# Findex: A Concurrent and Database-Independent Searchable Encryption Scheme

Théophile Brézot and Chloé Hébant

Cosmian, Paris, France

**Abstract.** State-of-the-art database implementations offer a wide range of functionalities and impressive performances while supporting highly concurrent loads. However they all rely on the server knowing the content of the database, which raises issues when sensitive information is being stored on a server that cannot be trusted. Encrypting documents before sending them to a remote server solves the confidentiality issue at the cost of loosing the keyword search functionality. Cryptographic primitives such as Symmetric Searchable Encryption (SSE) schemes have been proposed to recover this functionality. However, no SSE construction properly defines correctness and successfully guarantees security in a concurrent setting. This paper attempts a first step in this direction by recommending linearizability as the standard notion of correctness for a concurrent SSE. We study the impact of concurrency on security and stress the need for finer-grained security models. Hence, we propose the log-security model that guarantees security against an adversary having access to persistency-related logs, fixing a blind spot in the snapshot model while capturing security in a concurrent setting. We also build the first concurrent SSE solution proven correct and secure in a concurrent setting, that can be implemented on top of any database. Our scheme proved to be fast thanks to optimal wait-free search operations and sequentially-optimal, lock-free modifications, that both execute under one micro-second per binding, while only incurring a 13.3% storage expansion.

**Keywords:** SSE · Linearizability · Concurrency

## 1 Introduction

The use of databases is pervasive in today's numerical systems. State-of-the-art implementations offer a wide range of functionalities and impressive performances while supporting highly concurrent loads. However they all rely on the server knowing the content of the database, which raises issues when sensitive information is being stored on a server that cannot be trusted. With the development of the SaaS and cloud models, in which users do not control the machine that runs the database, these issues have become critical.

Encrypting data before storing it to the database is not acceptable for most use-cases as it breaks features like keyword search. One cryptographic solution used to recover this functionality is Symmetric Searchable Encryption (SSE) [Goh03, CGKO06, Bos16, BMO17, AKM19, MR23, AKM23]. Traditionally, a Symmetric Searchable Encryption scheme is a cryptographic primitive composed of three protocols executed in a sequential model, and thus involving a *single* client and a *single* server: the client initiates all the necessary materials thanks to a Setup algorithm; then, using its secret key, it can index a new entry under a given keyword using the Insert algorithm or retrieve values indexed under a keyword using the Search algorithm. However, to the best of our knowledge, no

E-mail: theophile.brezot@cosmian.com (Théophile Brézot), chloe.hebant@cosmian.com (Chloé Hébant)

SSE scheme has ever been designed with the aim of being used under the highly concurrent workload achieved by current database implementations, and thus, all fail short in replacing the plaintext indexes of database systems.

## 1.1 Contributions

The contributions of the present paper are twofold. The first contribution is to study the impact of a concurrent execution model on the notions of correctness and security model of an SSE schemes. We found out that both of them need to be modified to overcome the challenges that arise from the existence of concurrent clients. The second contribution is to propose a new security model we called the Log-Security, that fixes a blind spot of the snapshot model that prevents it to accurately model real-life systems and has already been exploited in [GPT23]. We argue that this model – even though it cannot model a persistent adversary – may be of practical use in systems where an untrusted storage is used to store the state of the encrypted database or its backups. As an illustration, we use this model and the notion of linearizability [HW90], as it is well-established in the distributed system literature, to instantiate the first SSE scheme proven correct and secure in a concurrent execution setting.

### 1.1.1 Correctness of Distributed Systems

Correctness issues arise in a distributed system because concurrent operations may leave the system in an invalid state. The classic example is those of a distributed system implementing a counter in a shared memory location, such that each client increments the value of this counter by reading its value, incrementing it locally, then writing the result to the shared memory. Two concurrent increment operations may overwrite each-other's modification, leaving the system in an incorrect state. Much worst, when more complex objects are involved, concurrent operations may leave the system in an invalid state. Capturing the meaning of a correct concurrent execution is therefore not trivial. However, in the distributed-system literature, *linearizability* has emerged as a consensual definition [HW90]. In a nutshell, a system is said to be linearizable if any of its operational histories can be sequentially reordered without modifying the processes' view of the system.

### 1.1.2 Log-Security

The security guarantees of an SSE scheme are captured by a leakage function, and are proved with respect to a security model. Many such models have been proposed in the literature, the predominant one being those of the *persistent adversary* [CGKO06] in which the adversary resides in the server and has access to all queries made by the client(s). The more recent *snapshot model* [AKM19] considers an adversary only catching glimpses of the state of the memory, called snapshots, the number of which is defined by an additional parameter $m$. This situation naturally arises when the server storing the encrypted index is breached and its memory dumped, or when the server on which *back-ups* are stored is breached or owned. An SSE scheme is then said to be *breach-resistant* if the only information leaked is the size of the database at the time of each snapshot.

However, the snapshot model fails to satisfyingly model real-life systems. The first difficulty comes from the parameter $m$ since it seems difficult to establish an upper-bound on the number of snapshots an adversary can obtain. More importantly, this model does not take into account the fact that, in most database systems, persistency relies on a *Write Ahead Log* (WAL) in which are written all operations modifying its state. In case of a server failure, this log is replayed to restore the last correct database state known. The WAL invalidates the snapshot model because it gives an adversary access to the entire database history up to its first entry after the first retrieved snapshot. Logs have already been used to break SSE schemes such as the Queryable Encryption feature of MongoDB [GPT23]. In order to prevent such attacks, we introduce a new security model we called the *log-security*, that considers an adversary having access to the state of the entire database after each modification.

Incidentally, this security model switches its focus from per-client to per-server SSE operation, which seems to be a requirement for a security model to correctly capture security in concurrent execution. Indeed, concurrent client SSE operations may modify their behavior if they are related, which leaks information. However, concurrency can only be defined once protocol operations are considered in a white-box fashion: universal time is expensive to establish in a distributed system [Lam78], plus operations happening "at the same time" from a client point-of-view may not be concurrent since the network can retain the response event of the first one until after the second one terminates.

### 1.1.3 Findex

In order to recover the keyword-search functionality of encrypted database systems, we consider that an SSE scheme should allow concurrent clients to engage in a protocol with a shared server, have decent *performances*, be fully *dynamic* and *scalable*. Furthermore, clients should not be required to communicate with each other. This last requirement implies that no client-side stash can be used.

We present in this paper the first SSE scheme that meets these requirements that is proven both correct and secure in a concurrent execution setting. Throughout its design we focused on the database independency and reducing the number of client-server communications performed during a search. As a result, our construction features optimal search operations, indistinguishable insert and delete operations that are optimal in the absence of concurrent modification on the same keyword, and an asymptotic server-storage expansion of 13.3%[1].

## 1.2 Related Work

Correctness and termination guarantees in a concurrent setting have never been defined and studied in the SSE literature. As SSE schemes are single-user schemes [CGKO06, Bos16, BMO17, AKM19, MR23, AKM23]. They can thus store elements on the client side to increase efficiency and security. However, in a setting in which multiple non-communicating clients are allowed to modify the index, it is not possible to maintain up-to-date client-side information.

While *multi-client/user* Searchable Encryption schemes also exist in the literature [BDOP04, VRMO18, WC24, MCT$^+$24], they correspond to use-cases in which the data-owners are allowed to add and delete content and in which some readers must only be able to request the database. However, they necessitate public-key cryptography and a linear scan of the database.

Despite being designed in a single-client setting, the DLS scheme [AKM19] is the closest academic SSE to our construction. This scheme provides protections against data breaches as well as forward security. However, a crucial and significant difference with respect to Findex is that our scheme is designed to be stateless, concurrency-safe, and more resilient to data breaches, thanks to a stronger underlying security model against a snapshot adversary.

Up to our knowledge, the MongoDB Queriable Encryption [Mon22] is the only industrial solution deployed at large scale. However it suffers from issues related to overwriting which will be not allowed by our correctness definition.

Finally, Findex is significantly easier to deploy in real-world scenarios because it is the first SSE scheme considering the database system as a black box, enabling seamless access

[1]A Rust implementation of the scheme is available at Findex repository.

to all the inherent benefits of database systems such as concurrency and scalability. The main drawback of the black box database abstraction is that it implies no cryptographic server-side operations which is yet a widely used technique of most SSE schemes [CJJ+13, Bos16, PM21, PPSY21, MR23].

### 1.3 Organization of the Paper

We begin by defining a concurrent SSE scheme in Section 3 and formalizing its corresponding correctness. Since the classical security definition of SSE cannot be directly applied to a concurrent scheme, we introduce a new model called log-security, which records the actual in-memory changes. Next, we present a new generic architecture for SSE in Section 5. This section describes all the fundamental elements without the encryption part, allowing us to later give and prove our construction meets all the requirements stated in the introduction in Section 6. Finally, Section 7 explains some additional functionalities of Findex, along with an analysis of the efficiency and complexity, supported by benchmarks from our implementation.

## 2 Preliminaries

### 2.1 Notations

We denote $\lambda$ the security parameter used in the paper and denote $f_{C,S}(x; y)$ a potentially interactive protocol $f$ between a client $C$ with input $x$ and a server $S$ with input $y$.

In this article, we use a Haskell-like type description:

- (), the unit type, stands for "nothing";
- `Index kw v` designates an `Index` type parameterized by the two types `kw` and `v`;
- `Maybe T` = `Just T` | `Nothing` designates a `Maybe` type parameterized by `T`, that can either be `Just T` or `Nothing`;
- `search` :: `Memory a w` → `kw` → `Set v` designates a function `search` that takes as first argument a `Memory` parameterized by the types `a` and `w`, as second argument a value of type `kw`, and returns a set of values of type `v`;

In order to hide some context, we will use a `bundle` :: `Set Procedure` → `Object` that takes as argument a list of procedures, and returns an *object* that exposes these procedures as methods. For example, $\mathsf{bundle}\,[\mathsf{read}, \mathsf{write}]$ returns an object $o$ with an $o.\mathsf{read}$ and $o.\mathsf{write}$ methods.

### 2.2 Symmetric Searchable Encryption

We adjust at the margins the SSE definition of [CGKO06]:

**Definition 1.** An SSE scheme $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Insert}, \mathsf{Delete})$ is defined by:

- $\mathsf{Setup}_C(1^\lambda)$: executed by the first client, it takes as input a security parameter $1^\lambda$ and outputs a secret key $k$;
- $\mathsf{Search}_{C,S}(k, kw; \mathsf{EDB})$: it is a protocol between a client with input the secret key $k$ and a queried keyword $kw$; and the server with input the encrypted database $\mathsf{EDB}$. At the end of the protocol, the client receives a list $R$ of results;
- $\mathsf{Insert}_{C,S}(k, (kw, v); \mathsf{EDB})$: it is a protocol between a client with input the secret key $k$, and a keyword-value pair $(kw, v)$; and the server with input the encrypted database $\mathsf{EDB}$. At the end of the protocol, the server also stores $(kw, v)$ securely;

- $\mathsf{Delete}_{C,S}(k, (kw, v); \mathsf{EDB})$: it is a protocol between a client with input the secret key $k$, and a keyword-value pair $(kw, v)$; and the server with input the encrypted database $\mathsf{EDB}$. At the end of the protocol, $v$ has been removed from the results of $kw$.

## 2.3 Distributed Systems

We use the definition of distributed systems given in [Lam78]:

**Definition 2** (Distributed System)**.** A *distributed system* consists of a collection of distinct processes which are spatially separated and communicate by exchanging messages.

The predominant characteristic of such a system is that the communication medium is unreliable and thus the relation of time ordering is only partial in such a system. Each process is considered to be a sequential computation unit, and thus emitting totally ordered invocation and response *events*.

**Definition 3** (Operational History)**.** The operational history of an execution of a distributed system is the sequence of invocation (*inv*) and response (*resp*) *events* emitted and received by all processes in this system.

**Definition 4** (Operational Precedence)**.** The operational precedence $<_{\mathcal{H}}$ with respect to an history $\mathcal{H}$ is the partial order on the operations defined by:

$$op_1 <_{\mathcal{H}} op_2 \iff \begin{cases} resp(op_1) < inv(op_2) & \text{or} \\ resp(op_2) < inv(op_1) \end{cases}$$

where the event precedence $<$ is such that given two events $a$ and $b$, then $a < b$ if $a$ comes before $b$ in some process, if $a$ is the sending of some message by some process and $b$ the receiving of the same message by another process, or there exists an event $c$ such that $a < c < b$.

We can now provide the definition of linearizability for a distributed systems:

**Definition 5** (Linearizability)**.** A distributed system is said to be *linearizable* if for each history $\mathcal{H}$, there exists a sequential history $\mathcal{S}$ such that:

- $\forall p_i, \mathcal{H}_{|p_i} = \mathcal{S}_{|p_i}$, where the projection $\mathcal{H}_{|p_i}$ of an history $\mathcal{H}$ on a process $p_i$ is the subset of its events that involve $p_i$, which means that both executions are identical from the point of view of each process involved;
- $<_{\mathcal{H}} \subseteq <_{\mathcal{S}}$, which means that if $op_1 <_{\mathcal{H}} op_2$, then $op_1 <_{\mathcal{S}} op_2$;
- $\mathcal{S}$ is a legal history, which means it is correct according to the sequential specification of the shared objects, and, in particular, that this sequential execution is correct in the single client model.

Finally, efficiency in a distributed is measured in term of *progress* properties, among which (by increasing strength):

- the *lock-freedom* guarantees that given a set of concurrent legal client processes, at least one terminates. This property prevents situations in which a process dies without releasing a lock, effectively blocking all other processes. However, it does not preclude a process from indefinitely failing to preempt a resource due to monopolization of this resource by a group of concurrent threads. Such a phenomenon is called *starvation*;
- the *wait-freedom* guarantees that each correct process eventually terminates. The number of steps a process needs to perform however may vary, and depend on the actions of concurrent processes;
- the *bounded wait-freedom* guarantees that the existence of a bound $B$ on the number of steps required by a correct process to terminate.

## 3 SSE in a Concurrent Execution Setting

Since SSE are symmetric primitives, its classical definition only considers a single key holder, requesting an encrypted index. However, in order to use SSE schemes as a tool to secure database systems, we need to consider several *clients* concurrently engaging in a protocol with the server but *may not* communicate with each other. The server is thus the only shared object of the distributed system defined by an SSE instantiation.

### 3.1 Correctness

No previous work in the SSE literature has ever considered multiple clients, and the established definition of correctness requires the result of a request performed on the encrypted database to be equal to the result of the same request performed on the equivalent database in the clear. This definition however does not capture the notion of correctness in a distributed system. We therefore turn to the linearizability property to define the correctness of a concurrent execution of an SSE scheme. Note that linearizability is a strict generalization of the sequential correctness as any linearizable SSE algorithm remains correct in every possible sequential executions with respect to the sequential definition of correctness. In the following paragraphs, we further discuss the consequences of using linearizability as the correctness definition for a concurrent execution of an SSE scheme.

**No Data Loss.** Linearizability guarantees that no data can be lost since it would invalidate the legality of the related operational history (it would not respect its sequential specification): after a client indexes a new value under a given keyword, this value should be contained in any further search for this keyword until it is deleted from the index. To the best of our knowledge, beside being not formally defined, the MongoDB SSE scheme [Mon22] is not linearizable since it allows data loss whenever two clients attempt indexing a new value under the same keyword at the same time with probability $\frac{1}{c}$, where $c$ is their contention factor.

**Client Stash vs Client Cache.** Linearizability precludes SSE schemes from relying for correctness on a mutable client-side state, usually called *stash* in the SSE literature. Indeed, clients in our model are non-communicating. A client has therefore no way to perform a correct operation relying on a given stashed value as soon as an operation modifying this value was processed by a concurrent client. We stress that the trivial solution consisting in downloading the stash from the server, modifying it locally then uploading its modification does *not* preserve linearizability as it is exactly similar to the distributed counter example presented in Section 1.1.1, and that naive use of a locking mechanism to protect a read/write sequence does not preserve the lock-freedom property presented in Section 2.3. What is allowed, however, is for a client to maintain a local copy called *cache* of some server state, as long as its freshness it guaranteed upon use. Such a technique is broadly used in real-life systems to save the cost of performing some network calls.

Another broad use of the client stash in the SSE literature, is to act as a zero-cost, zero-leakage storage used to *delay operations* by stashing incoming ones while performing some others, stashed or dummy operations, effectively decorrelating the actual leakage from the requested operation. Such a scheme does not rely on the client stash for correctness reasons, but for security and concurrent clients could each maintain their own (different) local stashes without breaking the correctness of the operations performed. However, a difficulty still arises when considering the sequential definition of the requested operations: indexing a new value under a given keyword can terminate before indexing this value server-side. Therefore, any search operation on this keyword performed by a concurrent client would not result in finding this new value, which is illegal with respect to linearizability. Even worst, such a distributed system is not eventually consistent – which arose in the distributed system literature as a relaxation of the linearizability – since there is no guarantee that it will reach a consistent state after quiescence. Indeed, in the state-of-the-art FIX scheme [AKM23], a server-side modification can only be performed upon processing a new modification request. A quiescent systems has by definition no such incoming modification request and is therefore stuck in an inconsistent state. More generally, for any SSE scheme relying on delayed modifications, the client in charge of this operation could die after terminating but *before* it has already been performed on the server.

### 3.2 Security

In a concurrent execution setting, concurrent operations may modify their behavior, which is the source of a new leakage. For example, when considering a simple SSE scheme reminiscent of [AKM19] in which all values are stored in an EDX with respect to a token derived from a secret key, the keyword they are indexed under, and their position in the list of values indexed under it, the correctness of this scheme would rely on a client stash storing the current number of values indexed under each keyword. Considering for the sake of the example that it is possible to somehow store these counters server-side without degrading the security or losing the linearizability, this scheme is clearly forward secure since modification only add new encrypted value using random-like tokens. However, two concurrent modifications would derive the same token and one of them would fail to insert its value and retry with an updated counter value. This breaks the forward security since two concurrent insertions of values for the same keyword cannot be simulated without knowing which keyword they are targeting.

In order to be valid in a concurrent execution setting, a security model therefore needs not only to manipulate SSE operations, but also the sequence of communications between a given client and the server during the execution of the protocol defined by each SSE operation. Indeed, as described in the previous paragraph, concurrent operations may be the source of new leakages. Furthermore, it seems natural to let the adversary control the network as in the distributed systems that consider the network to be an adversarial scheduler.

## 4 Log-Security Model

As explained in the previous section, a sequential security proof that considers the decomposition of the SSE operations into memory operations would not be enough to model concurrent use-cases as they are not deterministic. To solve this major security issue, we introduce the *log-security* model in which the adversary has access to a copy of the encrypted memory after each modification instead of each SSE operation ($\mathsf{Search}$, $\mathsf{Insert}$ or $\mathsf{Delete}$). This model is thus closer to the RAM model used in ORAM schemes [Gol87, GO96] while featuring relaxed security constraints for efficiency reason.

**Write Ahead Log.** In many database systems, each modification of the memory, encrypted in our case, is stored in a Write Ahead Log (WAL). Its presence is essential because it allows the current state of the database to be reconstructed at any time, ensuring persistence. Therefore, disabling it is not an option. In case it exists, an adversary having access to a copy of the server memory, as assumed for example in the $m$-snapshot model [AKM19], would also receive the WAL. However, it has never been taken into account in the security model and has already been used as an attack vector against the MongoDB solution [GPT23].

**Our Model.** In our log-security model, the adversary is allowed to adaptively choose a sequence of SSE operations to be executed by concurrent clients. The number of such clients is not necessarily fixed. In fact, the adversary is free to instantiate new clients on-the-fly. To simulate the non-determinism arising from a concurrent setting, we give the adversary the additional power to select which memory operation to apply next among all incoming operations. Notice though that in practice, an adversary that has access to the WAL cannot interfere with this order.

Similarly to previous SSE security models, the adversary sends to the challenger a set $\mathsf{op} = \{op_i\}$ of operations in $\{\mathsf{Search}, \mathsf{Insert}, \mathsf{Delete}\}$, called *SSE operations* all along the game. In the real experiment, the role of the challenger is to play the role of the clients and the server as defined by the protocol. The challenger thus begins the execution of the operations in $\mathsf{op}$ in parallel on each client $i$, but before performing a server-memory operation (*e.g.* `read` or `write`), the challenger sends to the adversary the set $\{m\_op_{i,j}\}$ containing the next server-memory operations to be executed to complete each parallel SSE operation. As memory operations are atomic, the adversary can only select at most one of them, leaving the others pending. The challenger executes the memory operation selected by the adversary, sends a copy of the encrypted memory to the adversary upon modification, and continues the execution of all pending operations, regularly asking to the adversary which memory operation must be applied. After each reception of a new memory state, the adversary is allowed to add a new SSE operation. The game finishes when all SSE operations are completed or the adversary decides to terminate the game.

In the ideal experiment, instead of performing the protocol, the challenger executes each operation $m\_op_{i,j}$ on a plaintext database $\mathsf{DB}$, and gives $\mathcal{L}(m\_op_{i,j}, \mathsf{DB}_{i,j})$ to the simulator.

To ease the reading, we introduce a stateful function `schedule` corresponding to the first part of the challenger's execution as depicted in Figure 1: `schedule` corresponds to the client-server protocol interfacing with the server-memory in the real experiment and corresponds to the same protocol without any encryption interfacing with a plaintext memory in the ideal experiment.

- `schedule.new_op`$(op_i)$: Given in input an SSE operation $op_i$ or $\perp$, it adds $op_i$ to an internal list of SSE operations and outputs $\{m\_op_{i,j}\}$ the list of possible memory operations that must be done to continue the protocol;
- `schedule.next_m_op`$(m\_op_{i,j})$: Given in input the result of the memory operation on the memory, it removes $m\_op_{i,j}$ from the list of memory operations that must be done and updates its internal states.

One can now formally define the log security model illustrated in Figure 1:

**Definition 6** (Log-security)**.** Let $\Sigma$ be an SSE scheme, $\mathcal{L}$ a stateful leakage function, $\mathcal{A}$ an adversary, `schedule` as defined above. $\Sigma$ is said $\mathcal{L}$-log secure if there exists a PPT simulator $\mathcal{S}$ such that for all PPT adversaries $\mathcal{A}$,

$$|\Pr[\mathbf{Real}^{\Sigma}_{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S}}(1^\lambda) = 1]| \leq \mathsf{negl}(1^\lambda).$$

where $\mathbf{Real}^{\Sigma}_{\mathcal{A}}$ and $\mathbf{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S}}$ are defined in Figure 2.

## 5 A Modular SSE Architecture

The aim of this section is to propose an architecture that simplifies concurrency management and inherits from the good properties of existing DataBase Management Systems (DBMS) like scalability, persistence and replication. We first recall the properties we claim to be required from a production SSE that aims at recovering the lost keyword-search feature of an encrypted database:

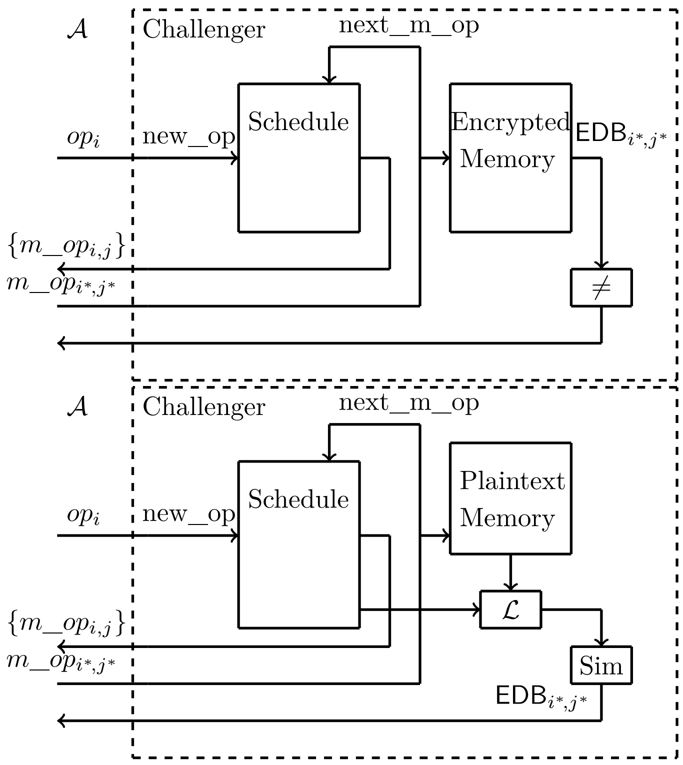

**Figure 1:** Representation of the log-model: above the **Real** game, below the **Ideal** game.

- decent performances: contrary to the line of works focusing on page efficiency [MR22, MR23, MCDP24], we claim, in tradition with the literature of distributed systems, that in a real-life distributed system, the performance of a protocol operation is dominated by the number and size of the communications, and the amount of client synchronization that are required. From this fact, we deduce as guiding principle that operations should thrive to achieve wait freedom – in particular the use of lock should be avoided, and that the number of sequential communications involved in a protocol should be minimal;
- light client requirements: typical database clients are light, which means that the amount of memory required to perform a search operation should be independent of the size of the database;
- other guarantees: since no call to the database will be performed directly as one needs encryption, the SSE can be the limiting factor of the equation, in term of both scalability, persistency and availability. Ideally, an SSE should be as scalable, persistent, and available as the database of which it indexes the data.

## 5.1 Server as a Memory

State-of-the-art DBMSs are very complex objects. A great deal of this complexity is due to the need to provide efficient operations sacrificing neither scalability, persistency nor availability. In order to build upon this work, we claim that a realistic SSE construction should restrict the server to a DBMS. In our architecture, we go even further by totally abstracting this DBMS behind a transactional memory implementing the following *Abstract Data Type* (ADT) `Memory a w`:

$\mathbf{Real}^{\Sigma}_{\mathcal{A}}(1^\lambda)$:

1. The challenger runs $k \leftarrow \mathsf{Setup}(1^\lambda)$.
2. While $\mathcal{A}$ does not output a bit $b \in \{0, 1\}$:
   (a) The adversary selects $m\_op_{i^*,j^*} \in \{m\_op_{i,j}\} \cup \{\perp\}$,
   (b) If $m\_op_{i^*,j^*} = \perp$, the challenger ignores the step, otherwise it runs `schedule.next_m_op`$(m\_op_{i^*,j^*})$ and executes the operation on the encrypted memory which produces the snapshot $\mathsf{EDB}_{i^*,j^*}$ and sends $\mathsf{EDB}_{i^*,j^*}$ to the adversary if the result differs from the previous one.
   (c) The adversary chooses an SSE operation $op_i = (f_i, x_i)$ with $f_i \in \{\mathsf{Search}, \mathsf{Insert}, \mathsf{Delete}, \perp\}$ and $x_i$ an input of $f_i$ and sends $op_i$ to the challenger,
   (d) The challenger executes $\{m\_op_{i,j}\} \leftarrow$ `schedule.new_op`$(op_i)$ and sends $\{m\_op_{i,j}\}$ to the adversary.
3. The experiment outputs the bit $b$ from the adversary.

$\mathbf{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S}}(1^\lambda)$:

1. While $\mathcal{A}$ does not output a bit $b \in \{0, 1\}$:
   (a) The adversary selects $m\_op_{i^*,j^*} \in \{m\_op_{i,j}\} \cup \{\perp\}$,
   (b) If $m\_op_{i^*,j^*} = \perp$, the challenger ignores the step, otherwise it runs `schedule.next_m_op`$(m\_op_{i^*,j^*})$, executes the operation on the plaintext memory which produces the snapshot $\mathsf{DB}_{i^*,j^*}$ and sends $\mathcal{L}(m\_op_{i^*,j^*}, \mathsf{DB}_{i^*,j^*})$ to the simulator. The simulator outputs $\mathsf{EDB}_{i^*,j^*}$ sent to the adversary.
   (c) The adversary chooses an SSE operation $op_i = (f_i, x_i)$ with $f_i \in \{\mathsf{Search}, \mathsf{Insert}, \mathsf{Delete}, \perp\}$ and $x_i$ an input of $f_i$ and sends $op_i$ to the challenger,
   (d) The challenger executes $\{m\_op_{i,j}\} \leftarrow$ `schedule.new_op`$(op_i)$ and sends $\{m\_op_{i,j}\}$ to the adversary.
2. The experiment outputs the bit $b$ from the adversary.

**Figure 2:** Experiments for the log-security model

**Definition 7** (Memory ADT)**.** A `Memory a w` exposes two atomic operations:

- `b_read` :: `List a` $\rightarrow$ `List w` returns the list of words $[w_i]$ stored at the addresses $[a_i]$;
- `g_write` :: (`a` $*$ `Maybe w`) $*$ `Set` (`a` $*$ `w`) $\rightarrow$ `Maybe w` takes as arguments a *guard binding* $(a_g, w_g^{ref})$ and a set of new bindings $\{(a_i, w_i)\}$; it writes these bindings to memory if the word $w_g$ currently bound to $a_g$ is $w_g^{ref}$, and returns $w_g$.

Note that this abstraction can be implemented over most DBMS supporting *transactions*. For example, Redis allows atomic scripting using an embedded Lua interpreter, and PostgreSQL language allows conditional expressions to be used in transactions. Restricting the range of possible transactions is not essential to our scheme. It however has the advantage to simplify the implementation and the security proof. On systems that do not support transactions but expose a *Compare and Swap* (CAS) operation, the *g_write* operation can be implemented using the $k$-CAS algorithm [GKMZ20] and the *b_read* operation using the double-collect technique [AAD+93], at the cost of the wait-freedom.

This memory abstraction can be viewed as a dictionary (DX) as defined in the Structured Encryption (STE) [CK10], we argue, however, that this abstraction is a better fit as it exposes its similarity with Software Transactional Memories (STE) and the Asynchronous Shared Memory (ASM) model.

## 5.2 The Encryption Layer

The external memory implemented over a DBMS presented in the previous section should only stores encrypted data. Since the server is reduced to an external memory, all encryption is performed client-side. However, one could wish to reason about an SSE construction in the same way one reasons about a data-structure. To this aim, we propose to clearly extract the cryptographic transformations from the client protocol to an *encryption layer*. This layer can formally be defined as an endomorphism on the space of memories that transforms a plaintext memory `Memory a w` into an encrypted memory `Memory a`$^*$ `w`$^*$, and thus allows to expose a plaintext memory interface to the rest of the client-side computation. In the rest of this paper, we use *virtual address* to design an address in the plaintext memory abstraction and *actual address* to design an address in the encrypted memory abstraction.

The encryption layer could be implemented by the algorithm presented in Figure 3, where $\mathcal{E}$:: `k` $\rightarrow$ (`W` $*$ `A` $\rightarrow$ `W`$^*$) $*$ (`W`$^*$ $*$ `A` $\rightarrow$ `Maybe W`) is an encryption scheme with auxiliary data that uses the actual address as auxiliary data to encrypt the memory words, and $\Pi$:: `k` $\rightarrow$ (`A` $\rightarrow$ `A`) $*$ (`A` $\rightarrow$ `A`) is a keyed random permutation on the address space $A$. Note that if $\mathcal{E}$ is not deterministic, a cache needs to be added so that a decrypted value $w_g^*$ must be re-encrypted identically, should it be given as the next $w_g^{ref}$ value. However, it is not displayed in the figure. The cryptographic primitives chosen in this Figure 3 could be modified to enforce different guarantees, for example:

- by adding a signature scheme [DH76], one can define a set a writers as a subset of readers. Any signature scheme could be used. For example, a group signature [Cv91, BMW03] scheme allows a signer to be anonymous inside a group of signers (to add new keyword or values in the SSE scheme without revealing the origin of the data even to the readers) while at the same time, an authority can reveal the identity in case of abuse;
- by using public-key schemes as $\mathcal{E}$ scheme, and in particular ABE ones [GPSW06, BdPP24], one opens the door to rights for the readers as in MC-SE [BDOP04, VRMO18, WC24, MCT+24, AC17, RW22]. In that case, all the data stored in the database must be authenticated as keyword guessing attack can be damaging [HL17].

```
function make_encryption_layer($(k, m^*)$)
    $k_\pi \leftarrow \mathsf{KDF}(k||0)$
    $k_\epsilon \leftarrow \mathsf{KDF}(k||1)$
    $(\pi, \pi^{-1}) \leftarrow \Pi(k_\pi)$
    $(\epsilon, \epsilon^{-1}) \leftarrow \mathcal{E}(k_\epsilon)$

    procedure b_read($([a_i])$)
        $[a_i^*] \leftarrow [\pi(a_i)]$
        $[w_i^*] \leftarrow m^*.\mathsf{b\_read}([a_i^*])$
        $[w_i] \leftarrow [\epsilon^{-1}(w_i^*, a_i^*)]$
        return $[w_i]$
    end procedure

    procedure g_write($((a_g, w_g^{ref}), \{(a_i, w_i)\})$)
        $a_g^* \leftarrow \pi(a_g)$
        $w_g^{ref,*} \leftarrow \epsilon(w_g^{ref}, a_g^*)$
        $\{(a_i^*, w_i^*)\} \leftarrow \mathtt{shuffle}\,(\{(\pi(a_i), \epsilon(w_i, \pi(a_i)))\})$
        $w_g^* \leftarrow m^*.\mathsf{g\_write}((a_g^*, w_g^{ref,*}), \{(a_i^*, w_i^*)\})$
        $w_g \leftarrow \epsilon^{-1}(w_g^*, a_g^*)$
        return $w_g$
    end procedure

    return bundle(b_read, g_write)
end function
```

**Figure 3:** Implementation of an Encryption Layer

## 5.3 SSE as an Index

We recall that the goal of an SSE scheme is to implement an index with added security guarantees, which functionality is captured by the following ADT:

**Definition 8** (Index ADT)**.** An `Index kw v` exposes three procedures:

- `search` :: `kw` → `Set v` returns the set of values $\{v_i\}$ bound to a keyword $kw$ in the index;
- `insert` :: `kw` ∗ `Set v` → () binds a set of values $\{v_i\}$ to a keyword $kw$ in the index;
- `delete` :: `kw` ∗ `Set v` → () unbinds a set of values $\{v_i\}$ from a keyword $kw$ in the index.

Previous publications in the STE line of works focused on the multimap (MM) and encrypted multimap (EMM) ADTs [CK10, KMO18], however, we note that it differs in the fact that it associates keywords with *sets* of values rather than tuples of values of varying lengths, and thus can formally be described as a map from the set of keywords to the power-set of the possible values. Moreover, an MM is formally defined as a data-structure rather than an abstract data-type and is strongly oriented toward an associative array.

We stress once again that now that the encryption has been extracted in a separate layer, one is free to directly reuse all the constructions in the algorithm and distributed system literature. As an illustration, we present in the next section a simple implementation of the index ADT using vectors stored in a perfect hash-table, that is proven correct in a concurrent execution setting and secure in the log-security model.

# 6 Findex

## 6.1 Index ADT implementation

The natural way to implement an index is to use a hash-table, the difficulty then resides in the management of hash collisions. However since in our model allocations have virtually zero cost, we can use the entire memory space to guarantee the absence of collision with overwhelming probability. Since an index stores sets of values, we store vectors as values of our perfect hash-table. In this section, we present the relevant ADTs and their implementations.

**Definition 9** (Vector ADT)**.** A `Vector v` exposes two procedures:

- `read` :: () → `List v` returns the values $[v_i]$ stored in the vector.
- `push` :: `List v` → () pushes new values $[v_i]$ at the end of the vector.

We implicitly implement the perfect hash-table using a deterministic address derivation: typically a hash `hash` :: `kw` → `a` of this keyword that only needs to guarantee a good-enough collision resistance. Our implementation of the vector is given in Figure 4. `make_vector` takes as argument the vector address $a$ and an implementation $m$ of a plaintext memory abstraction. It returns an object with the two `read` and `push` methods. We use two helper functions to manage headers and keep the pseudo-code free from implementation details:

- `inc_cnt` :: `Maybe Header` ∗ `Int` → `Header` takes as argument an optional base header and an increment, and returns the header built by adding this increment to the internal counter of the base header or a newly instantiated header if none was given;
- `get_cnt` :: `Maybe Header` → `Int` returns the internal counter of the given header or 0 if none was given.

Once we have implemented a vector, the implementation of the index ADT follows straightforwardly. We give a cache-less implementation of this ADT in Figure 4 that relies on two helper functions:

- `decompose` :: (`Insert` | `Delete`) ∗ `Set v` → `List w` that takes as input Insert or Delete, and a set of values, and returns the list of memory words $[w_i]$ encoding the application of this operation on these values;
- `recompose` :: `List w` → `Set v` that takes as input a list of memory words and returns the list of values obtained by applying all the operations encoded in these words.

These implementations are stripped to the bare minimum for the sake of the discussion. The careful reader would have noticed that our vector implementation fails to respect the homogeneity of the memory words since it indifferently stores its header and the values it was given: our actual implementation needs to serialize both header and values into memory words of the same length (to guarantee their indistinguishability). Additionally, since *Remote Procedure Calls* (RPC) are used to access the memory, such accesses are costly. Our implementation therefore uses a cache to store the vector headers: this allows performing all index operations with a single RPC in case no concurrent modification on the same keyword was perform. Another way to see it is that each client maintains a local stash similar to the one used in [AKM19], while a global stash is stored server-side. Client operations perform optional mutations on this global stash (in case of Index :: insert and Index :: delete), and synchronize the entry of their local cache associated to the keyword being operated on with the respective entry of the global stash.

```
function make_vector($a, m$)
    procedure push($\{v_i\}_{i \in [1,n]}$)
        procedure push_with($h_o$)
            $h_n \leftarrow$ inc_cnt($h_o, |\{v_i\}|$)
            $h_c \leftarrow m$.g_write(
                    $(a, h_o)$,
                        $\{(a, h_n)\} \cup \{(a +$
get_cnt$(h_o) + i, v_i)\}_i$)
            if $h_o \neq h_c$ then
                push_with($h_c$)
            end if
        end procedure
        push_with(Nothing)
    end procedure

    procedure read()
        $h \leftarrow m$.b_read($[a]$)
        return        $m$.b_read($[a$        $+$
$i]_{i \in [1;\text{get\_cnt}(h)]}$)
    end procedure

    return bundle(read, push)
end function
```

```
function make_index($m$)
    procedure search($kw$)
        $\vec{v}_{kw} \leftarrow$ make_vector(hash($kw$), $m$)
        return recompose($\vec{v}_{kw}$.read())
    end procedure

    procedure insert($kw, \{v_i\}$)
        $\vec{v}_{kw} \leftarrow$ make_vector(hash($kw$), $m$)
        $\vec{v}_{kw}$.push(decompose(Insert, $\{v_i\}$))
    end procedure

    procedure delete($kw, \{v_i\}$)
        $\vec{v}_{kw} \leftarrow$ make_vector(hash($kw$), $m$)
        $\vec{v}_{kw}$.push(decompose(Delete, $\{v_i\}$))
    end procedure

    return bundle(search, insert, delete)
end function
```

**Figure 4:** Implementations

**Laziness.** The decompose and recompose helpers allow to implement a *lazy* delete operation on the index: both Index :: insert and Index :: delete push new values to the underlying vector. These values only need to be encoded along with their associated operation into memory words. Therefore, making an operation lazy consists in registering it in some vector, which can then be viewed as a log of committed operations. More generally, as long as *application* forms a monoid on the set of logged operations, laziness allows to transform any dynamic construction storing *sequences* of values into a construction performing arbitrarily complex operations. Readers then only need to *left-fold* the logged operations. If moreover logging only requires a single insertion, both schemes have the same leakage. Increased server-side storage and client-side search computation seem to be a small price to pay for this transformation since these are not limiting factors in our model, and that one can enforce an upper bound on this overhead by regularly applying the log server-side.

Lazy operations also come handy when they avoid the need for clients to fetch data before manipulating it. For example, in the concrete example of the delete operation, a client attempting to delete a given value from the set of values indexed under can just insert a negation of this value. This simplification may prevent important leakages. Indeed, in the simple scheme exposed in Section 3.2, eager deletion would not be forward private while lazy deletion would.

Note that in case the set of logged operations forms a commutative group for application then the same transformation can be applied on any index implementation, and thus any SSE scheme. Sadly this is not the case for the insert and delete operations.

## 6.2 SSE ADT implementation

Finally, we present here the Findex implementation of the SSE ADT. Briefly, we instantiate the scheme by passing it some connection to a server implementing the Memory ADT. We instantiate an object $m^*$ that implements the same Memory ADT. Then for each one of the procedures $\mathsf{Search}(k, kw)$, $\mathsf{Insert}(k, kw, v)$ and $\mathsf{Insert}(k, kw, v)$, we first instantiates an encryption layer, which creates an object $m$ implementing a plaintext memory interface. We stress that any call to one of its methods actually performs an *encrypted*, *server-side* operation. The client then instantiate an object index that implements the index ADT as defined previously, and calls its appropriate method.

```
function make_findex($connection$)
    $m^* \leftarrow$ make_memory($connection$)

    procedure Setup($1^\lambda$)
        $k \xleftarrow{\$} \{0,1\}^\lambda$
        return $k$
    end procedure

    procedure Search($k, kw$)
        $m \leftarrow$ make_encryption_layer($k, m^*$)
        index $\leftarrow$ make_index($m$)
        return index.Search($kw$)
    end procedure

    procedure Insert($k, kw, v$)
        $m \leftarrow$ make_encryption_layer($k, m^*$)
        index $\leftarrow$ make_index($m$)
        index.Insert($kw, \{v\}$)
    end procedure

    procedure Delete($k, kw, v$)
        $m \leftarrow$ make_encryption_layer($k, m^*$)
        index $\leftarrow$ make_index($m$)
        index.Delete($kw, \{v\}$)
    end procedure

    return bundle(Setup, Search, Insert, Delete)
end function
```

## 6.3 Concurrent Behavior

We can now prove all the claimed Findex properties.

**Theorem 1.** *Findex is a linearizable scheme that exposes lock-free insert and delete operations and a bounded-wait-free read operation.*

*Proof.* Linearizability is easily obtained in Findex thanks to the use of a transactional memory: the linearization point for write operations is the success of the transaction while the linearization point for read operations is the fetching of the current header value. Indeed, transactional writes preserve what we call the vector invariant ($I_v$): the value of the counter stored in the header is equal to the number of values stored in the vector, and those values are stored in-order, in a contiguous memory area. Therefore, reading a counter value allows reading all vector values written by operations up to the last to increase this counter.

Write operations loop on a failing to commit their transaction, which happens when the client is not aware of the up-to-date header value. In a sequential context – if headers are cached – modifications can thus be performed with a single communication. In a concurrent context, this freshness of the cached value cannot be guaranteed since a concurrent modifications may occur. In that case, the client receives the current state of the header, and communicate an updated transaction to the server. Now, given a group of client processes concurrently modifying the same vector and starting with an up-to-date knowledge of the header state, at least one of them commits its transaction. Others retry with an updated transaction, and at least one of them succeeds, etc. Therefore, given a group of legal, concurrent writers, at least one terminates in a finite number of states which makes the write operation *lock-free*. Write operations are not wait-free since they do not prevent starvation of slow processes.

Read operations are trivially bounded-wait-free since reading is not a fallible operation, and require exactly two sequential communications. □

We insist on the fact that the log-security model only considers sources of leakage such as back-ups and WALs. These two sources cannot be avoided since they are used to guarantee data persistency. In many databases systems, other events like search queries can be logged. However, these log are only used for monitoring and can be disabled – which needs to be done to stay in the log-security model.

Moreover, Delete and Insert are identical as the operation is simply encoded into the values during the decompose/recompose step. During an *update* (i.e. Insert or Delete), several scenarios can appear generating more or less leakages.

If the keyword to be updated is not already present in the database and no other concurrent calls are made, the update produces only one snapshot of an encrypted memory that has two additional bindings. This comes from the fact the writing in the memory `g_write` is implemented through a transaction. Hence, an adversary cannot receive an intermediate step where the counter of elements in the header of a keyword does not correspond to the current number of stored elements for that keyword in the database.

If the keyword to be updated is already stored, an adversary learns that a binding has been modified and an additional one appears. It can deduce the two belongs to the same keyword. More generally, the adversary can estimates the *volume* of each stored keyword, the number of binding related to a keyword.

**Theorem 2.** *Findex is a correct concurrent SSE scheme and if $\mathcal{E}$ is a secure symmetric encryption scheme and $\pi$ is a pseudo-random permutation, then Findex is $\mathcal{L}_F$-log secure where $\mathcal{L}_F$ is a stateful function tracking the change on* DB *during the memory operation m_op: $\mathcal{L}_F(m\_op, \mathsf{DB}') = (\ell_{update}, \ell_{add})$ with $\ell_{update}$ is $\perp$ or a representative of the updated binding and $\ell_{add}$ is the number of added bindings from the last* DB *received.*

*Proof.* As Findex is linearizable, Findex is a correct SSE scheme.

For the security, we are interested by all the server writings as the goal of the simulator will be to produce, for each memory operation *m_op* chosen by the adversary, a sequence of snapshots, one snapshot per server-memory modification, indistinguishable from a real execution of the protocol where the *m_op* come from SSE operations *op*. One first analyzes Findex.Insert to highlight those writings. Informally, $\mathsf{Findex.Insert}(k, (kw, v))$ corresponds to the three steps:

1. $m_{ptx} \leftarrow \mathsf{EL.decrypt}(k, m_{ctx})$
2. $\mathsf{index} \leftarrow \mathsf{make\_index}(m_{ptx})$
3. $\mathsf{index.Insert}(kw, v)$

Hence, all the writings on the server side come from writings in the index object followed by the EL transformations. While this structuring was originally intended to simplify the implementation of the SSE, the writings in the index object will correspond to the challenger part while the EL transformations will be replaced by random values by the simulator in the ideal experiment.

*Real experiment.* The challenger follows the protocol playing the role of the client and the server. Before each writings in the encrypted memory, the challenger asks to the adversary which one must be performed. This generates a copy of the memory and given to the adversary if the memory was changed.

*Ideal experiment.* The challenger executes the same protocol without the encrypted layer. Before each writings in the plaintext memory, the challenger asks to the adversary which one must be performed. If the memory changed, $\mathcal{L}_F(m\_op, \mathsf{DB}') = (\ell_{update}, \ell_{add})$ is given to the simulator which produces $\mathsf{EDB}$ as follows:

- if $\ell_{update} = \perp$, it adds $\ell_{add}$ bindings $(a_i, c_i)$ with $a_i, c_i \overset{\$}{\leftarrow} \{0,1\}^\lambda$,
- else, it replaces $(a_{\ell_{update}}, c_{\ell_{update}})$ by $(a_{\ell_{update}}, c^*)$ and adds $\ell_{add}$ bindings $(a_i, c_i)$ with $c^*, a_i, c_i \overset{\$}{\leftarrow} \{0,1\}^\lambda$,

Hence, the advantage of the adversary distinguishing between the two games is upper bounded by the advantage of an adversary against $\mathcal{E}$ and $\pi$. $\square$

## 7 Performance Evaluation

It is trivial to see from the Figure 4 that write operations involve $O(M)$ communications, that read operations involve one $O(1)$ and one $O(V)$ communications and that both read and write operations involve a constant number of communications in a sequential setting where $M$ corresponds to the number of modifications and $V$ the number of values. Furthermore, we have demonstrated in Section 6.3 that our scheme was linearizable and that read and write operations are respectively bounded-wait-free and lock-free. We therefore validate all the requirements established in Section 5. We now demonstrate the claims given in Section 6.

### 7.1 Search optimality

**Theorem 3.** *Let $n(t)$ be the number of bits used to store an information $\mathcal{I}(t)$ without loss. In the asynchronous shared-memory model, the minimum number of client-server communications of size $O(n(t))$ after which a client can read $\mathcal{I}(t)$ from the memory is two.*

*Proof.* First off, note that $n(t)$ must depend on the entropy of $\mathcal{I}(t)$ (source coding theorem). Then recall that in the asynchronous shared-memory model, clients do not communicate and no assumption can be made on the relative speed of the client processes. Therefore no client can know for sure the entropy of $\mathcal{I}(t)$ (the knowledge of the entropy of $\mathcal{I}(t')$ for some $t' < t$ may have been deprecated by a concurrent addition to this set). For the same reason, no upper bound can be assumed either. For these reasons, the number of memory words from which to read cannot be known without communicating with the server and thus a single communication cannot allow to read $\mathcal{I}(t)$ with certainty, while a second can. $\square$

**Corollary 1.** *In a setting in which the server does not perform any operation other than memory operations, the minimum number of client-server communications needed to fetch the results bound to a given keyword using $O(V)$ operations is two.*

### 7.2 Benchmarks

Our implementation is written in Rust[2], and can therefore be compiled to highly optimized binaries that takes advantage of hardware features like AES instructions. It can also be compiled to a WASM target, and run in a browser. We used AES256-XTS as encryption

[2] https://github.com/Cosmian/findex/releases/tag/eprint.

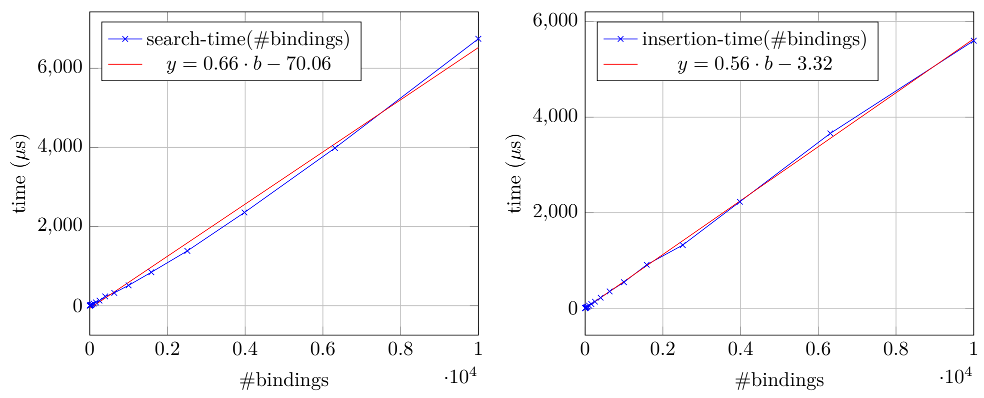

**Figure 5:** (Left) Client-side computation time (in $\mu$s) for a single-keyword search, given the number of bound one-word values. (Right) Client-side computation time (in $\mu$s) for a single-keyword insert, given the number of bound one-word values.

scheme $\mathcal{E}$ and AES256 as keyed pseudo-random permutation $\pi$ in the encryption layer, and the memory was implemented in-memory using a `std :: collections :: HashMap`. In particular, the benchmarks presented in this section do not include any communication time. In real use-cases, the network time would need to be added, accordingly to the number of round-trips discussed in the previous sections. All timings were obtained by benchmarking on an Intel(R) Xeon(R) CPU @ 2.30GHz.

The Figure 5 (Left) displays the client-side computation time, in micro-seconds, involved by searching for a single keyword, and varies with the number of values bound to this keyword. It was computed by measuring the total time for such operations, to which was removed the time needed to search for the according number of words from the memory. A linear regression is presented, that does not take into account searches with less than 10 results, in order not to be influenced by the constant overhead. We therefore demonstrate that time varies linearly with the number of bindings: Findex search operations really are $O(V)$.

The Figure 5 (Left) is similar to the Figure 5 (Right), but is relative to insertions: it displays the client-side computation time, in micro-second, needed to insert a given number of bindings to the index. Similarly to the search timings, these timings are linear with respect to the number of bindings to add, which proves once again that Findex insert operations are $O(M)$.

Finally, the Figure 6 displays the overhead arising from conflicts between concurrent writers. These timings were measured by benching concurrent clients inserting 100 bindings to the same keyword, to which were removed the times needed for the same number of clients to insert the same number of bindings on different keywords. As noted before, no latency was added to communications. This figure once again shows a linear dependency, which can be explained by the fact that Findex's conditional write essentially *linearizes* concurrent clients. Indeed, the total time needed to finalize concurrent insertions is equal to the insertion time of the client that last commits its modification. If latency variance is neglected, all clients synchronously retry committing their modification after a failure. Since after each failure, one client has succeeded, the number of times the last client to commit its modification has retried is therefore equal to the number of concurrent clients. Following this reasoning, a first approximation on the time a concurrent batch of insertion terminates must be $n \times t_m$, where $n$ is the number of concurrent clients, and $t_m$ the time to insert $m$ bindings. In our experiment, each client inserts 100 bindings, which gives $t_m = 56\mu$s by using the coefficient of the linear regression in Figure 5 (Right), is about the same as the coefficient of the linear regression in 6 and validates our hypothesis.

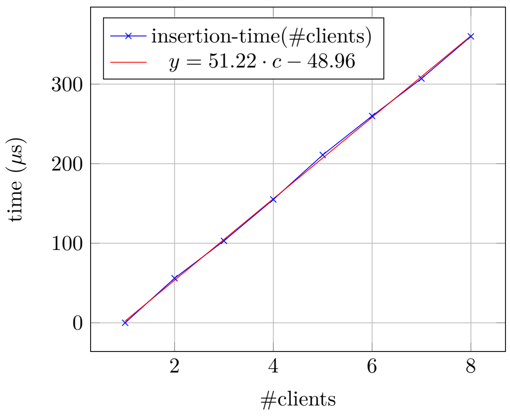

**Figure 6:** Concurrency overhead (in $\mu$s) for adding 100 bindings on the same keyword, given the number of concurrent clients.

## 7.3 Expansion of the index

The first source of overhead our scheme introduces is the *storage of vector headers*. This cost is static and thus amortized as the size of the indexed data increases. The second source of overhead is the *encoding* of the operations into memory words. This cost depends on the number of both encoded operations and used words. The dependency on the number of operations can be relaxed by considering a compacted state, which leaves an encoding overhead that must be roughly proportional to the amount of encoded data. The third and last source is the *encryption overhead*. Since encryption is performed on a per-word basis, its cost is also roughly proportional to the amount of data stored in the index. Additionally, the server-side memory may be implemented using a key-value store, which implies *addresses* are stored alongside their associated word and thus incur an overhead that is similar to the encryption overhead.

Our implementation uses one byte of metadata per encoded word, AES256-XTS as encryption scheme, and a server-side memory based on a key-value store. Since AES-XTS does not introduce any overhead, our implementation has an overall 17-byte overhead per word. In order to fully use the metadata byte and to optimize our scheme to store 16-byte values (size of DB UIDs), we use eight 16-byte blocks per words. Considering a compacted state, the asymptotic expansion of the index can neglect the header and padding overheads, and amounts to 13.3%.

## 7.4 Limitations

Firstly, as discussed in the previous section, a storage overhead must be paid for each modification. The insertion overhead is due to the padding of the last memory word and to the encoding of the operation, while deletions are pure overhead since they are stored and performed client-side. In storage-critical applications, it is possible to use the padding in the vector headers to store a *sparsity budget*. Each operation uses this budget as described by some rule (deletions should cost more than insertions) and clients need to perform a compact operation of the target vector in case the associated budget does not allow to carry on with the intended operation.

Secondly, Findex modifications are only lock-free. This means that this scheme does not prevent some client from eventually starving, which can happen, for example, if other threads have a faster connection to the server and keep modifying bindings associated to the same keyword, and in which that case the starving client cannot keep-up and its attempts to modify some bindings are constantly rejected by the server.

Finally, Findex is not forward-secure. Indeed, with only snapshots from the log-security model, an attacker can keep track of the header modifications which in turn can be linked to a particular keyword through a statistical attack. It is also possible to track how many memory words are allocated to the storage of the bindings of each keyword and deduce an approximation on the indexed volumes. In the literature, forward security is usually obtained through a client-side storage (to store the volume of each keyword) and a one-way function (so that only the client is able to compute the next line to be added for the next modifications and not the server). However, application of this technique is incompatible with performances and correctness in the concurrent setting. It turns out that achieving forward security in this setting while preserving $O(1)$, wait-free searches is not trivial, and is left as an open problem.

## 8 Conclusion

In this work, we introduced the first formal definition of SSE correctness in a concurrent environment together with a new security model stronger than the snapshot model that may be of independent interest.

We have demonstrated it was possible to design an SSE scheme that is both log-secure, correct and efficient in the concurrent setting. Furthermore, assimilating the server to a transactional memory in the aim to make our scheme database independent by default allowed for a great deal of simplifications in the client-side implementation, and led to a modular design: changing the cryptographic primitives used in the encryption layer allows introducing private rights, while a change in the choice of data-structure used to implement the index ADT allows modifying its access pattern and thus its security characteristics and finally, being able to expose a plaintext transactional memory to the client-side implementation allows to reuse many of the classic distributed system techniques.

The design space therefore seems to be large, and finding a local minimum along the three axes that are security level, communication, and progress, is not trivial. In particular, the existence of a scheme that is both forward-secure, wait-free and with $O(1)$ operations, is an open problem.

Finally, our scheme proved to be fast, scalable and easy to deploy on top of any database system and we find its security guarantees to be credible enough to use it as a solution for secured production-grade offloaded storage. We hope this work will open new doors for practical SSE applications, and pave the way for a shift in the academic study of SSE schemes toward the support of a concurrent setting and security models compatible with real-life applications, on par with our novel log-security.

## References

[AAD+93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, sep 1993. doi:10.1145/153724.153741.

[AC17] Shashank Agrawal and Melissa Chase. FAME: Fast attribute-based message encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 665–682. ACM Press, October / November 2017. doi:10.1145/3133956.3134014.

[AKM19] Ghous Amjad, Seny Kamara, and Tarik Moataz. Breach-resistant structured encryption. *PoPETs*, 2019(1):245–265, January 2019. doi:10.2478/popets-2019-0014.

[AKM23] Ghous Amjad, Seny Kamara, and Tarik Moataz. Injection-secure structured and searchable symmetric encryption. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VI*, volume 14443 of *LNCS*, pages 232–262. Springer, Heidelberg, December 2023. doi:10.1007/978-981-99-8736-8_8.

[BDOP04] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 506–522. Springer, Heidelberg, May 2004. doi:10.1007/978-3-540-24676-3_30.

[BdPP24] Théophile Brézot, Paola de Perthuis, and David Pointcheval. Covercrypt: An efficient early-abort kem for hidden access policies with traceability from the ddh and lwe. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *Computer Security – ESORICS 2023*, pages 372–392, Cham, 2024. Springer Nature Switzerland.

[BMO17] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1465–1482. ACM Press, October / November 2017. doi:10.1145/3133956.3133980.

[BMW03] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 614–629. Springer, Heidelberg, May 2003. doi:10.1007/3-540-39200-9_38.

[Bos16] Raphael Bost. $\Sigma o\phi o\varsigma$: Forward secure searchable encryption. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1143–1154. ACM Press, October 2016. doi:10.1145/2976749.2978303.

[CGKO06] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 79–88. ACM Press, October / November 2006. doi:10.1145/1180405.1180417.

[CJJ$^+$13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for Boolean queries. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373. Springer, Heidelberg, August 2013. doi:10.1007/978-3-642-40041-4_20.

[CK10] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 577–594. Springer, Heidelberg, December 2010. doi:10.1007/978-3-642-17373-8_33.

[Cv91] David Chaum and Eugène van Heyst. Group signatures. In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 257–265. Springer, Heidelberg, April 1991. doi:10.1007/3-540-46416-6_22.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. doi:10.1109/TIT.1976.1055638.

[GKMZ20] Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. *CoRR*, abs/2008.02527, 2020. URL: https://arxiv.org/abs/2008.02527, arXiv:2008.02527.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, may 1996. doi:10.1145/233551.233553.

[Goh03] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. https://eprint.iacr.org/2003/216.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987. doi:10.1145/28395.28416.

[GPSW06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 89–98. ACM Press, October / November 2006. Available as Cryptology ePrint Archive Report 2006/309. doi:10.1145/1180405.1180418.

[GPT23] Zichen Gui, Kenneth G. Paterson, and Tianxin Tang. Security analysis of MongoDB queryable encryption. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7445–7462, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/gui.

[HL17] Qiong Huang and Hongbo Li. An efficient public-key searchable encryption scheme secure against inside keyword guessing attacks. *Information Sciences*, 403-404:1–14, 2017. URL: https://www.sciencedirect.com/science/article/pii/S0020025516321090, doi:https://doi.org/10.1016/j.ins.2017.03.038.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

[KMO18] Seny Kamara, Tarik Moataz, and Olga Ohrimenko. Structured encryption and leakage suppression. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 339–370. Springer, Heidelberg, August 2018. doi:10.1007/978-3-319-96884-1_12.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. doi:10.1145/359545.359563.

[MCDP24] Priyanka Mondal, Javad Ghareh Chamani, Ioannis Demertzis, and Dimitrios Papadopoulos. I/o-efficient dynamic searchable encryption meets forward & backward privacy. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/mondal.

[MCT$^+$24] Long Meng, Liqun Chen, Yangguang Tian, Mark Manulis, and Suhui Liu. Fease: Fast and expressive asymmetric searchable encryption. Cryptology ePrint Archive, Paper 2024/054, 2024. https://eprint.iacr.org/2024/054. URL: https://eprint.iacr.org/2024/054.

[Mon22] Inc MongoDB. Queriable encryption. In *Manual*, 2022. URL: https://www.mongodb.com/docs/manual/core/queryable-encryption/.

[MR22] Brice Minaud and Michael Reichle. Dynamic local searchable symmetric encryption. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 91–120. Springer, Heidelberg, August 2022. doi:10.1007/978-3-031-15985-5_4.

[MR23] Brice Minaud and Michael Reichle. Hermes: I/O-efficient forward-secure searchable symmetric encryption. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VI*, volume 14443 of *LNCS*, pages 263–294. Springer, Heidelberg, December 2023. doi:10.1007/978-981-99-8736-8_9.

[PM21] Sikhar Patranabis and Debdeep Mukhopadhyay. Forward and backward private conjunctive searchable symmetric encryption. In *NDSS 2021*. The Internet Society, February 2021.

[PPSY21] Sarvar Patel, Giuseppe Persiano, Joon Young Seo, and Kevin Yeo. Efficient boolean search over encrypted data with reduced leakage. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 577–607. Springer, Heidelberg, December 2021. doi:10.1007/978-3-030-92078-4_20.

[RW22] Doreen Riepel and Hoeteck Wee. FABEO: Fast attribute-based encryption with optimal security. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2491–2504. ACM Press, November 2022. doi:10.1145/3548606.3560699.

[VRMO18] Cédric Van Rompay, Refik Molva, and Melek Önen. Secure and scalable multi-user searchable encryption. In *Proceedings of the 6th International Workshop on Security in Cloud Computing*, SCC '18, page 15–25, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3201595.3201597.

[WC24] Jiafan Wang and Sherman S. M. Chow. Unus pro omnibus: Multi-client searchable encryption via access control. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024. URL: https://www.ndss-symposium.org/ndss-paper/unus-pro-omnibus-multi-client-searchable-encryption-via-access-control/, doi:10.14722/ndss.2024.23288.