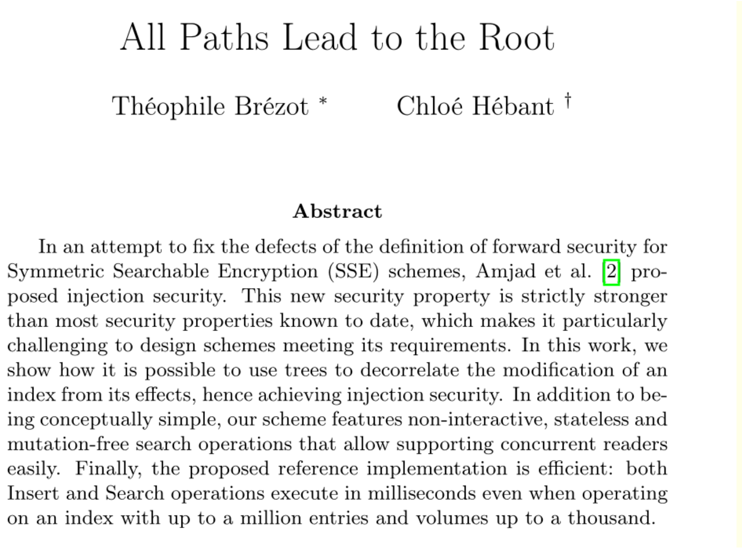
All paths lead to the root

Théophile BRÉZOT

We are on e-print (2025/XXX)!



All Paths Lead to the Root

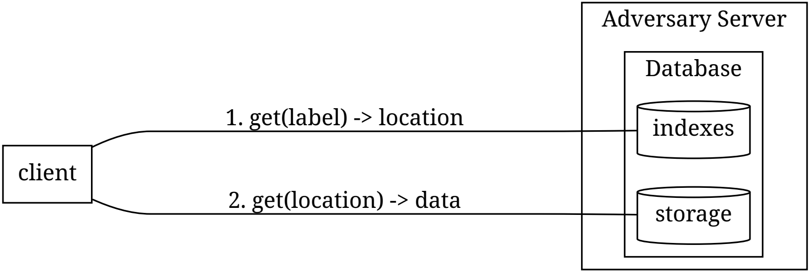
Théophile Brézot * Chloé Hébant †

Abstract

In an attempt to fix the defects of the definition of forward security for Symmetric Searchable Encryption (SSE) schemes, Amjad et al. [2] proposed injection security. This new security property is strictly stronger than most security properties known to date, which makes it particularly challenging to design schemes meeting its requirements. In this work, we show how it is possible to use trees to decorrelate the modification of an index from its effects, hence achieving injection security. In addition to being conceptually simple, our scheme features non-interactive, stateless and mutation-free search operations that allow supporting concurrent readers easily. Finally, the proposed reference implementation is efficient: both Insert and Search operations execute in milliseconds even when operating on an index with up to a million entries and volumes up to a thousand.

Searchable Symmetric Encryption (SSE)

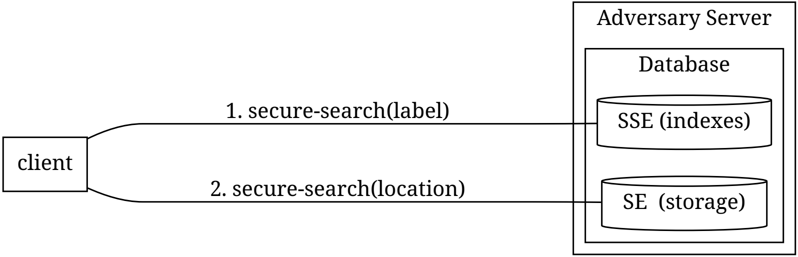
How to retrieve data from an adversary server?



Leaks:

1. the label and the location;
2. the location and the data.

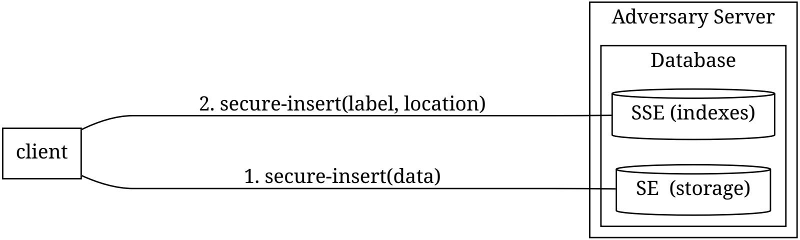
Controlling the leakage



Leaks:

1. $\mathcal{L}_{SSE}(label)$;
2. $\mathcal{L}_{SE}(location)$.

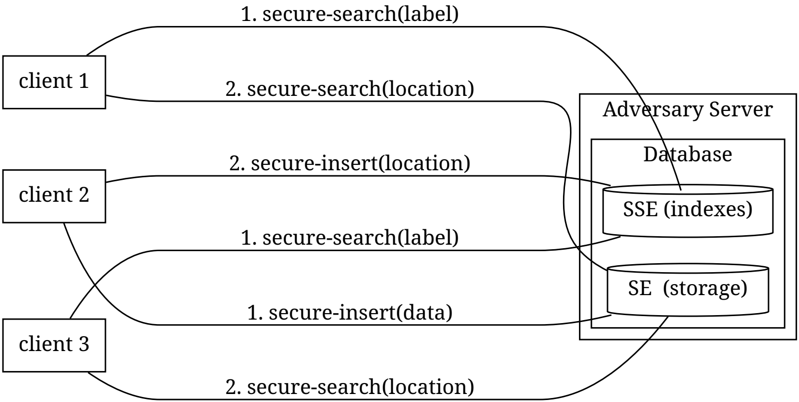
Supporting dynamic states



Leaks:

1. $\mathcal{L}_{SE}(Insert, data)$.
2. $\mathcal{L}_{SSE}(Insert, label, location)$;

Supporting concurrent queries



Structured Encryption

Index

Index ADT:

- *States*: $\mathbb{L}^{\mathbb{V}^*}$
- *Operations*:

```
Search :: state -> label -> Set value
Insert :: state -> label -> value -> ()
```

Index STE:

- *States*: $\mathbb{K}^{\mathbb{L}^{\mathbb{V}^*}}$
- *Operations*:

```
 Setup :: () -> (key * IO ())
Search :: key -> label -> IO (Set value)
Insert :: key -> label -> value -> IO ()
```

Multi-Map (MM)

MM ADT:

- *States*: $\mathbb{L}^{\mathbb{V}^*}$
- *Operations*:

```
Search :: state -> label -> List value
Insert :: state -> label -> value -> ()
```

MM STE:

- *States*: $\mathbb{K}^{\mathbb{L}^{\mathbb{V}^*}}$
- *Operations*:

```
 Setup :: () -> (key * IO ())
Search :: key -> label -> IO (List value)
Insert :: key -> label -> value -> IO ()
```

Multi-Map STE to Index STE transformation

- Index setup and search are the very MM operations:

```
Index::setup  = MM::setup
Index::insert = MM::insert
```

- The result of an Index search is the result of the MM search without duplicates:

```
Index::search = unique . MM::search
                where
                  unique vs = fold Set::insert
                                   EmptySet vs
```

Why bother implementing a more constrained ADT?

Semi-dynamic MM to fully-dynamic MM transformation

Associative ADT:

States: $\mathbb{L}^{\mathbb{T}(\mathbb{V})}$

Operations:

Search `:: s -> l -> T v`

Mutate `:: (Mutation M) => s -> l -> M T v -> ()`

where:

```
class (Container T) => Mutation T v where
  apply :: v -> (T v) -> (T v)
```

Semi-dynamic MM to fully-dynamic MM transformation

The fully-dynamic Multi-Map is an associative ADT:

- Insertion:

```
instance Mutation (MMInsertion List value) where
  (apply) = (Cons)
```

- Deletion:

```
instance Mutation (MMDeletion List value) where
  (apply) = (remove)
           where
             remove v []    = []
             remove v [v l] = remove v l
             remove v [w l] = Cons w (remove v l)
```

Semi-dynamic MM to fully-dynamic MM transformation

Specialized mutations are T transformations[1]*:*
all we need is to log them!

Journaling Multi-Map ADT:

- *States*: $\mathbb{L}^{(\mathbb{T}(\mathbb{V})^{\mathbb{T}(\mathbb{V})})^*}$
- *Operations*:

```
Search state -> l -> List (Tx T v)
Insert state -> l -> (Tx T v) -> ()
```

where `Tx T v = T v -> T v`.

[1] More precisely, they form a monoid and can therefore be reduced.

Semi-dynamic MM to fully-dynamic MM transformation

Implementing any (fully-dynamic) associative ADT on top of a (semi-dynamic) multi-map is therefore simple!

```
search s l     = let transformations = MM::search s l
                 in (reduce transformations) T::empty

mutate s l m v = MM::insert s l (m v)
```

PLOC is actually *simple*!

Challenge

1. Do not to leak anything during insertion:
   $\mathcal{L}(\mathit{Insert}, \mathit{label}, \mathit{value}) = \perp$

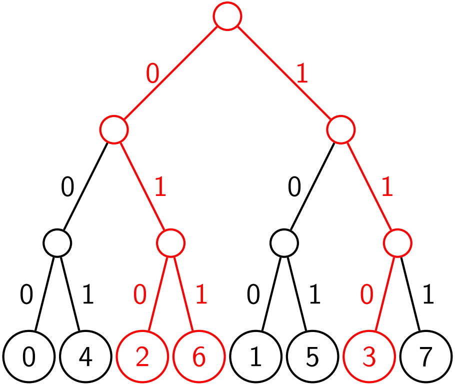
2. Only leak a (meaningless) UID of the label:
   $\mathcal{L}(\mathit{Search}, \mathit{label}) = \mathrm{sp}$

Search

Simply derive the set of target branches:

- $\mathrm{PRF}(key, cat, 0) = 3 = \texttt{b011}$
- $\mathrm{PRF}(key, cat, 1) = 2 = \texttt{b010}$
- $\mathrm{PRF}(key, cat, 2) = 6 = \texttt{b110}$

Mind the endianness!

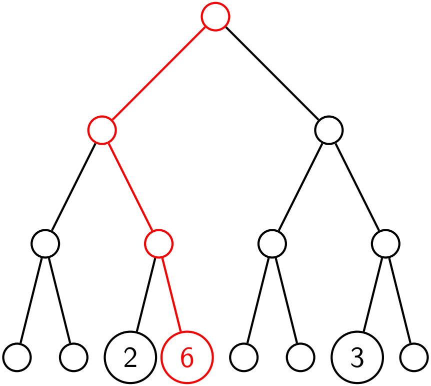


Insert – Datum

```
type Datum = target * value
```

A datum must always be stored on its target branch.

For example with `(6 food)`:



Insertion – Uniform Scheduling

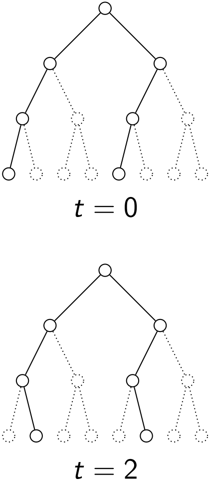
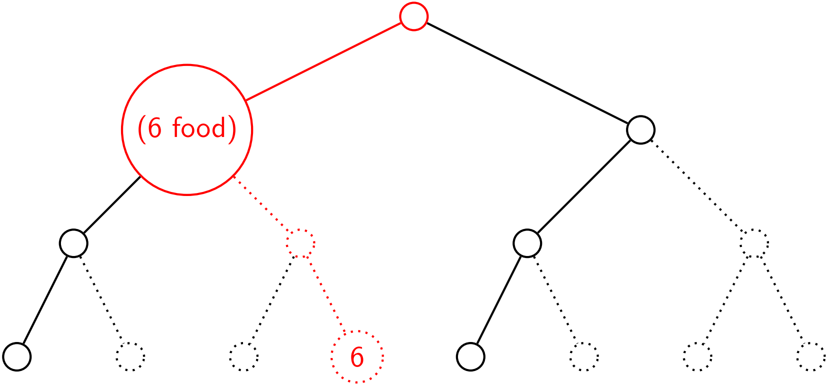
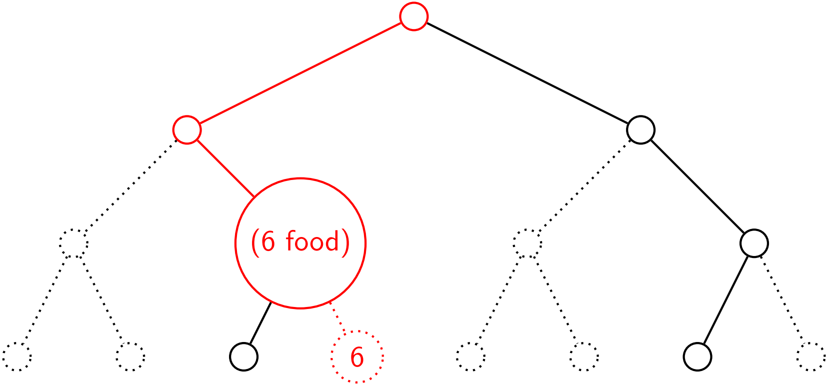


Figure 1: Scheduled subtrees for $N = 8$ and $n = 2$.

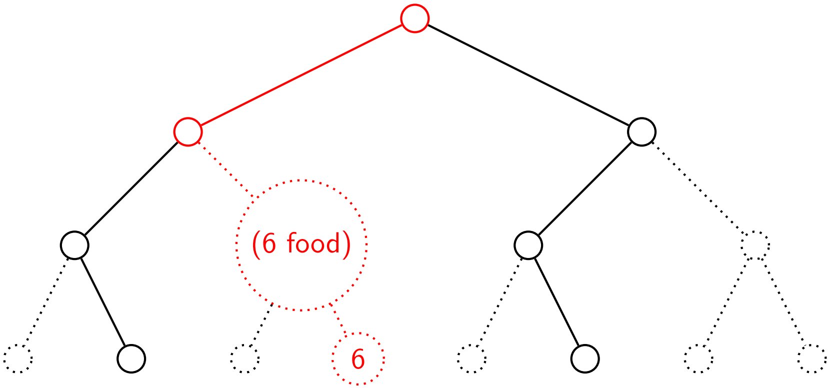
Insert – Compaction 1



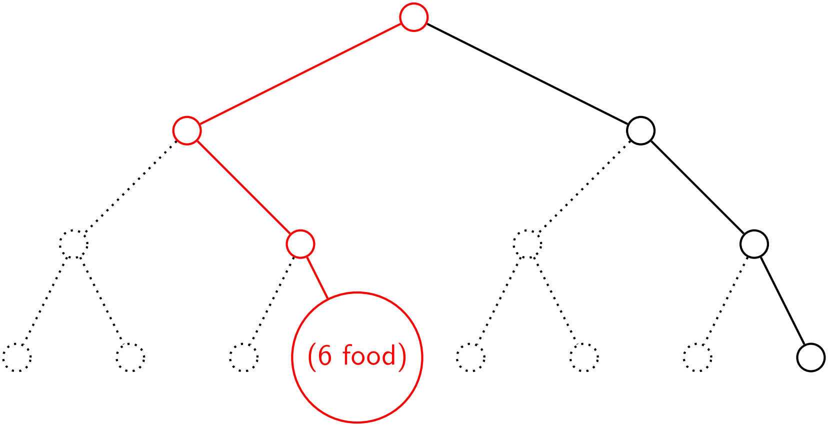
Insert – Compaction 2



Insert – Compaction 3



Insert – Compaction 4



Insert – Compaction

Can the compaction prevent tree overflow?

Conclusion – PLOC is *injection-secure*

- $\mathcal{L}(\mathit{Search}, \mathit{label}) = \{\mathit{target\text{-}branch}\}$
- $\mathcal{L}(\mathit{Insert}, \mathit{label}, \mathit{value}) = \perp$

Conclusion – PLOC is *efficient*

With a *simple* implementation:

$n \setminus B$	2^{10}		2^{16}		2^{20}	
16	2.1msec	1.8msec	26msec	3.1msec	0.17sec	3.8msec
64	2.1msec	5.4msec	25msec	11msec	0.12sec	13msec
256	2.0msec	18msec	25msec	34msec	0.12sec	45msec

Table 1: (*Search* *Insert*) performances in function of n and B for $V = \sqrt{B}$.

Future works

Can we improve the performance?

- Search performance is in $O(V)$:
  - can we store more than one datum per target branch? $\Rightarrow O(\frac{V}{m})$
- Search bandwidth is in $O(c \lg B)$:
  - can we reduce the depth by $\lg c$? $\Rightarrow O(\lg B)$

What about concurrency?

Reliance on an synchronized mutable state due to:

- MM semantics (order)
  - implement the index directly?
  - relax progress property?
- Uniform scheduling (next scheduled branches)
  - can compaction work with a random scheduling?
  - relax progress property?

What about concurrency?

Reliance on an synchronized mutable state due to:

- MM semantics (order) $O(L) \Rightarrow$ bad
  - ~~implement the index directly?~~
  - relax progress property + relax target selection
- Uniform scheduling (next scheduled branches) $O(1)$
  - ~~can compaction work with a random scheduling?~~
  - relax progress property?

What about the data-related leakage?

- Store data directly inside the SSE?
  - what is the impact on performance?
- Use an independent scheme with no leakage?
  - with what performances?
  - can it be compatible with concurrent queries?

What about the data-related leakage?

- Store data directly inside the SSE?
  - what is the impact on performance?
    Big due to volume-hiding
- Use an independent scheme with no leakage?
  - with what performances?
  - can it be compatible with concurrent queries?

What about the data-related leakage?

- Store data directly inside the SSE?
  - what is the impact on performance?
    Big due to volume-hiding
- Use an independent scheme with no leakage?
  - with what performances?
    OK $O(\log N)$
  - can it be compatible with concurrent queries?

What about the data-related leakage?

- Store data directly inside the SSE?
  - what is the impact on performance?
    Big due to volume-hiding
- Use an independent scheme with no leakage?
  - with what performances?
    OK $O(\log N)$
  - can it be compatible with concurrent queries?
    NO requires a lock

Thanks!