SENSSE: Simple, Efficient Searchable Symmetric Encryption for Sensor Networks

Bojan Spasić[1,2][a], Olivier Markowitch[1][b] and Philippe Thiran[2]
[1]*Université Libre de Bruxelles, Belgium*
[2]*Université de Namur, Belgium*

Keywords: Dynamic Searchable Symmetric Encryption, DSSE, Forward Privacy, Client Storage.

Abstract: In this work, we focus on the problem of forward-private dynamic searchable symmetric encryption (DSSE) in the multi-client setting. In order to achieve forward privacy, efficient DSSE schemes require clients to store local information, such as per-keyword search counters. Such construction choices prevent these schemes from being used in a multi-user scenario. We revisit the concept of forward privacy with a goal to examine the need for client storage. As a result, we propose a new method of realising forward privacy without requiring the clients to keep any state information. Based on this method, we construct a dynamic, forward-private searchable symmetric encryption scheme supporting multiple concurrent clients with minimal overhead. The proposed construction requires no state to be kept by clients, yet provides optimal asymptotic behaviour both in time, storage and communication cost, while having similar leakage profile to other state-of-the-art DSSE schemes.

1 INTRODUCTION

1.1 Motivation

Searchable encryption is a family of cryptographic techniques that enable outsourcing of data to another party (e.g. a cloud provider) for storage, with the ability to perform searching - ideally without disclosing anything about search queries, search results or the data itself. Practical searchable encryption schemes, however, always leak certain amount of information (Kamara et al., 2018). Among different techniques, searchable symmetric encryption has so far provided the best tradeoffs between performance, query expressiveness and security (Kamara and Moataz, 2017). Dynamic searchable symmetric encryption (DSSE) schemes allow the clients to interact with the server in order to add or remove records from the database. To avoid devastating attacks, capable of defeating confidentiality (Zhang et al., 2016; Cash et al., 2015), such schemes must ensure that they limit the amount of data leaked during adding or removing records. Schemes providing such measures are known as forward- and backward-private, respectively. Currently, all known forward-private DSSE schemes having optimal performance require that clients keep some kind of local state (e.g. a map of all the search keywords and an associated number of queries) for encryption scheme to achieve forward privacy (Kim et al., 2019). The existence of client storage presents a problem if the DSSE scheme is to be used in a multi-client scenario, due to the need to securely synchronise the state between clients.

This limitation has traditionally confined the forward-private DSSE schemes to use cases with a single reader / single writer client, precluding their use in dynamic and distributed environments, such as sensor networks, or any other architecture involving extensive collection of data from a set of writing nodes and using public cloud as data storage and processing backbone - a common Internet-of-Things pattern (Lin et al., 2017). Without this shortcoming, DSSE would be a good candidate for implementing on sensor nodes typically utilising simple, low-power hardware designs, due to its reliance on efficient symmetric cryptographic primitives.

1.2 Previous Work

Searchable Symmetric Encryption. (SSE) uses symmetric-key cryptography (mainly pseudorandom

[a] https://orcid.org/0000-0002-3118-176X
[b] https://orcid.org/0000-0001-5467-5060

Spasić, B., Markowitch, O. and Thiran, P.
SENSSE: Simple, Efficient Searchable Symmetric Encryption for Sensor Networks.
DOI: 10.5220/0009824403630371
In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications (ICETE 2020) - SECRYPT*, pages 363-371
ISBN: 978-989-758-446-6

functions and pseudorandom permutations) to secure a data set or a structure so that it can be privately and efficiently queried. SSE was introduced in the seminal work by Song, Wagner and Perrig (Song et al., 2000), who proposed the first provably secure and practical encryption scheme allowing sequential search over cyphertexts.

The first SSE scheme with search time that is proportional to the number of documents that match the searched keyword (and shown to be optimal) was developed by Curtmola, Garay, Kamara and Ostrovsky. They proposed an inverted index with entries corresponding to each distinct word in the whole database, as opposed to the previously used per-document indices (Curtmola et al., 2006). Their construction was the first one to use encrypted linked lists of document identifiers that correspond to a search keyword, pointed to by a trapdoor computed using only a pseudorandom permutation and a pseudorandom function applied to the keyword.

A generalisation of the previous work on SSE to the setting of arbitrarily-structured data was made by (Chase and Kamara, 2010). An important contribution of their work is the introduction of the formal concept of the stateful leakage function. Updated security definitions included this important concept, allowing the security guarantees of an SSE scheme to be precisely defined up to a given leakage profile.

Kamara, Papamanthou and Roeder introduced the first efficient (time sublinear in the number of stored documents) **dynamic searchable symmetric encryption (DSSE) scheme** (Kamara et al., 2012). A DSSE scheme supports the modification of the encrypted storage structures by allowing updates. However, the introduction of this additional functionality opened up the attack surface. File-injection attacks described by Zhang, Katz and Papamanthou demonstrated the vulnerability of early DSSE schemes to maliciously crafted documents that client would insert to the encrypted database. Such attacks were shown to be devastating for query privacy, because the database server could learn client's keywords after injecting a relatively small number of documents (Zhang et al., 2016).

The problem was addressed through the introduction of the concepts of *forward privacy* and *backward privacy*, first informally proposed by Stefanov, Papamanthou and Shi (Stefanov et al., 2014). Forward privacy is a strong requirement on DSSE schemes which, informally, states that the server must not be able to infer whether or not a freshly added record contains any of the keywords from the previous searches (Bost, 2016). Conversely, the other important notion, *backward privacy*, formalised by Bost, Minaud and Ohrimenko (Bost et al., 2017), is a requirement that the server does not learn about deleted records from searches made after deletion. Therefore, forward privacy is a mandatory property in a secure DSSE supporting updates, while a secure DSSE supporting deletes must be backward-private.

While efficient **multiple-client** SSE constructions have been known since (Curtmola et al., 2006), their more complex variants with multiple writers and multiple readers have traditionally required use of public-key encryption with keyword search (PEKS) which does not offer optimal search and update asymptotics (Bösch et al., 2014).

However, in the 'pure' DSSE setting, the presence of client storage, usually required for constructing the mechanism that ensures forward privacy, remains a burden for extension to multiple-client scenario. A recent work by (Bakas and Michalas, 2019) presents a scheme, albeit only in multi-reader mode, using Intel's SGX. In a concurrent work (Frimpong et al., 2020), a multiple-client forward-private DSSE scheme dedicated to sensor networks is presented that attempts to circumvent the problem by introducing intermediary processing "fog" nodes (that keep their own state) and the execution of the search operation is distributed between the cloud provider and the fog nodes. However, the authors do not state asymptotic performance of their solution.

2 OUR CONTRIBUTION

We revisit the problem of forward-private, dynamic searchable symmetric encryption in multiple-writer setting. As a result, we present SENSSE - a simple and efficient DSSE scheme satisfying the following properties:

(1) forward-privacy

(2) optimal asymptotic behaviour of database updates and searches

(3) supporting multiple writer and reader clients

(4) requiring no client storage or state

To achieve forward privacy, many previous works on DSSE schemes rely on client-side storage, such as a local copy of the keyword dictionary holding the number of updates containing the given keywords, e.g. (Etemad et al., 2018). This not only creates a storage burden on the client, but effectively prohibits scenarios with multiple clients, due to the implicit need for synchronisation of this state between clients and the associated communication complexity (Kim et al., 2019). Instead, we find inspiration

in the recently revisited line of research focussing on *rebuildable* DSSE schemes (Amjad et al., 2019), (Demertzis et al., 2019). Informally, a rebuildable DSSE scheme describes an additional algorithm, or client/server protocol, during which operations on the server (and possibly client) state are performed in order to improve or restore performance, reclaim space, or re-establish correctness after a series of query, update or delete operations. Our construction achieves zero client storage by allowing for a small temporary violation of correctness; namely, the visibility of most recently added records to search. Correctness is restored periodically or on demand by a (typically reader) client, who initiates the REBUILD protocol. The possibility to have concurrent client access enables parallelisation of REBUILD by distributing its execution to multiple clients. The above properties make our scheme especially well-suited for implementation in IoT scenarios, where multiple low-power sensor nodes produce copious amounts of data that is stored in a database managed by a cloud service provider and used by one or more application clients. In such scenarios, the efficiency of updates is paramount, while the applications that read and process data may tolerate the small penalty of running the REBUILD protocol. The lack of the requirement to keep state at the client also enables implementation in thin-client scenarios, such as web-based application clients for email, messaging and the like.

3 PRELIMINARIES

3.1 Notation

$\{0,1\}^n$ represents the set of all binary strings of length n. $\{0,1\}^*$ denotes the set of all finite binary strings. We denote deterministic assignment of y to x by $x \leftarrow y$, and pseudorandom sampling of x from a set X by $x \xleftarrow{\$} X$. We use the abbreviation PPT to denote probabilistic polynomial-time when referring to algorithms or adversaries.

In our construction, we use containers such as maps, multimaps and sets. A map is a collection of label/value pairs l_i, v_i. We denote values with a lowercase letter. We instantiate an empty map by writing $map \leftarrow \varnothing$. To read a value associated to a label l from a map, we write $val \leftarrow map[l]$. To write a value associated to a label l in a map, we write $map[l] \leftarrow val$. We refer to the number of label/value pairs as the map size and write $|map|$ to denote it. A multimap is a collection of label/set pairs l_i, S_i. We denote sets by an uppercase letter. We instantiate an empty multimap by writing $multimap \leftarrow \varnothing$. To read a set associated to a label l from a multimap, we write $S \leftarrow map[l]$. To write a set associated to a label l in a multimap, we write $multimap[l] \leftarrow S$. We refer to the number of label/set pairs as the multimap size and write $|multimap|$ to denote it. A set is an unordered collection of values. We instantiate an empty set by writing $set \leftarrow \varnothing$. To add a value to a set, we write $set \leftarrow set \cup \{val\}$. To remove a value from a set, we write $set \leftarrow set \setminus \{val\}$.

3.2 Cryptographic Primitives

We make use of the following basic cryptographic primitives:

A **Symmetric Key Encryption** SKE=(KEYGEN, ENC, DEC) is a semantically-secure encryption scheme consisting of the three PPT algorithms: GEN takes a security parameter as input and generates a secret key k. ENC takes the secret key k and a message m and outputs a cyphertext c. DEC takes as input the secret key k and a cyphertext c and returns the corresponding cleartext message m (Katz and Lindell, 2014).

For the **Hash Function** family $H : \mathcal{K} \times \mathcal{D} \rightarrow \mathcal{R}$ and for a polynomial-time adversary A, the hash function family H is said to be collision-resistant if

$$\mathsf{Adv}^{\mathrm{col}}_{H,A}(\lambda) := \Pr\left[K \xleftarrow{\$} \mathcal{K}, (M, M') \leftarrow A(K) : M \neq M' \wedge H_K(M) = H_K(M')\right] = \mathsf{negl}(\lambda)$$

(Katz and Lindell, 2014).

A **Pseudorandom Function** PRF is a function indistinguishable from a truly random function by a PPT adversary. Let $F : \mathcal{K} \times \mathcal{D} \rightarrow \mathcal{R}$ be a PPT computable function family where $\mathcal{K}$ is a finite *key space* and $\mathcal{R}$ is the *range* of F. For a key $K \in \mathcal{K}$, we write $F_K : \mathcal{D} \rightarrow \mathcal{R}$. Let $\mathsf{Adv}^{\mathrm{prf}}_{F,A}(\lambda)$ be the pseudorandom function distinguishing advantage of a PPT adversary A against F, defined as

$$\left|\Pr\left[K \xleftarrow{\$} \mathcal{K} : A^{F_K(\cdot)}(1^\lambda) = 1\right] - \Pr\left[(\pi \xleftarrow{\$} \Pi : \mathcal{D} \rightarrow \mathcal{R}) : A^{\pi(\cdot)}(1^\lambda) = 1\right]\right|$$

The function F is referred to as a PRF if $\mathsf{Adv}^{\mathrm{prf}}_{F,A}(\lambda) = \mathsf{negl}(\lambda)$ for any PPT adversary A (Katz and Lindell, 2014). When SKE is set to operate on a set, it is implied that the operation is performed on each element of the set.

4 CONSTRUCTION

4.1 Data Structures

Server holds a map **DB** of encrypted records such that each record is individually addressable by a unique id. To enable keyword search, as is common with searchable symmetric schemes, our construction deploys an inverted index **INVIDX**, mapping keywords to record ids. Each entry in the inverted index thus consists of a pair $(\tau, EIDS)$ where τ is a search token obtained from the keyword via a PRF, while *EIDS* is a set of indices into **DB**, encrypted using a CPA-secure symmetric encryption scheme. In addition to the inverted index, our scheme also keeps a forward index **FWDIDX**. Each entry in the forward index consists of a pair (id, ETS) where *id* is a record id and ETS is an encrypted search token set, corresponding to the keywords found in the record with the given id. This index is directly populated upon adding new records. On the client side there is no requirement for any state storage.

4.2 Informal Description

Our approach to updates differs from those found in the majority of DSSE schemes in the literature. Instead of directly updating the inverted index when adding a new record, the client asks the server to store the *encrypted* search tokens (corresponding to the distinct keywords present in the record) in a map indexed by the record id. This automatically satisfies the requirement for the forward privacy, as no token is revealed to the server during the addition of the new record. A drawback of this approach is that searching, which operates classically on the inverted index, will not return any results matching newly added records until a *rebuild* is performed. Rebuild, intuitively, is an operation of transposing the map of encrypted tokens that is populated during record addition into a format such that it can be readily merged with the inverted index on the server. We reason about the leakage and its possible ramifications and potential for exploiting it by a persistent adversary in the section on security.

4.3 Definition

In this section, we formally define SENSSE, our dynamic, forward-private searchable symmetric encryption scheme. Our definition is loosely based on the rebuildable DSSE formalism of (Amjad et al., 2019).

Definition 1. A dynamic, rebuildable structured encryption scheme Σ = (SETUP, ADD, REBUILD, SEARCH), consists of the following polynomial-time protocols between the client and the server:

- $(S, K_G, K_{SKE}) \leftarrow$ SETUP(λ) is a probabilistic algorithm that takes as input a security parameter λ and outputs the initialised server state S and two pseudorandom secret keys K_G and K_{SKE}.
- $(S') \leftarrow$ ADD$((W,R), K_G, K_{SKE}, S)$ is a probabilistic protocol between the client and the server that takes as input a data record and its corresponding set of distinct keywords (W,R), the server state S, the secret keys K_G and K_{SKE} and outputs the updated server state S'.
- $(S') \leftarrow$ REBUILD$(\kappa, K_G, K_{SKE}, S)$ is a probabilistic protocol between the client and the server that takes as input a security parameter κ, the secret keys K_G and K_{SKE}, and the server state S and outputs the updated server state S'.
- $(ids) \leftarrow$ SEARCH(w) is a probabilistic protocol between the client and the server that takes as input a keyword w, the secret keys K_G and K_{SKE}, and the server state S and outputs the set of record ids of records matching the keyword w.

4.4 Detailed Description

Our construction uses the following cryptographic primitives: a PRF G producing μ bits of range, and a private-key encryption scheme SKE. The algorithms, formally defined above, are described further as follows.

SETUP. On the client side, the Setup algorithm takes as input a security parameter λ and outputs a PRF key and a SKE key to be used by the client. On the server side, Setup instantiates an empty map **DB** which will contain the data records labelled by ids (generated from a monotonically increasing counter of records), and an empty multimap (inverted index) **INVIDX**, to hold encrypted record ids, labelled by search tokens (generated by the PRF). The global counter of all inserted records in **DB** is initialised to 0. In addition, a set to hold encrypted tokens corresponding to the added records is instantiated. We assume that all clients (readers and writers) participating to the scheme are in possession of the keys K_G and K_{SKE}. The mechanism of distributing the keys among clients is out of scope of this work. **ADD.** Add is a two-party protocol between the client and the server. Client initiates by taking a record R and a set of distinct keywords $W \in R$ as inputs. The purpose of Add is to efficiently and securely store the pair (W,R) in the **DB**. To do so, the client prepares a unique keyword token τ_w for each $w \in W$ by computing the PRF G, instantiated with K_G, and encrypts the

Algorithm 1: SETUP.

CLIENT.SETUP(λ)
  $K_G \xleftarrow{\$} \{0,1\}^\lambda$
  $K_{SKE} \xleftarrow{\$} SKE.KeyGen(\lambda)$
  return (K_G, K_{SKE})
end
SERVER.SETUP
  $State.invIDX \leftarrow \varnothing$
  $State.DB \leftarrow \varnothing$
  $State.fwdIDX \leftarrow \varnothing$
  $State.numRecords \leftarrow 0$
end

record itself using the SKE with the key K_{SKE}. The client then sends the pair $(Enc(\{\tau_w\}_{w\in W}), Enc(R))$ to the server. The server first computes the new record id by taking the current value of the record counter, then adds the encrypted record R to the map **DB** under the label id. Note that, unlike with most DSSE schemes, server does not update the inverted index multimap **INVIDX** at this point. Instead, the server stores the pairs $(id, Enc(\{\tau_w\}_{w\in W}))$ in a separate set, which will be accessed by the Rebuild algorithm later on. As a consequence, records immediately added are not visible to the Search algorithm before Rebuild is executed. By deferring the update of the inverted index multimap the scheme gets forward privacy, as the leakage of Add does not reveal any of the keyword tokens, and thus server cannot realise whether or not the freshly added record contains any of the tokens previously searched for. Server concludes by increasing the record counter by one, thus preparing for the next execution of the Add algorithm. As Add protocol requires no client storage, it can be run concurrently by multiple instances of the client. **REBUILD.** Rebuild is a two-party protocol between the client and the server. The purpose of Rebuild is to restore search correctness after several executions of the Add protocol. The client initiates the protocol by requesting a random sample of κ pairs $(id_i, Enc(\{\tau_w\}_{w\in W}))$ from the server's index populated by Add protocol. The client proceeds by transposing the sample (finding all ids corresponding to a given τ_w), encrypts the ids using SKE and inserts the obtained sets $\{Enc(id_i)\}$ under their corresponding τ_w label in the multimap **INVIDX**. As we will show in the section about security, κ is a security parameter, directly related to the significance of the leakage information that the server may learn while executing the protocol Rebuild. As Rebuild protocol requires no client storage, it can be run concurrently by multiple instances of the client. **SEARCH.** Search is a two-party protocol between the client and the server. Client takes the search keyword

Algorithm 2: ADD.

CLIENT.ADD($(W,R), K_{SKE}, K_G$)
  $ER \leftarrow SKE.Encrypt(K_{SKE}, R)$
  $T \leftarrow \varnothing$
  for all $w \in W$ **do**
    $\tau_w \leftarrow G(K_G, w)$
    $T \leftarrow T \cup \{\tau_w\}$
  end for
  $ET \leftarrow SKE.Encrypt(K_{SKE}, T)$
  $Server.Add(ET, ER)$
end
SERVER.ADD(ET, ER)
  $id \leftarrow State.numRecords$
  $State.DB[id] \leftarrow ER$
  $State.fwdIDX \leftarrow State.fwdIDX \cup (id, ET)$
  $State.numRecords \leftarrow State.numRecords + 1$
end

Algorithm 3: REBUILD.

CLIENT.REBUILD(κ, K_{SKE}, K_G)
  $TS \leftarrow Server.GetShuffledIndex(\kappa)$
  if $TS.count < \kappa$ **then**
    end
  end if
  $iidx \leftarrow \varnothing$
  for $i \leftarrow 0, \kappa - 1$ **do**
    $(id, ET) \leftarrow TS[i]$
    $eid \leftarrow SKE.Encrypt(K_{SKE}, id)$
    $T \leftarrow SKE.Decrypt(K_{SKE}, ET)$
    for all $\tau_w \in T$ **do**
      $iidx[\tau_w] \leftarrow iidx[\tau_w] \cup eid$
    end for
  end for
  $Server.Update(iidx)$
end
SERVER.GETSHUFFLEDINDEX(κ)
  $TS \xleftarrow{\$\kappa} State.fwdIDX$
  $State.fwdIDX \leftarrow State.fwdIDX \setminus TS$
  return TS
end
SERVER.UPDATE($iidx$)
  $State.invIDX.Merge(iidx)$
end

w and computes the corresponding search token τ_w by instantiating a PRF G with key K_G and evaluating it over w. Client then sends the τ_w to the server, which finds the matching set of encrypted record *ids* in the multimap **INVIDX**. The client then proceeds to decrypt the *ids* with SKE, using the key K_{SKE}. Upon completion, the client has the set of record *ids* matching the keyword w, and can proceed to request indi-

vidual records from the server.

Algorithm 4: SEARCH.

```
CLIENT.SEARCH(w, K_SKE, K_G)
    τ_w ← G(K_G, w)
    EIDS ← Server.Search(τ_w)
    IDS ← ∅
    for all eid ∈ EIDS do
        id ← SKE.Decrypt(K_SKE, eid)
        if id = 0 then exit for
        end if
        IDS ← IDS ∪ id
    end for
    return IDS
end
SERVER.SEARCH(τ_w)
    EIDS ← State.invIDX[τ_w]
    return EIDS
end
```

4.5 Security

In our threat model, we consider the server to be the single, persistent, honest-but-curious PPT adversary, that has access to the server state (section 4.1) and faithfully executes the server-side portions of the protocols; having access to above, the server tries to infer as much information as possible about the secret records it stores, and/or the searched keywords. As is common in the literature, we define security in terms of the information leakage present in each of the algorithms of the scheme. Thus, we define the notion of security as follows: we expect our scheme to reveal no information beyond the leakage profile $\{\mathcal{L}_{Setup}, \mathcal{L}_{Add}, \mathcal{L}_{Search}, \mathcal{L}_{Rebuild}\}$. We proceed by giving a precise description of this leakage profile.

Our scheme is designed to be fully dynamic, i.e. all the record data comes exclusively from the client executing the Add protocol. Thus, there are no operations concerning record data, nor are there any tokens or identifiers being generated or exchanged in the Setup protocol. Hence, we conclude that Setup adds no meaningful leakage information outside of the information that is already available to the server:

$$\mathcal{L}_{Setup} = \varnothing$$

During the execution of the algorithm Add, server receives pairs $(Enc(\{\tau_w\}_{w\in W}), Enc(R))$. During execution on the server side, server generates the unique identifier *id* associated with each given such pair. Thus, for each pair $(Enc(\{\tau_w\}_{w\in W}), Enc(R))$, the server learns the associated *id*, the size (or upper bound on the size) of the encrypted record R and the number (or the upper bound of the number) of the distinct keywords belonging to the record R. The server expressly does not learn any of the keywords and thus cannot infer a link between a newly added record and the search history, satisfying the forward privacy requirement introduced by (Stefanov et al., 2014). The leakage of the protocol Add is:

$$\mathcal{L}_{Add}(Enc(\{\tau_w\}_{w\in R}), Enc(R)) = (id_r, |R|, |\{\tau_w\}_{w\in R}|)$$

The query algorithm sends a search token τ_w corresponding to the keyword w to the server. Server performs a lookup into the multimap **INVIDX** and returns encrypted record *ids*. At this moment, the server learns only the (upper bound of) the size of the resulting record set RS containing w. However, as the client subsequently decrypts the *ids* and requests the encrypted records from the server, server will have also learned the exact number of matching records, the ids themselves, and the size of each record.

$$\mathcal{L}_{Search}(\tau_w) = \{(id, |R|) \in RS\}$$

This leakage is consistent with the standard leakage profile of the DSSE schemes in the literature (Etemad et al., 2018), (Bost, 2016), (Bösch et al., 2014).

As we have so far seen mostly standard forward-private DSSE leakage behaviour from Setup, Add and Search algorithms, the leakage behaviour of the Rebuild protocol is of most interest for our scheme. Rebuild takes a set of random samples $\mathfrak{R}$ of κ pairs $(id, Enc(\{\tau_w\}_{w\in R_{id}}))$, and then iterates through the set to assign matching set of *ids* to every distinct τ_w found in the set. But the server sees only the final result of this assignment; i.e. for a given set $\{id\}$, the server sees the set $\{\tau_w\}$. The mappings are not revealed, as the *ids* in the resulting multimap are encrypted. Therefore, the leakage of the Rebuild protocol is:

$$\mathcal{L}_{Rebuild}() = \{\tau : \tau \in \mathfrak{R}\} \cup \{id : id \in \mathfrak{R}\}$$

The question crucial to security of the Rebuild protocol is what can the server infer from this information. A persistent adversary observing the protocol execution on the server can only resort to guessing the actual permutation of *ids* among the τ_w slots in order to infer the link between keyword tokens and their corresponding records. There are $(2^\kappa - 1)$ different ways to choose one or more *ids* for each of the τ_w slots. The number of τ_w slots in $\mathfrak{R}$ may vary from 1 to $|D|$, where D is the dictionary of all possible keywords. So, the number of permutations in the best case is $(2^\kappa - 1)$ and in the worst case $(2^\kappa - 1) \cdot |D|$. The probability of guessing the correct permutation of κ ids over all τ_w slots given the leakage $\mathcal{L}_{Rebuild}$ is thus $O(2^{-\kappa})$. The server can observe the inverted index INVIDX for changes of the set of associated encrypted indices, but can only ascertain that one or more additions took place, without inferring which IDs got associated to which token, as the IDs are encrypted. Even, if the

server maintains a temporal record of changes to correlate own record injections with changes to its state structures, it cannot infer the association of a new record ID to an already searched token due to the fact that the inverted index is only updated in Rebuild and those updates are probabilistic due to the random sampling of the forward index performed in Rebuild.

We proceed by defining the adaptive security following the established game formalism (Kamara et al., 2018):

Definition 2. Let Σ = (SETUP, ADD, REBUILD, SEARCH), be a dynamic, rebuildable structured encryption scheme. Consider further the probabilistic experiments below, where $\mathcal{A}$ is a stateful PPT adversary, $\mathcal{S}$ is a stateful simulator, and $\mathcal{L}_{Setup}$, $\mathcal{L}_{Add}$, $\mathcal{L}_{Search}$, $\mathcal{L}_{Rebuild}$ are leakage profiles:

Real$_{\Sigma,\mathcal{A}}$ the adversary $\mathcal{A}$ receives the server state from the challenger. The adversary then adaptively chooses polynomially-many operations from Add, Rebuild, Search and receives their transcripts. Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.

Ideal$_{\Sigma,\mathcal{A},\mathcal{S}}$ given the $\mathcal{L}_{Setup}$, the simulator $\mathcal{S}$ returns the server state to the adversary $\mathcal{A}$. The adversary then adaptively chooses polynomially-many operations from Add, Rebuild, Search, for which the simulator receives $\mathcal{L}_{Add}$, $\mathcal{L}_{Rebuild}$, $\mathcal{L}_{Search}$, and returns their transcripts. Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.

We say that Σ is adaptively $(\mathcal{L}_{Setup}, \mathcal{L}_{Add}, \mathcal{L}_{Search}, \mathcal{L}_{Rebuild}) - secure$, if there exists a PPT simulator $\mathcal{S}$ such that for all PPT adversaries $\mathcal{A}$, the following expression is negligible in λ:

$\Pr\left[[\mathbf{Real}_{\Sigma,\mathcal{A}}(\lambda)=1]-\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(\lambda)=1]\right]$.

Theorem 3.1. *If* SKE *is a CPA-secure encryption scheme and G is a pseudorandom function, then* SENSSE is $(\mathcal{L}_{Setup}, \mathcal{L}_{Add}, \mathcal{L}_{Search}, \mathcal{L}_{Rebuild}) - secure$.

Proof. Consider the following simulator *S*, simulating Setup, Add, Rebuild and Search, whose input is the respective leakage profile. Server is the adversary $\mathcal{A}$. $\mathcal{H}_1$, $\mathcal{H}_2$, $\mathcal{H}_3$, $\mathcal{H}_4$ and $\mathcal{H}_5$ are random oracles, and we denote programming a random oracle by $\mathcal{H}(x) \leftarrow y$.

Given the simulator *S*, the goal of the proof is to show that the games REAL and IDEAL are indistinguishable. We proceed by defining a series of hybrids, with the aim to show that the adversary cannot distinguish the output of REAL and IDEAL games unless an underlying assumption has been violated.

Game 0: Game 0 is the same as Real.

Game 1: Game 1 is the same as Game 0, except that instead of using the SKE to encrypt the records in the ADD protocol, we generate a random string of the same length. Since SKE is a semantically-secure PRP, its output is indistinguishable from the random string of the same length and thus games 1 and 0 are indistinguishable.

S.SETUP($\mathcal{L}_{Setup} = \varnothing$).

```
CLIENT.SETUP(λ)
    K_G ←$ {0,1}^λ
    K_SKE ←$ {0,1}^λ
    return (K_G, K_SKE)
end
SERVER.SETUP
    State.invIDX ← ∅
    State.DB ← ∅
    State.fwdIDX ← ∅
    State.numRecords ← 0
end
```

S.ADD($\mathcal{L}_{Add} = (id_r, |R|, |\{\tau_w\}_{w\in R}|)$).

```
CLIENT.ADD(id_r, |R|, |{τ_w}_{w∈R}|)
    ER ←$ {0,1}^log|R|
    ET ← ∅
    for i ← 1, |{τ_w}_{w∈R}| do
        eτ_w ← H_1(id_r)
        ET ← ET ∪ {eτ_w}
    end for
    Server.Add(ET, ER)
end
SERVER.ADD(ET, ER)
    id ← State.numRecords
    State.DB[id] ← ER
    State.fwdIDX ← State.fwdIDX ∪ (id), ET)
    State.numRecords ← State.numRecords + 1
end
```

Game 2: Game 2 is the same as Game 1, except we get the search token τ_w from the random oracle $\mathcal{H}_1$ instead of using the PRF G in Add. With that, we remove the SKE.Encrypt as it is no longer needed. We also replace the PRF G in Search with $\mathcal{H}_1(id)$. Game 2 is indistinguishable from the Game 1 because G is a PRF, and as such, its output is indistinguishable from the output of a random oracle.

Game 3: Game 3 is the same as Game 2, but the SKE has been removed from the Rebuild algorithm. Instead, the random oracle $\mathcal{H}_2(id)$ is used to compute eid and $\mathcal{H}_3(id)$ is programmed to hold a mapping between id and eid, which is subsequently used in Search, replacing SKE.Decrypt. $\mathcal{H}_4(id)$ used to get τ_w instead of using decryp-

S.REBUILD($\mathcal{L}_{Rebuild} = TS = \{\tau : \tau \in \mathfrak{R}\} \cup IDS = \{id : id \in \mathfrak{R}\}$).

CLIENT.REBUILD(κ)
$TS \leftarrow Server.GetShuffledIndex(\kappa)$
if $TS.count < \kappa$ **then**
end
end if
$iidx \leftarrow \varnothing$
for $i \leftarrow 0, \kappa - 1$ **do**
$(id, ET) \leftarrow TS[i]$
$eid \leftarrow \mathcal{H}_2(id)$
$\mathcal{H}_3(eid) \leftarrow id$
for all $e\tau_w \in ET$ **do**
$\tau_w \leftarrow \mathcal{H}_4(e\tau_w)$
$iidx[\tau_w] \leftarrow iidx[\tau_w] \cup eid$
$\mathcal{H}_5(id) \leftarrow \tau_w$
end for
end for
$Server.Update(iidx)$
end

SERVER.GETSHUFFLEDINDEX(κ)
$TS \xleftarrow{\$\kappa} State.fwdIDX$
$State.fwdIDX \leftarrow State.fwdIDX \setminus TS$
return TS
end

SERVER.UPDATE($iidx$)
$State.invIDX.Merge(iidx)$
end

S.SEARCH($\mathcal{L}_{Search} = \{(id, |R|) \in RS\}$).

CLIENT.SEARCH($(id, |R|) \in RS$)
$\tau_w \leftarrow \mathcal{H}_5(id)$
$EIDS \leftarrow Server.Search(\tau_w)$
$IDS \leftarrow \varnothing$
for all $eid \in EIDS$ **do**
$id \leftarrow \mathcal{H}_3(eid)$
if id = 0 **then exit for**
end if
$IDS \leftarrow IDS \cup \{id\}$
end for
return IDS
end

SERVER.SEARCH(τ_w)
$EIDS \leftarrow State.invIDX[\tau_w]$
return $EIDS$
end

tion. Indistinguishability of the Game 4 from the Game 3 follows from the fact that SKE is a semantically secure PRP whose output, following the PRP/PRF switching lemma, is indistinguishable from a PRF. An output of a PRF is, in turn, indistinguishable from a random oracle.

Game 4: Game 4 is the same as Game 3, but now we replace the input parameters of all protocols in REAL with the leakage profile. $\mathcal{H}_5(id)$ is programmed in Rebuild to store τ_w. We change $\mathcal{H}_1(w)$ to $\mathcal{H}_5(id)$ in Add to use it to get distinct values in place of encrypted token, which is later used in Search to retrieve valid records. Game 4 is now the same as the as the Ideal experiment, and the adversary $\mathcal{A}$ is thus unable to distinguish between the games Real and Ideal. □

4.6 Efficiency

The asymptotic cost of adding a record is $O(|W|)$ (where W is the set of keywords in the inserted record R) time to prepare the (W, R) pair before sending it to server. The server performs the actual update of its data structures in $O(1)$. Overall, the asymptotic cost of Add is $O(|W|)$ which is optimal (Curtmola et al., 2006).

In Search, the asymptotic cost on the client side is $O(|R|)$, where R is the result set. The server cost is $O(1)$. Overall, the cost of the search operation is $O(|R|)$ which is optimal (Curtmola et al., 2006).

The cost of Rebuild is at least $\Omega(\kappa)$ (each item in the forward index sample contains only one keyword token) and at most $O(\kappa \cdot |D|)$ (each item in the forward index sample contains all existing keyword tokens) where D is the dictionary of all keywords and κ is a security parameter. The pseudorandom sampling introduces only a constant cost of generating κ pseudorandom numbers from the interval $[0..|fwdIDX|)$. In a multi-writer setting, Rebuild protocol would typically be run by the reader clients, either in the background with some frequency, or on demand, when a search operation is required to return the most recently added records.

Storage Efficiency. SENSSE requires client storage of $O(1)$ and server storage of $O(N)$, for the index, where N is the total number of (keyword, record) mappings (the size occupied by the multimap **INVIDX**. There is an additional (intermittent) storage cost for the random sample from the server's index of $O(|D|)$, where D is the dictionary containing all keywords.

Communication Efficiency. SENSSE has optimal communication cost of $O(1)$ calls from client to server in all three interactive protocols ADD, REBUILD and SEARCH. SETUP incurs no client/server communication cost.

5 CONCLUSION

We presented a simple, efficient, forward-private DSSE, requiring no client storage, making it an interesting candidate for sensor networks. In future work, we aim to construct an experiment to evaluate the scheme performance in a practical setting.

REFERENCES

Amjad, G., Kamara, S., and Moataz, T. (2019). Breach-resistant structured encryption. *Proceedings on Privacy Enhancing Technologies*, 2019(1):245–265.

Bakas, A. and Michalas, A. (2019). Multi-client symmetric searchable encryption with forward privacy. Cryptology ePrint Archive, Report 2019/813. https://eprint.iacr.org/2019/813.

Bost, R. (2016). $\Sigma o \varphi o \varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1143–1154, New York, NY, USA. Association for Computing Machinery.

Bost, R., Minaud, B., and Ohrimenko, O. (2017). Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, pages 1465–1482, New York, NY, USA. Association for Computing Machinery.

Bösch, C., Hartel, P., Jonker, W., and Peter, A. (2014). A survey of provably secure searchable encryption. *ACM Comput. Surv.*, 47(2).

Cash, D., Grubbs, P., Perry, J., and Ristenpart, T. (2015). Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 668–679, New York, NY, USA. Association for Computing Machinery.

Chase, M. and Kamara, S. (2010). Structured encryption and controlled disclosure. In Abe, M., editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 577–594, Berlin, Heidelberg. Springer Berlin Heidelberg.

Curtmola, R., Garay, J., Kamara, S., and Ostrovsky, R. (2006). Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 79–88, New York, NY, USA. Association for Computing Machinery.

Demertzis, I., Chamani, J. G., Papadopoulos, D., and Papamanthou, C. (2019). Dynamic searchable encryption with small client storage. Cryptology ePrint Archive, Report 2019/1227. https://eprint.iacr.org/2019/1227.

Etemad, M., Küpçü, A., Papamanthou, C., and Evans, D. (2018). Efficient dynamic searchable encryption with forward privacy. In *Proceedings on Privacy Enhancing Technologies*, volume 1, pages 5–20, Berlin. Sciendo.

Frimpong, E., Bakas, A., Dang, H.-V., and Michalas, A. (2020). Do not tell me what i cannot do! (the constrained device shouted under the cover of the fog): Implementing symmetric searchable encryption on constrained devices (extended version). Cryptology ePrint Archive, Report 2020/176. https://eprint.iacr.org/2020/176.

Kamara, S. and Moataz, T. (2017). Boolean searchable symmetric encryption with worst-case sub-linear complexity. In Coron, J.-S. and Nielsen, J. B., editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 94–124, Cham. Springer International Publishing.

Kamara, S., Moataz, T., and Ohrimenko, O. (2018). Structured encryption and leakage suppression. In Shacham, H. and Boldyreva, A., editors, *Advances in Cryptology – CRYPTO 2018*, pages 339–370, Cham. Springer International Publishing.

Kamara, S., Papamanthou, C., and Roeder, T. (2012). Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 965–976, New York, NY, USA. Association for Computing Machinery.

Katz, J. and Lindell, Y. (2014). *Introduction to Modern Cryptography*. Chapman and Hall / CRC Press, second edition.

Kim, H., Hahn, C., and Hur, J. (2019). Analysis of forward private searchable encryption and its application to multi-client settings. In *2019 Eleventh International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 529–531.

Lin, J., Yu, W., Zhang, N., Yang, X., Zhang, H., and Zhao, W. (2017). A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet of Things Journal*, 4(5):1125–1142.

Song, D., Wagner, D., and Perrig, A. (2000). Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*, pages 44–55.

Stefanov, E., Papamanthou, C., and Shi, E. (2014). Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium*, volume 71, pages 72–75.

Zhang, Y., Katz, J., and Papamanthou, C. (2016). All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 707–720, Austin, TX. USENIX Association.