One way to avoid that would be to use Oblivious Random Access Machine (ORAM) but it is not practical because not efficient. Hence, to avoid that, the simplest way is to generate fake requests to scramble the communications.

5.4 ReIndexing

To render statistical analysis even more complicated, the reindexing completely change the Index Entry Table and a fraction of Index Chain Table.

5.5 Dynamic Symmetric Searchable Encryption

The classical version of a Symmetric Searchable Encryption (SSE) scheme considers a static database meaning the outsourced data can not be updated or new records can not be sent later. To overcome this issue a Dynamic version has been designed in [1] and security formalized in [2]. Such a scheme handles dynamic file collection.

Findex is a *Dynamic Symmetric Searchable Encryption*[7] scheme meaning one can define a triple $(\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ consisting of one algorithm and two protocols (between a client and a server) as defined below:

Notation: Let $\mathsf{DB} = \{(w_i, L_{w_i})\}_i$ be the (non-encrypted) table associating each keyword w_i to its matching results[8].

- $\mathsf{Setup}(\mathsf{DB}) \rightarrow (\mathsf{EDB}, \mathcal{K})$: takes as input DB and outputs EDB an encrypted version of DB together with $\mathcal{K} = (K, K^*)$ a master secret key.
- $\mathsf{Search}(K, q; \mathsf{EDB}) = (\mathsf{SearchClient}(K, q), \mathsf{SearchServer}(\mathsf{EDB}))$: is a protocol between a client with input the search key K and a search query q and a server with input the encrypted table EDB. It outputs a list R of results to the client.
- $\mathsf{Update}(\mathcal{K}, q; \mathsf{EDB}) = (\mathsf{UpdateClient}(\mathcal{K}, q), \mathsf{UpdateServer}(\mathsf{EDB}))$: is a protocol between a client with input the master secret key $\mathcal{K}$ and an update query $q = (\mathsf{op}, \mathsf{in})$ where the operation can be add or delete and in is the content to be added or deleted. It outputs a new EDB' to the server.

Findex considers search queries restricted to a *single* keyword w[9].

A SSE scheme is said to be *correct* if the search protocol returns the correct result for every query: $\forall w_i, R \leftarrow \mathsf{Search}(K, w_i; \mathsf{EDB})$ is equal to L_{w_i}.

[7] See [3] for a thesis with a good introduction on SSE and the formal security definitions.

[8] See Overview for a presentation of L_{w_i}.

[9] Few SSE schemes deal with multiple keyword queries.

Forward Secrecy

In Symmetric Searchable Encryption, a security notion called Forward Secrecy ([4],[5]) guarantee that updates do not reveal any information a priori about the modifications they carry out.

In our solution there is no Forward Secrecy but it is possible to scramble the information during changes on the Index Tables by adding fake modifications.

6 Appendix

6.1 Cryptographic Algorithms

Hash Function

- $\mathcal{H}(\text{key}, m)$: Hash-based Message Authentication Code of m under the key key.

The implementation uses the HMAC-SHA256 scheme with a key key of size 256 bit.

Symmetric Scheme

- $\text{Enc}_{\text{Sym}}(\text{key}, m)$: Symmetric Encryption of m under the key key.
- $\text{Dec}_{\text{Sym}}(\text{key}, m)$: Symmetric Decryption of m under the key key. This algorithm “reverses” the $\text{Enc}_{\text{Sym}}(\text{key}, m)$ function.

The implementation uses the AES-GCM scheme with a key key of size 256 bit.

6.2 Keys

Key	Size (bytes)	Known by	Used in
K	32	All	AES-GCM
K^*	32	Index Authority	AES-GCM

where ‘All’ means all the authorized users and the Index Authority.

Remark: the other keys used in the entire protocol are derived from the keys above:

- K_{uid} derived from K
- K_{value} derived from K
- For each keyword w_i, K_{w_i} derived from K^*

References

[1] S. Kamara and C. Papamanthou, 'Parallel and dynamic searchable symmetric encryption', in *Financial cryptography and data security*, 2013, pp. 258–274.

[2] S. J. D. Cash J. Jaeger, 'Dynamic searchable encryption in very-large databases: Data structures and implementation.', 2014.

[3] R. Bost, 'Searchable encryption new constructions of encrypted databases', 2018, [Online]. Available: https://raphael.bost.fyi/phd_docs/R_BOST_PhD_Thesis.pdf.

[4] E. S. Emil Stefanov Charalampos Papamanthou, 'Practical dynamic searchable encryption with small leakage', 2014, [Online]. Available: https://www.ndss-symposium.org/ndss2014/programme/practical-dynamic-searchable-encryption-small-leakage/.

[5] R. Bost, 'ΣΟφΟς: Forward secure searchable encryption', in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1143–1154, doi: 10.1145/2976749.2978303.