Findex

Full Documentation

Version 1.0

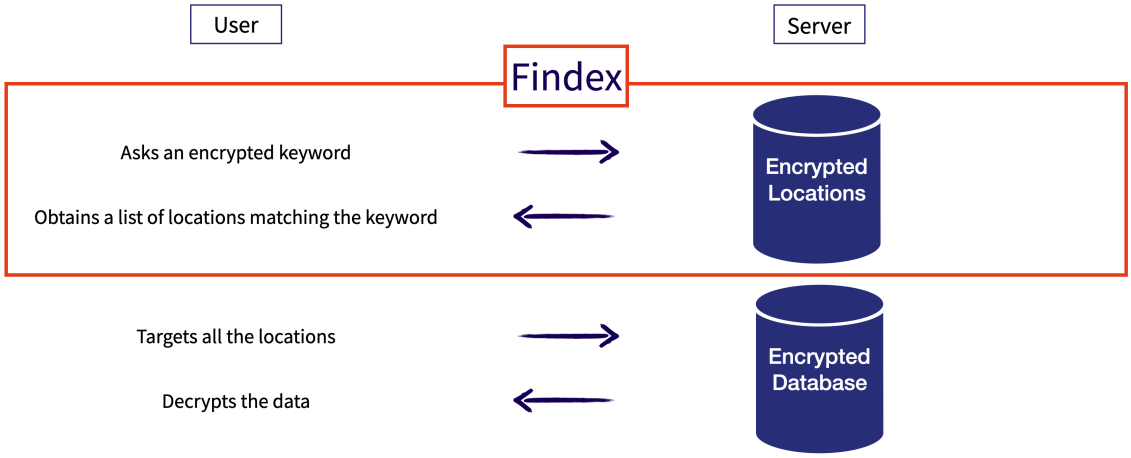
Date: 16/12/2022

Table Of Contents

1 Introduction

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

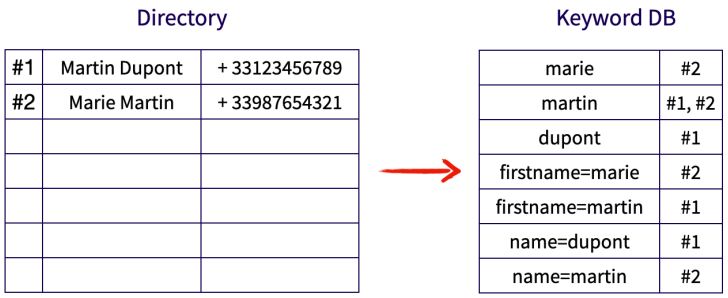
This documentation shows its running and explains the cryptographic details.



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.

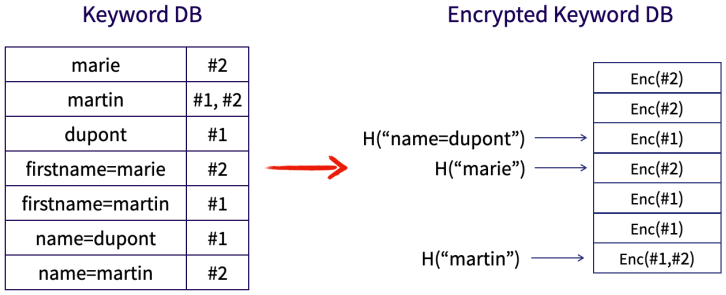
Here the directory is composed of two users. The first step consists of building a Keyword Database:



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

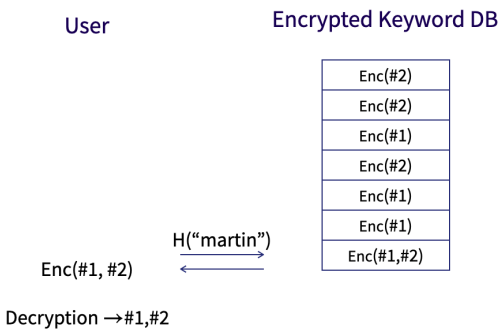
marie	#2
martin	#1, #2
dupont	#1
firstname=marie	#2
firstname=martin	#1
name=dupont	#1
name=martin	#2

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

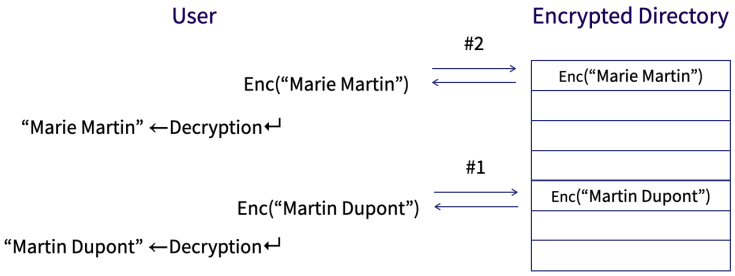


marie	#2
martin	#1, #2
dupont	#1
firstname=marie	#2
firstname=martin	#1
name=dupont	#1
name=martin	#2

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:

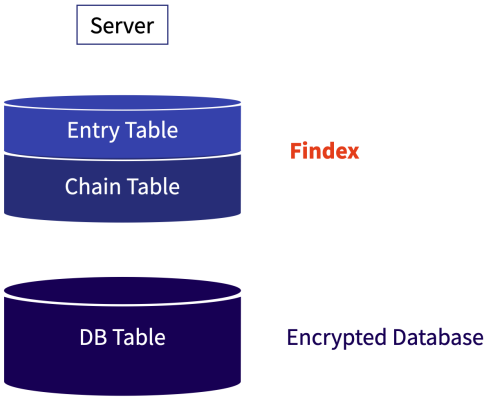


The user receives an encrypted message containing the location of all the matching queries: `#1`, `#2`. The user can then requests the server hosting the encrypted directory for these two locations:



3 Full Process

3.1 Overview



Findex relies on two server-side tables, *Entry Table* and *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.

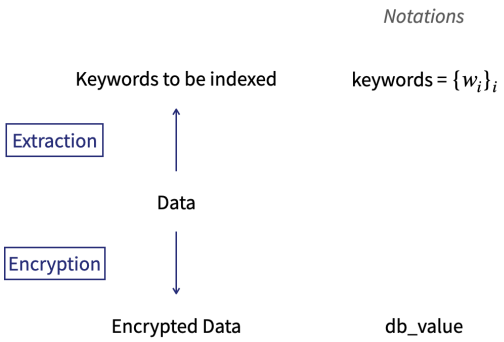
3.1.1 Symmetric Searchable Encryption

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 the same keyword can be extracted from several lines.



In this 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,
- $(\text{entry\_uid}_i, \text{entry\_value}_i) = (\text{uid}_i, \text{value}_i)$ of Entry Table,
- $(\text{chain\_uid}_i, \text{chain\_value}_i) = (\text{uid}_i, \text{value}_i)$ of Chain Table.

3.1.3 Index Tables

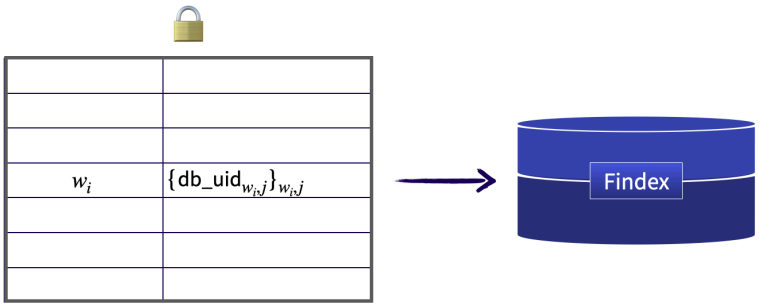


Figure 1: Findex Input

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

Chain Table: *securely* stores DB, thus all the lists $\text{DB}[w_i]$.

Example: The keyword “Martin” is present in lines 3, 5, and 10 of a cleartext 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 Chain Table will securely store $\{\text{db\_uid}_a, \text{db\_uid}_b, \text{db\_uid}_c\}$.

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

3.1.4 Search Query

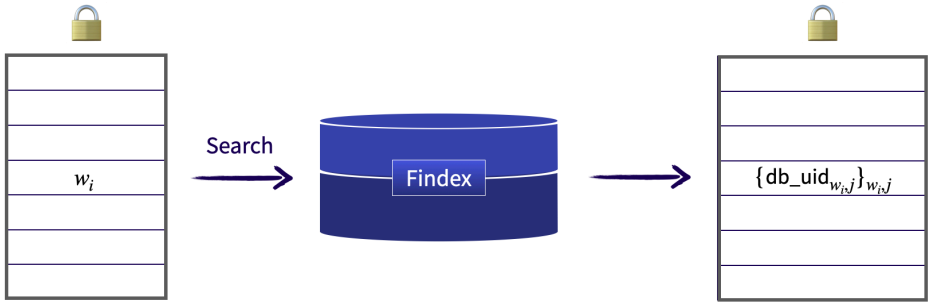


Figure 2: Search Query

Search Query: takes as input a bulk of keywords $\{w_i\}_i$ and outputs the bulk of the lists $\{\text{DB}[w_i]\}_i$.

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 Chain Table

Let us see the content of the Chain Table.

3.2.1 Chain Table Value

We denote by $\mathsf{DB}[w_i]$ the list of the UIDs[1] of DB Table matching the keyword w_i and $|\mathsf{DB}[w_i]|$ the number of such elements.

To hide $|\mathsf{DB}[w_i]|$[2], $\mathsf{DB}[w_i]$ will be divided into blocks of equal size B[3]:

$$\mathsf{DB}[w_i] = \{\mathsf{DB}[w_i]_1, \dots, \mathsf{DB}[w_i]_{L_i}\}$$

Potentially, the last block is not full and will be padded.

Then, all the blocks are encrypted with a symmetric encryption scheme[4] under a key $K_{w_i,\mathsf{value}}$[5] that is specific to the keyword w_i:

$$\mathsf{Enc}_{\mathsf{Sym}}(K_{w_i,\mathsf{value}}, \mathsf{DB}[w_i]_1), \dots, \mathsf{Enc}_{\mathsf{Sym}}(K_{w_i,\mathsf{value}}, \mathsf{DB}[w_i]_{L_i})$$

3.2.2 Chain Table UID

Before storing the chain values in the Chain Table, one needs to create their respective UIDs.

It would be possible to store all of them with random numbers but it would imply transmitting later all these chain_uids to the user, as they correspond to the matching entries of its search request. The number of these UIDs is exactly the number L_i of blocks, and thus, is strictly smaller than the number of UIDs in the original list $\mathsf{DB}[w_i]$. However, for the same reason that the length $|\mathsf{DB}[w_i]|$ must be hidden, L_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 Chain Table:

- from the key $K_{w_i,\mathsf{uid}}$, one can compute: $\mathcal{H}(K_{w_i,\mathsf{uid}}, \mathcal{H}(w_i)) \rightarrow \mathsf{chain\_uid}_1$
- then, from $\mathsf{chain\_uid}_1$ and the key $K_{w_i,\mathsf{uid}}$, one can compute: $\mathcal{H}(K_{w_i,\mathsf{uid}}, \mathsf{chain\_uid}_1) \rightarrow \mathsf{chain\_uid}_2$
- then, from $\mathsf{chain\_uid}_2$ and the key $K_{w_i,\mathsf{uid}}$, one can compute: $\mathcal{H}(K_{w_i,\mathsf{uid}}, \mathsf{chain\_uid}_2) \rightarrow \mathsf{chain\_uid}_3$

[1] In the implementation, $\mathsf{DB}[w_i]$ is a list of locations where a location can be a UID or something else such as another keyword. The size of a location is not fixed but, in this case, the location is first divided into blocks of fixed size.

[2] See in Section Security to learn more about the importance of not only hiding the keywords and the db_uids but also the length of the result.

[3] The size B is constant for a given Findex index. However, it may vary between use cases since the average length of the searchable keywords can vary.

[4] See Appendix for the description of the symmetric encryption scheme

[5] The two keys $K_{w_i,\mathsf{uid}}$ and $K_{w_i,\mathsf{value}}$ are derived from K_{w_i} randomly chosen by the Index Authority.

- and so on, until the L_i values have been generated.

In the end, instead of having L_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} to be able to derive $K_{w_i,\text{uid}}$ and $K_{w_i,\text{value}}$). This will be the goal of the Entry Table.

Example: If $L_i = 3$, the Chain Table could be:

UID	Value
…	…
chain_uid$_2$	$\text{Enc}_{\text{Sym}}(K_{w_i,\text{value}}, \text{DB}[w_i]_2)$
…	…
chain_uid$_1$	$\text{Enc}_{\text{Sym}}(K_{w_i,\text{value}}, \text{DB}[w_i]_1)$
…	…
chain_uid$_3$	$\text{Enc}_{\text{Sym}}(K_{w_i,\text{value}}, \text{DB}[w_i]_3)$
…	…

with in the other lines, blocks of B DB Table UIDs matching other keywords.

3.2.3 Size

About the size of Chain Table,

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

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

3.3 Entry Table

The role of the Entry Table is to store for each w_i: the last value of its associated linked list, the key K_{w_i} used in this list and $\mathcal{H}(w_i)$.

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

Example: For a keyword w_i, the entry_uid is computed:

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

Then, if chain_uid_3 is the last value of the linked list and K_{w_i} is the key used in Chain Table, $(\text{chain\_uid}_3, K_{w_i}), \mathcal{H}(w_i)$ is encrypted under K_{value}:

$$\text{entry\_value}_i = \text{Enc}_{\text{Sym}}(K_{\text{value}}, (\text{chain\_uid}_3, K_{w_i}, \mathcal{H}(w_i)))$$

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

Finally, the Entry Table looks like this:

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

3.3.1 Size

About the size of the 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`:

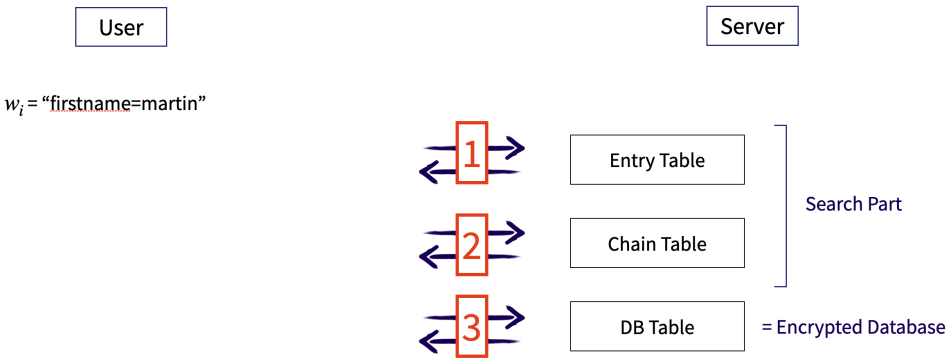
	Nonce	AES-GCM encrypted data	MAC
Size (bytes)	12	`UID.len` + `key.len` + `hash.len`	16

where:

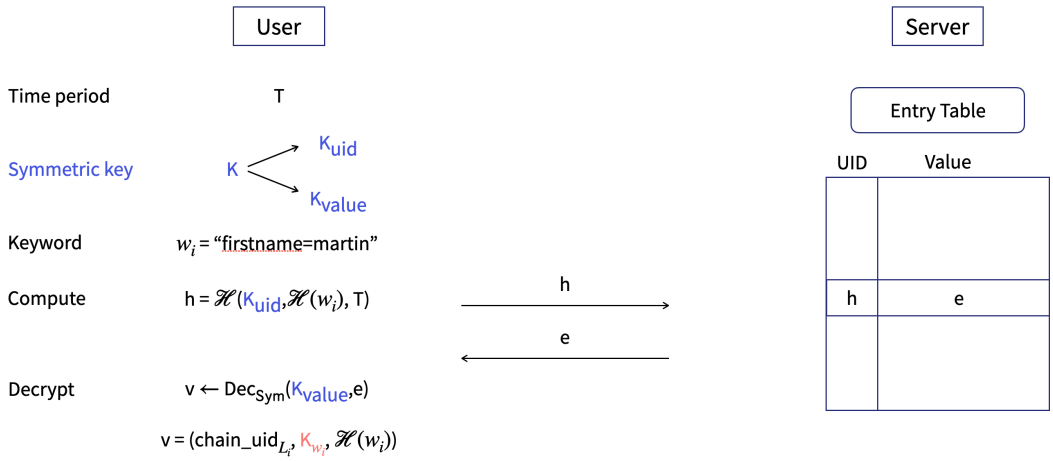
- `UID.len`: 32 bytes
- `key.len`: 16 bytes
- `hash.len`: 32 bytes

3.4 Search Query Process

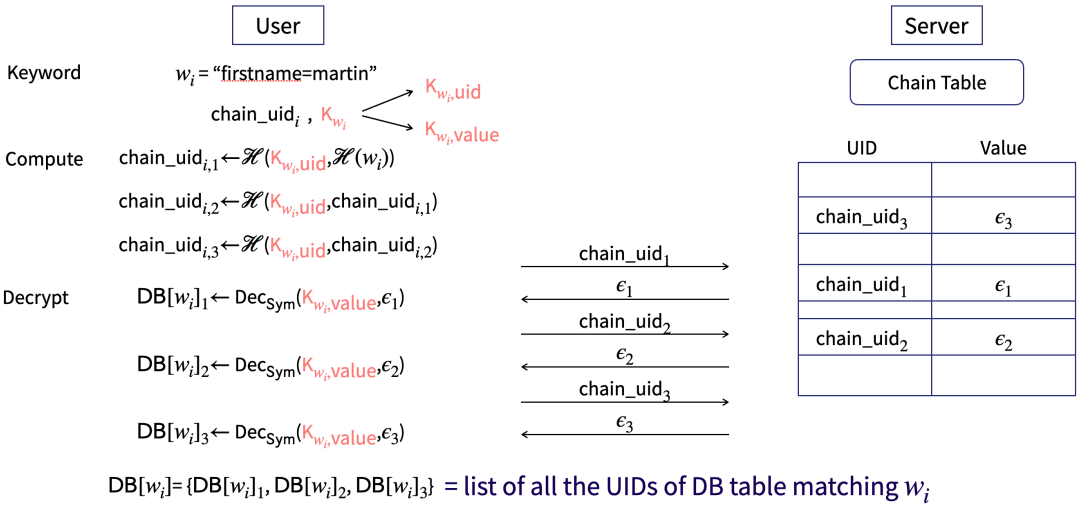
We recall the scenario: a user wants to make a search query on an encrypted database hosted in an external (untrusted) server. Now, we will see the interactions between this user and the server. Basically, the search query process travels along the tables in the opposite order as the one described before.



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

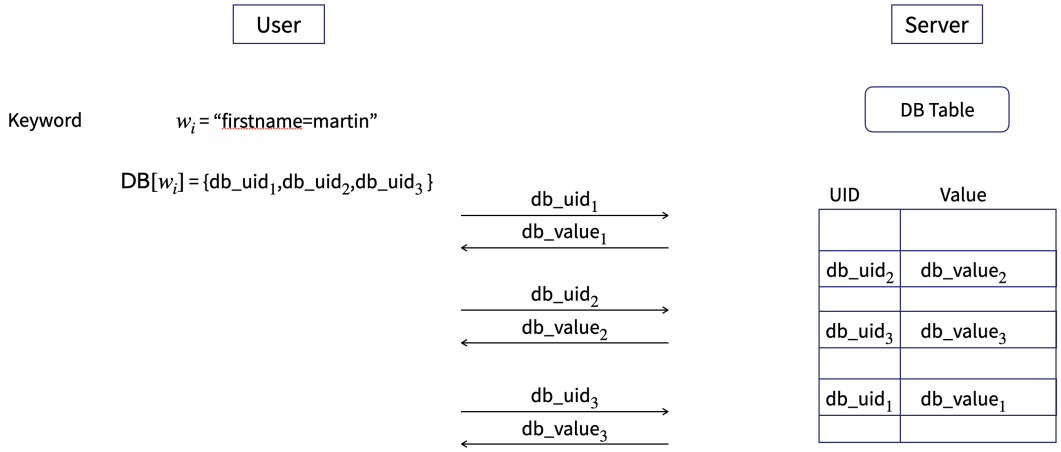


Then, the user computes $chain\_uid_{i+1}$ until $chain\_uid_{L_i}$ is found and queries the Chain Table:



$DB[w_i] = \{DB[w_i]_1, DB[w_i]_2, DB[w_i]_3\}$ = list of all the UIDs of DB table matching w_i

Finally, the user queries the DB Table:



and tries to decrypt the results.

4 Update Process

4.1 Overview

Now, one can explain how to update the indexes to update a search request.

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 in the Index Tables are supposed to be made by the Index Authority.

4.1.1 Impact on the Efficiency

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

- **Changing the content of DB Table** can increase the size of the index tables (as new keywords may be indexed).
- **Reducing the number of searchable keywords** also reduces the size of the Index Tables but does not change the size of the 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* of 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[6].

[6] See Section ReIndexing

4.2.2 Add Line

The addition of a line simply follows 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 of the content (or possibly of the right access) will simply be the deletion of the old line followed by the addition of the new one containing the change.

A change will be:

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

Hence, it requires knowing which db_uid must be modified for the deletion.

Here, the three tables change.

4.3 Change in Index Tables

In our solution, the choice 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, it could make sense to modify 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 Entry Table and Chain Table grow indefinitely. After the deletion of lines in the 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.

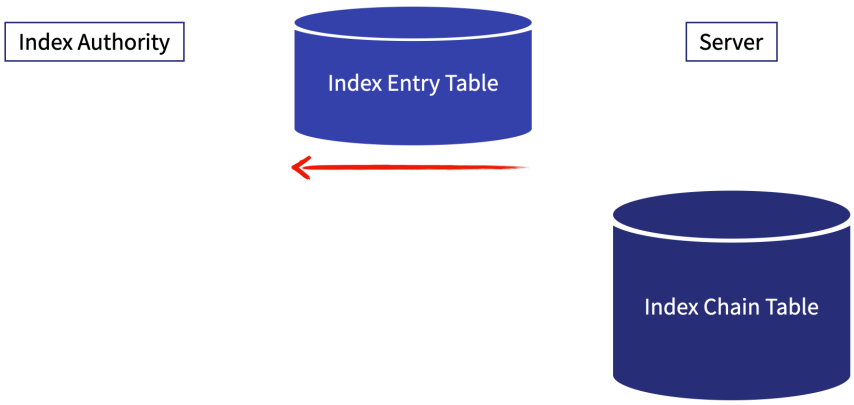
4.4.2 How

A reindexing:

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

Entry Table

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



Notation: - old_entry_table = {(entry_uid$_i$, entry_value$_i$)} (the downloaded Entry Table) - entry_uid$_i$ = $\mathcal{H}(K_{\text{uid}}, \mathcal{H}(w_i), \mathsf{T})$ - entry_value$_i$ = $\text{Enc}_{\text{Sym}}(K_{\text{value}}, (\text{chain\_uid}_{L_i}, K_{w_i}, \mathcal{H}(w_i)))$ (AES-256-GCM encryption with nonce$_i$) - $new_entry_table = \{(\text{entry\_uid}'_i, \text{entry\_value}'_i)\}$

Then, the Index Authority increments the (public) label $\mathsf{T} \rightarrow \mathsf{T}'$ and, for all the Entry Table lines:

- computes $\text{entry\_uid}'_i = \mathcal{H}(K'_{\text{uid}}, \mathcal{H}(w_i), \mathsf{T}')$,
- changes nonce$_i$ used in the AES-256-GCM encryption into nonce$'_i$ to create entry_value$'_i$ an AES-256-GCM encryption with nonce$'_i$.

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

Cleaning

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

For that, the Index Authority:

- implicitly chooses a fraction X of the keywords (of size x[7]) by choosing a random fraction of the Entry Table and decrypting it,
- for each of them,
  - makes the search request (i.e. obtains the list of all the positions matching the keywords) and deletes all the corresponding outsourced lines,
  - 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 positions matching the keyword.

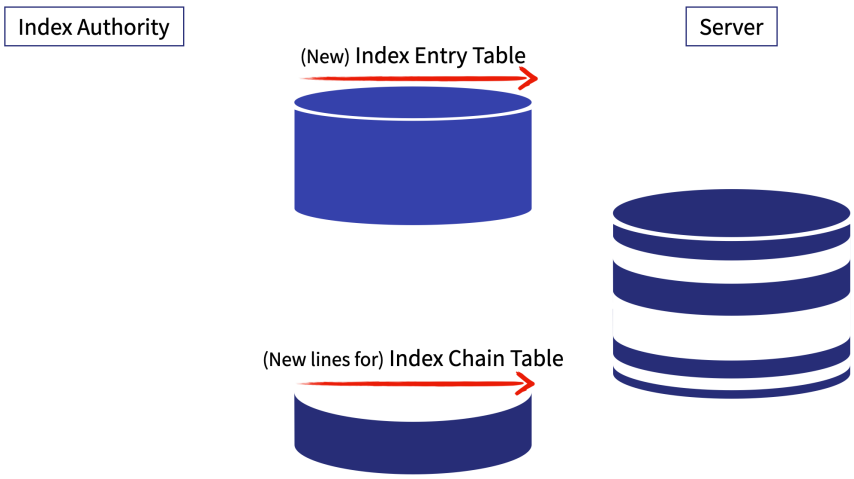
[7] $x = [n \cdot (\log n + 0.58)]/t$ is the required number of keywords per reindexing needed to reach the full set of keywords (#keywords $= n$) in t reindexing phases. (cf. Coupon collector's problem).

For each modified list of results, the Index Authority 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 $K_{w_i} \rightarrow K'_{w_i}$ and a new UID value $\text{chain\_uid}_i \rightarrow \text{chain\_uid}'_i$ to be stored in 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 Authority can push all the computed lines in a *randomized* order:

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



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 Key

The key K in Findex allows the authorized users to make search queries and the Index Authority to make update queries.

There is no security guarantee if the key is stolen. However, the key 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: Entry Table and 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 the Entry Table and/or Chain Table cannot learn information on the data.

5.3 Client - Server Communication

The requests consist of hashed values and the answers to encrypted messages. Hence, for an adversary, the learnable information is limited to 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: the Entry Table, the Chain Table and the 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 communications.

5.4 ReIndexing

To make statistical analyses even more complicated, reindexing operations completely change the Entry Table and a fraction of the 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*[8] scheme meaning one can define a triple $(\text{Setup}, \text{Search}, \text{Update})$ consisting of one algorithm and two protocols (between a client and a server) as defined below:

Notation: Let $\text{DB} = \{(w_i, \text{DB}[w_i])\}_i$ be the (non-encrypted) table associating each keyword w_i to its matching results[9].

- $\text{Setup}(\text{DB}) \rightarrow (\text{EDB}, K)$: takes as input DB and outputs EDB an encrypted version of DB together with K a secret key.
- $\text{Search}(K, q; \text{EDB}) = (\text{SearchClient}(K, q), \text{SearchServer}(\text{EDB}))$: is a protocol between a client with input the secret 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.
- $\text{Update}(K, q; \text{EDB}) = (\text{UpdateClient}(K, q), \text{UpdateServer}(\text{EDB}))$: is a protocol between a client with input the secret key K and an update query $q = (\text{op}, \text{in})$ 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[10].

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

Forward Secrecy

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

[9] See Overview for a presentation of $\text{DB}[w_i]$.

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

In Symmetric Searchable Encryption, a security notion called Forward Secrecy ([4],[5]) guarantees 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

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

The implementation uses the KMAC128 scheme with a key of size 32 bytes.

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 AES256-GCM scheme with a key of size 32 bytes.

6.2 Keys

Key	Size (bytes)	Obtained from	Used in	Known by
K	16	random		All
K_{uid}	32	derivation of K	KMAC128	
K_{value}	32	derivation of K	AES256-GCM	
K_{w_i}	16	random		
$K_{w_i,\text{uid}}$	32	derivation of K_{w_i}	KMAC128	
$K_{w_i,\text{value}}$	32	derivation of K_{w_i}	AES256-GCM	

where 'All' means all the authorized users and the Index Authority.

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, 'ΣOϕOς: 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.

