Verifiable Dynamic Symmetric Searchable Encryption Optimality and Forward Security

Raphaël Bost*†
raphael_bost@alumni.brown.edu

Pierre-Alain Fouque†
pierre-alain.fouque@ens.fr

David Pointcheval‡
david.pointcheval@ens.fr

Abstract

Symmetric Searchable Encryption (SSE) is a very efficient and practical way for data owners to outsource storage of a database to a server while providing privacy guarantees. Such SSE schemes enable clients to encrypt their database while still performing queries for retrieving documents matching some keyword. This functionality is interesting to secure cloud storage, and efficient schemes have been designed in the past. However, security against malicious servers has been overlooked in most previous constructions and these only addressed security against honest-but-curious servers.

In this paper, we study and design the first efficient SSE schemes provably secure against *malicious* servers. First, we give lower bounds on the complexity of such verifiable SSE schemes. Then, we construct generic solutions matching these bounds using efficient verifiable data structures. Finally, we modify an existing SSE scheme that also provides forward secrecy of search queries, and make it provably secure against active adversaries, without increasing the computational complexity of the original scheme.

1 Introduction

Searchable Encryption schemes encode a database of documents into a data structure that can be outsourced to a remote server, in order to allow keyword-based search while preserving the privacy of both the database and the queries. Cloud-based storage is a direct application of these schemes: cloud providers supply data storage with high-availability guarantees and strong security protection. While availability failures can be immediately detected by users, security breaches can be known long after their use by malicious people. Even if the cloud providers should do their best to guarantee both availability and security, external or even internal people may want to (illegally) get access to the data or to alter computations performed on the data. As a consequence, data privacy and correctness of the computations should be guaranteed *by design*.

Since external storage is shared between many users with huge amounts of data to be processed, these security guarantees should be at a low cost for both the server and the client. The goal of searchable encryption schemes is to address this problem: the server itself should not learn anything about the search

*Direction Générale de l'Armement - Maitrise de l'Information. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DGA or the French Government.

†Université de Rennes 1, IRISA

‡Ecole Normale Supérieure, Département d'Informatique

queries (which keywords the user is looking for), nor about the database updates triggered by the client. Symmetric Searchable Encryption [SWP00] provides elegant and practical solutions to this problem using symmetric cryptographic primitives, while avoiding costly oblivious RAM-based systems [GO96], at the price of a small amount of leakage of information. For example, classical constructions for efficient SSE schemes encrypt a reversed index which maps each keyword w with the list $\mathsf{DB}(w)$ of the indices of the matching documents. Usually, the keywords are deterministically encrypted using a pseudo-random function, as well as the list $\mathsf{DB}(w)$. In this case, the size pattern (the total number of document/keyword pairs stored), the search pattern (the repetition of searched keywords), and the access pattern (the document indices of a search result) cannot be hidden to the server, since the encryption used for the keywords and indices is deterministic. Such leakage is tolerated though, to improve performance, but may lead to some attacks [CGPR15].

Our Contributions. This paper studies the efficiency and security of dynamic, verifiable and forward secure SSE schemes. We give lower bounds on dynamic and verifiable SSE and build optimal constructions achieving these bounds. Then, we revisit a dynamic and forward secure SSE scheme described by Stefanov *et al.* [SPS14], and make it verifiable by fixing the propositions of Stefanov *et al.* and providing a complete security proof. These results are summarized in Table 1.

Our goal is to provide efficient dynamic SSE solutions secure against *malicious* servers: the server can return incorrect answers to the clients. Consequently, we formally define the *verifiability* of an SSE scheme: the client must be able to verify that the answers output by a search query are indeed correct, *i.e.* that the server provided the indices of all the documents matching the keywords. In particular, in case of empty search results, the client must check that indeed no document matches the searched keyword. This last point is crucial, and previous informal solutions addressing malicious behaviors of the server, such as [SPS14], were unable to provide such guarantees.

To this end, we first study this problem from a theoretical point of view and give lower bounds on the computational complexity of search and update queries. Efficient schemes have time complexity close to optimal in terms of the number of answers for the search operations and the number of distinct keywords for the update operations. More precisely, we show that if the client's private storage is not linear in the number of distinct keywords in the database, the verification of an SSE scheme has to have a logarithmic overhead for either search or update queries. We prove this result by exhibiting a reduction between this problem and memory checkers (see Section 3). Then, we present generic solutions for the construction of verifiable SSE schemes that match these lower bounds (see Section 5). The first solution relies on Merkle tree-like data structures and Set Hashing, while the second one uses cryptographic accumulators instead of Merkle trees.

Yet, these schemes do not provide the strongest property of forward privacy: if the user looks for a keyword w and later adds a new document containing w, the server will learn that this new document has a keyword that has been looked for in the past. We think that this property is essential in practice as SSE schemes must limit, as much as possible, the amount of information leaked to the server. In Section 6, we present an efficient dynamic and verifiable SSE scheme that achieves forward secrecy. We show how we can adapt previous solutions proposed by Stefanov *et al.* [SPS14] in order to add verifiability, while keeping good performances, both in theory and in practice, and without being subject to the weaknesses of the original propositions (*cf.* Section 6.2). Indeed, we give a full security proof for our construction.

Related Work. Curtmola *et al.* [CGKO06] gave the first efficient SSE construction, achieving *sublinear search time*: the previous schemes, like the seminal work of Song *et al.* [SWP00], required search complexity linear in the number of documents stored in the database. They also improved the previous security models

and introduced the notion of adaptive security for SSE. A highly efficient construction was presented by Cash *et al.* in [CJJ+13]. This construction also supports complex queries involving several keywords, as does the scheme of Pappas *et al.* [PKV+14]. Cash and Tessaro [CT14] studied the trade-off between locality and server storage size of SSE schemes.

Dynamic SSE was first considered by Song *et al.* [SWP00], but no solution with sublinear search time existed before the work of Kamara *et al.* [KPR12]. Recently, two new dynamic SSE schemes have been proposed. The first one, by Cash *et al.* [CJJ+14], extends [CJJ+13], and show that SSE is feasible on very large databases. The other one, by Stefanov *et al.* [SPS14], is the first forward secure construction, and remains practical. However, the problem of malicious servers has not been studied, except in [SPS14], but, as we will see later, their proposition is flawed.

A way to construct SSE with very little leakage would be using ORAM, either generically [GO96, SvDS+13, GMOT12], or through garbled RAM programs [LO13, WNL+14, WHC+14]. However, these techniques are less efficient by several orders of magnitudes than specifically designed SSE schemes. Indeed, they incur large bandwith overhead and need to work on large blocks to be practically efficient (a few kilobytes, as the documents indices are encoded over a few tens of bits).

As we stated before, another issue when outsourcing computation or storage is correctness. This problem has been widely studied, in particular in the case of storage, as the memory checking problem [BEG+91, DNRV09]. In this particular setting, some lower bounds have been proven by Dwork *et al.* [DNRV09].

Authenticated/verifiable hash tables have also been extensively studied [TT05, PT08, PTT09, CL02, STSY01], using different cryptographic primitives, as hash functions, or cryptographic accumulators. The main difference with verifiable SSE is that a hash table can contain at most one element per keyword, and that the queried keys are not meant to be hidden to the server, only soundness of the returned results is ensured.

Table 1 – Comparison of SSE schemes. In this table, N is the total number of document/keyword pairs, d is the number of documents, m is the size of the result, $|W|$ is the number of distinct keywords, $0 < \varepsilon < 1$ is a fixed constant, α is the number of times the queried keyword was historically added to the database, and μ is the number of updated document/keyword pairs. For every scheme, except the one by Kurosawa and Ohtaki [KO13], the storage is linear in the number of document keyword pairs. For [KO13], the storage space is $O(|W|.d)$.

Scheme		Time Complexity		Forward Secure	Verifiable
		Search	Update		
Previous work					
Cash *et al.* [CJJ+14]	$\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$	$O(m)$	$O(\mu)$	✗	✗
Stefanov *et al.* [SPS14]	Linear SPS	$O(\alpha + \log N)$	$O(\mu \log^2 N)$	✓	✗
	Sublinear SPS	$O(m \log^3 N)$			
Kurosawa and Ohtaki [KO13]		$O(d)$	$O(\mu d)$	✗	✓
This work					
Lower Bound (Sec.5)	GSV-Hash	$O(m + \log \|W\|)$	$O(\mu \log \|W\|)$	✗	✓
Optimal Search (Sec.5)	GSV-Acc.-Pairing	$O(m)$	$O(\mu \|W\|^{\varepsilon})$	✗	✓
Optimal Update (Sec.5)	GSV-Acc.-RSA	$O(m + \|W\|^{\varepsilon})$	$O(\mu)$	✗	✓
Verifiable SPS	Linear (Sec. 6.3)	$O(\alpha + \log^2 N)$	$O(\log^2 N)$	✓	✓
	Sublinear (Sec. 6.4)	$O(m \log^3 N)$			

Finally, verifiable SSE was first studied by Kurosawa and Ohtaki in [KO12], and then extended to the dynamic setting in [KO13]. Yet, their solution is inefficient, both from a computational point of view – as searches or update need work linear in the number of documents in the database – and from a storage point of view.

2 Definitions

In this paper, the security parameter will be denoted λ. We will also use the standard definitions of pseudo-random functions (PRF) and semantically secure symmetric encryption schemes [Gol04]. For the sake of simplicity, we suppose that the keys are strings of λ bits, and that the key generation algorithm uniformly chooses a key in $\{0,1\}^\lambda$. Unless otherwise specified, we always consider probabilistic algorithms/protocols running in time polynomial in λ, also called *efficient*. Adversaries are probabilistic polynomial time (ppt) algorithms and $\mathrm{negl}(\lambda)$ denotes a negligible function in λ.

2.1 Notations and Tools

2.1.1 Games.

Our security and correctness notions are defined using the code-based games introduced in [BR06]. A game G is a set of oracle procedures – including an initialization Init procedure and a finalization Final procedure – that is executed with an adversary A, *i.e.* A has access to the procedures, with some possible restrictions. For instance, the Init oracle is always the first one to be called and Final the last one, once A halted, taking A's output as input. The output of Final is called the output of the game and is denoted $G^A(\lambda)$. When Final is omitted, it just forwards the adversary's output.

At startup, the boolean variables are initialized to false and the integer variables to 0. When the variable T is a dictionary, $T[v]$ denotes the item associated to v, if there is one, whereas $\perp$ denotes the absence of this item.

2.1.2 Protocols.

In the paper, we will construct and use some two-party protocols, involving a client C and a server S. We will denote a protocol P as

$$P(\mathtt{input}_C; \mathtt{input}_S) = (P_C(\mathtt{input}_C), P_S(\mathtt{input}_S))$$

meaning that P_C (resp. P_S) is executed by the client (resp. the server) with input $\mathtt{input}_C$ (resp. $\mathtt{input}_S$). We write

$$(\mathtt{out}_C; \mathtt{out}_S) \xleftarrow{\$} C(\mathtt{input}_C) \leftrightarrow S(\mathtt{input}_S)$$

to mean that $\mathtt{out}_C$ and $\mathtt{out}_S$ are the outputs of the interaction between C on input $\mathtt{input}_C$ and S on input $\mathtt{input}_S$. When C and S run a protocol P, we simply write $(\mathtt{out}_C; \mathtt{out}_S) \xleftarrow{\$} P(\mathtt{input}_C; \mathtt{input}_S)$. In the following, we will often consider the messages τ sent by the client (the transcript) as part of the client's output.

2.1.3 MAC: Message Authentication Code.

A message authentication code MAC [Gol04] is a deterministic pair of algorithms $(\mathsf{MAC}, \mathsf{Verif})$ with $\mathsf{MAC} : \mathcal{K} \times \{0,1\}^m \to \{0,1\}^t$ and $\mathsf{Verif} : \mathcal{K} \times \{0,1\}^m \times \{0,1\}^t \to \{\texttt{ACCEPT}, \texttt{REJECT}\}$ where $\mathcal{K}$ is the key space. It should be *correct*, *i.e.* for all $K \in \mathcal{K}$, $\mathsf{Verify}_K(x, \mathsf{MAC}_K(x)) = \texttt{ACCEPT}$, and it should withstand *chosen-message attacks*: it must be impossible, without the key K, to forge a new valid (accepted) pair of message/code. The exact security game CMA is formalized in Figure 6 of Appendix A, and any polynomial adversary A should not win the game with non-negligible advantage $\mathbf{Adv}_A^{\mathsf{MAC}}(\lambda)$. In the paper, we will suppose that MAC is a pseudo-random function and that $\mathsf{Verif}_K(x, s) = \texttt{ACCEPT}$ if $s = \mathsf{MAC}_K(x)$ and REJECT otherwise.

2.1.4 Authenticated Encryption with Associated Data.

The security of our construction heavily relies on authenticated encryption with associated data (AEAD) schemes. Authenticated encryption (resp. decryption) of m with associated data a under key sk is denoted $\mathsf{AEnc}(sk, a, m)$ (resp. $\mathsf{ADec}(sk, a, m)$). Intuitively, AEAD ensures that only the encryption of m with associated data a will decrypt to m and that it is impossible, without the key sk, to forge a ciphertext that will correctly decrypt (*i.e.* whose decryption will not return REJECT). Full security definitions of authenticated encryption are given in [RBB03]. For the sake of simplicity in the algorithms descriptions, we will omit the initialization vector (IV) and consider the encryption scheme as probabilistic.

2.2 SSE: Symmetric Searchable Encryption

To define symmetric searchable encryption schemes, we follow the formalization of [CGKO06] with the modifications of [CJJ+14]. A *database* $\mathsf{DB} = (\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d$ is a tuple of identifier/keyword-set pairs with $\mathsf{ind}_i \in \{0,1\}^\lambda$ and $\mathsf{W}_i \subseteq \{0,1\}^*$. The set of keywords of the database DB is $\mathsf{W} = \cup_{i=1}^d \mathsf{W}_i$. We set $m = |\mathsf{W}|$ to be the total number of keywords in DB, and $N = \sum_{i=1}^d |\mathsf{W}_i|$ to be the number of document/keyword pairs. $\mathsf{DB}(w)$ denotes the set of documents containing keyword w, *i.e.* $\mathsf{DB}(w) = \{\mathsf{ind}_i | w \in \mathsf{W}_i\}$. N can also be written as $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$.

A *dynamic searchable encryption scheme* $\Pi = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ consists of one algorithm and two protocols between a client and a server:

- $\mathsf{Setup}(\mathsf{DB})$ is an algorithm that takes as input a database DB. It outputs a pair $(\mathsf{EDB}, K, \sigma)$ where K is a secret key, EDB the encrypted database, and σ the client's state.

- $\mathsf{Search}(K, \sigma, q; \mathsf{EDB}) = (\mathsf{Search}_C(K, \sigma, q), \mathsf{Search}_S(\mathsf{EDB}))$ is a protocol between the client with input the key K and a search query q, and the server with input EDB. We write $(V, \sigma', \tau) \xleftarrow{\$} \mathsf{Search}(K, q; \mathsf{EDB})$ to mean that V, σ' and τ are sampled by running the protocol with these inputs, V being the client output, σ' the new state of the client, τ the messages sent by the client (the transcript). V (and σ') can take the special value REJECT. In this paper, a search query consists of a single keyword w.

- $\mathsf{Update}(K, \sigma, \mathsf{op}, \mathsf{in}; \mathsf{EDB}) = (\mathsf{Update}_C(K, \sigma, \mathsf{op}, \mathsf{in}), \mathsf{Update}_S(\mathsf{EDB}))$ is a protocol between the client with input the key K and state σ, an operation op and an input in parsed as the identifier ind and a set W of keywords, and the server with input EDB. As in previous papers [CJJ+14], the update operations are taken from the set $\{\mathsf{add}, \mathsf{edit}^+, \mathsf{del}, \mathsf{edit}^-\}$, meaning, respectively, the addition of a full document, of a document/keyword pair, the deletion of a full document and the deletion of a single

Init(DB)
  $(\mathsf{EDB}, K, \sigma) \leftarrow \mathsf{Setup}(\mathsf{DB})$
  return EDB

Search(q)
  $(V, \sigma, \tau) \xleftarrow{\$} \mathsf{Search}(K, \sigma, q; \mathsf{EDB})$
  if $V \neq \mathsf{DB}(q)$ **or** $\sigma = \texttt{REJECT}$
    $\texttt{win} \leftarrow \texttt{true}$
  return τ

Final()
  return win

Update(op, in)
  $(\sigma', \tau; \mathsf{EDB}') \xleftarrow{\$}$
    $\mathsf{Update}(K, \sigma, \mathsf{op}, \mathsf{in}; \mathsf{EDB})$
  if $\sigma' = \texttt{REJECT}$ **then**
    $\texttt{win} \leftarrow \texttt{true}$
  else
    $\mathsf{DB} \leftarrow \mathsf{Apply}(\mathsf{DB}, \mathsf{op}, \mathsf{in})$
    $\mathsf{EDB} \leftarrow \mathsf{EDB}', \sigma \leftarrow \sigma'$
  end if
  return τ

Figure 1 – Correctness game for SSE SSECORR.

pair. We write $(\sigma', \tau; \mathsf{EDB}') \xleftarrow{\$} \mathsf{Update}(K, \sigma, \mathsf{op}, \mathsf{in}; \mathsf{EDB})$ to mean that σ', τ and EDB' are sampled by running the protocol, σ' being the output of the client *i.e.* its new state, τ being the messages sent by the client and EDB' be the server output *i.e.* the updated encrypted database. Note that σ' can take the special value REJECT.

2.2.1 Correctness.

The correctness of an SSE scheme is the basic property we want to ensure: the search protocol must return the correct result for every query, except with negligible probability. We formally define correctness with the security game SSECORR defined in Figure 1. The game uses the function Apply that outputs DB updated according to the input operation op, and the input in for that operation.

Definition 1 (SSE Correctness)**.** *An SSE scheme* Π *is* correct *if for all adversary* A, $\mathbf{AdvCor}_A^{\mathsf{SSE},\Pi}(\lambda)$ *is negligible in* λ*, where*

$$\mathbf{AdvCor}_A^{\mathsf{SSE},\Pi}(\lambda) = \mathbb{P}[\text{SSECORR}_\Pi^A(\lambda) = 1].$$

2.2.2 Soundness.

The soundness for SSE is the key new property introduced in this paper. As for regular soundness definitions, it describes the fact that no adversary can cheat the client, and make him accept incorrect search results.

To give a proper soundness definition, we use the game SSESOUND defined in Figure 2. The game closely follows the game used to define soundness of interactive provers [Gol04]: the client should not accept an invalid search result. Also, the dynamism of the database raises a difficult point: the verification has to be done over the current version of the database, yet this one must not be modifiable by a malicious server. Hence, SSESOUND does not apply the update operation on the database when the client rejects the execution of the Update protocol with the server.

Definition 2 (SSE Soundness)**.** *An SSE scheme* Π *is* sound *(or* verifiable*) if for all adversary* A, $\mathbf{AdvSnd}_A^{\mathsf{SSE},\Pi}(\lambda)$ *is negligible in* λ*, where*

$$\mathbf{AdvSnd}_A^{\mathsf{SSE},\Pi}(\lambda) = \mathbb{P}[\text{SSESOUND}_\Pi^A(\lambda) = 1].$$

and in the game SSESOUND*,* ChallengeUpdate *and* ChallengeVerify *are called only once, just before* Final.

2.2.3 Confidentiality.

The confidentiality definition of an SSE scheme uses the real word vs. ideal word formalization [CGKO06, KPR12, CJJ+14]. It is parametrized by a *leakage function* $\mathcal{L}$ describing what the protocols leak to the adversary, and formalized as a stateful algorithm.

It uses the games SSEREAL and SSEIDEAL defined in Figure 3, similarly to the real and ideal games in MPC security definitions. In the real game the adversary chooses a database DB and gets back EDB. Then, it adaptively runs Search and Update protocols on inputs of its choice, and is given the transcripts of these protocols. The Final call simply forwards the bit b output by the adversary. In the ideal game, these transcripts are generated by a simulator S, an efficient algorithm, helped by the outputs of the leakage function. Finally, the scheme will be secure if no efficient adversary can distinguish between the real and the ideal games.

Definition 3 (SSE Confidentiality)**.** *Let* $\Pi = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *be a dynamic SSE instantiation,* A *and* S *probabilistic polynomial-time algorithms, and let* $\mathcal{L}$ *be a stateful algorithm.*

$\underline{\mathsf{Init}(\mathsf{DB})}$
$(\mathsf{EDB}, K, \sigma) \leftarrow \mathsf{Setup}(\mathsf{DB})$
return EDB
$\underline{\mathsf{Search}(q)}$
$(V, \sigma', \tau) \xleftarrow{\$} \mathsf{Search}_C(K, \sigma, q) \leftrightarrow A$
if $V \neq$ `REJECT` **then**
$\sigma \leftarrow \sigma'$
if $V \neq \mathsf{DB}(q)$ **then** `win` $\leftarrow$ `true`
end if
return τ

$\underline{\mathsf{Final}()}$
return `win`
$\underline{\mathsf{Update}(\mathsf{op}, \mathsf{in})}$
$(\sigma', \tau; \mathsf{EDB}) \xleftarrow{\$}$
$\mathsf{Update}_C(K, \sigma, \mathsf{op}, \mathsf{in}) \leftrightarrow A$
if $\sigma' \neq$ `REJECT` **then**
$\mathsf{DB} \leftarrow \mathsf{Apply}(\mathsf{DB}, \mathsf{op}, \mathsf{in})$
$\mathsf{EDB} \leftarrow \mathsf{EDB}', \sigma \leftarrow \sigma'$
end if
return τ

Figure 2 – Soundness game for SSE SSESOUND. The notation $\leftrightarrow A$ represents interactions with the adversary.

SSEREAL$_\Pi$
$\underline{\mathsf{Init}(\mathsf{DB})}$
$(\mathsf{EDB}, K, \sigma) \leftarrow \mathsf{Setup}(\mathsf{DB})$
return EDB
$\underline{\mathsf{Search}(q)}$
$(V, \sigma', \tau) \xleftarrow{\$} \mathsf{Search}_C(K, \sigma, q) \leftrightarrow A$
if $V \neq$ `REJECT`
$\sigma \leftarrow \sigma'$
return τ
$\underline{\mathsf{Update}(\mathsf{op}, \mathsf{in})}$
$(\sigma', \tau; \mathsf{EDB}') \xleftarrow{\$} \mathsf{Update}_C(K, \sigma, \mathsf{op}, \mathsf{in}) \leftrightarrow A$
if $\sigma' \neq$ `REJECT`
$\sigma \leftarrow \sigma', \mathsf{EDB} \leftarrow \mathsf{EDB}'$
return τ

SSEIDEAL$_{S,\mathcal{L}}$
$\underline{\mathsf{Init}(\mathsf{DB})}$
$(\mathsf{EDB}, \sigma_S) \leftarrow S(\mathcal{L}(\mathsf{DB}))$
return EDB
$\underline{\mathsf{Search}(q)}$
$(\sigma'_S, \tau) \xleftarrow{\$} S(\sigma_S, \mathcal{L}(q)) \leftrightarrow A$
if $\sigma'_S \neq$ `REJECT`
$\sigma_S \leftarrow \sigma'_S$
return τ
$\underline{\mathsf{Update}(\mathsf{op}, \mathsf{in})}$
$(\sigma'_S, \tau) \leftarrow S(\sigma_S, \mathcal{L}(\mathsf{op}, \mathsf{in})) \leftrightarrow A$
if $\sigma'_S \neq$ `REJECT`
$\sigma_S \leftarrow \sigma'_S$
return τ

Figure 3 – SSE security games SSEREAL (left) and SSEIDEAL (right). The notation $\leftrightarrow A$ represents interactions with the adversary.

We define the advantage of A against S in the SSE confidentiality security game as $\mathbf{AdvConf}^{\mathsf{SSE},\Pi}_{A,S,\mathcal{L}}(\lambda)$ *where*

$$\mathbf{AdvConf}^{\mathsf{SSE},\Pi}_{A,S,\mathcal{L}}(\lambda) = \left|\mathbb{P}[\text{SSEREAL}^{\Pi}_{A}(\lambda) = 1] - \mathbb{P}[\text{SSEIDEAL}^{\Pi}_{A,S,\mathcal{L}}(\lambda) = 1]\right|.$$

We say that Π *is a* $\mathcal{L}$-adaptively-secure instantiation *if for all adversaries A, there exists an efficient algorithm S such that* $\mathbf{AdvConf}^{\mathsf{SSE},\Pi}_{A,S,\mathcal{L}}(\lambda) \leq \text{negl}(\lambda)$.

Query Pattern. In almost all SSE work, the schemes leak the repetion of tokens sent by the client to the server. For example, two search queries using the same keywords will leak the repetition of these keywords. In previous works [CGKO06, CJJ+13], this is called the *search pattern*. In other constructions (*e.g.* [CJJ+14]), this type of leakage is not limited to search queries: repetition of keywords for both search and update queries leaks. Hence, we call it the *query pattern*. The query pattern $\mathsf{qp}(x)$ of x is constructed as follows: initially qp creates an empty list L as state and sets a counter $i \leftarrow 0$, and then on input x, it increments i, adds (i, x) to L and outputs $\mathsf{qp}(x) = \{j | (j, x) \in L\}$. In the following, for the sake of simplicity, we denote $\overline{x} = \mathsf{qp}(x)$.

3 Lower Bounds

Before giving some constructions of verifiable SSE, we want to show that protection against active adversaries has an inherent cost, and to lower bound this cost. For semi-honest adversaries, lower bounds are trivial: search cannot be done in less than $\Omega(m)$ where m is the number of results for a query, and update has to run in $\Omega(1)$ per modified document/keyword pair.

For malicious adversaries, the result is not as straightforward. Fortunately, we can rely on the literature on authenticated data structures (namely authenticated hash tables) [TT05, PT08, PTT09] and memory checking [BEG+91, NR05, DNRV09] to have a better insight into this question. Actually, in the next section, we show how to reduce memory checking to verifiable SSE, and, using the lower bound result of [DNRV09], give a general computational lower bound on the Search and Update algorithms of *any* SSE scheme secure against malicious adversaries. Then, based on [TT05], we argue that, if we only use symmetric primitives for verification, we cannot actually hope for less than the lower bounds described in the introduction, namely $O(m + \log |W|)$ for Search and $O(\log |W|)$ for Update.

3.1 Memory Checking

Memory checking is the problem of outsourcing memory to an untrusted party while ensuring authenticity and using limited (trusted) local storage. A memory checker $\mathcal{C}$ is a probabilistic algorithm that receives from the user read and write requests, and, by making its own requests to the untrusted remote memory, and accessing a small private memory, ensures that the original requests were either answered correctly, or that a fault was reported.

The formal definition of a memory checker can be found in [DNRV09]. In particular, the authors define $\mathcal{C}$ as a (Σ, n, q, s)-*checker* as a checker that can be used to store a database of n indices with query complexity q (the average number of remote memory accesses necessary for each memory query processed by the memory checker) and space complexity s, where the secret and public memory are over the alphabet Σ.

The authors of this work prove a lower bound on q, assuming that the private memory is not "large": s is $O(n^{\alpha})$, with $0 \leq \alpha < 1$. The actual theorem is given as follows:

Theorem 1 (Theorem 3.1 of [DNRV09])**.** *Let $\mathcal{C}$ be a (Σ, n, q, s) memory checker with $s \leq n^{1-\varepsilon}$ for some $\varepsilon > 0$ and $|\Sigma| \leq n^{poly\ \log n}$. It must be that $q = \Omega(\frac{\log n}{\log \log n})$.*

3.2 A General Lower Bound on Verifiable SSE

The problem of verifiable SSE is somewhat similar to the one of memory checking, with two caveats:

- SSE schemes must be able to store a variable number of indices (*i.e.* keywords);
- in SSE, each keyword can match zero, one, or more documents, instead of having one keyword matching exactly one value in the case of memory checkers.

SSE actually supports more features than regular memory checking, and writing a reduction from memory checkers to VSSE is straightforward. Finally, the lower bounds on memory checkers from Theorem 1 will also transfer to VSSE, as stated in Theorem 2 and proven in Appendix D.

Theorem 2 (Lower bound of verifiable searchable symmetric encryption)**.** *Let* Π *be a VSSE scheme with client memory of size* $s \leq |W|^{1-\varepsilon}$ *for some* $\varepsilon > 0$*, where* w *is the number of distinct stored keywords (keywords with at least one matching document). Then, either* Search *queries have computational complexity* $\Omega(\mathsf{max}(\frac{\log |W|}{\log \log |W|}, m))$ *or* Update *queries have computational complexity* $\Omega(\frac{\log |W|}{\log \log |W|})$.

We emphasize that this result bounds the minimal complexity of the costliest operation between Search and Update and do not imply that both operation complexity is lower bounded by $\Omega(\frac{\log |W|}{\log \log |W|})$.

3.3 Lower Bound for Practical Construction

Theorem 2 gives us a general bound for generic searchable encryption. However, in this paper, we focus more on *symmetric* searchable encryption schemes that (mostly) use symmetric primitives like hash functions or block ciphers. More exactly, all the existing schemes use deterministic encryption to perform the search and update protocols (deterministically too), and we might want to use similar techniques for verification.

One technique we can rely on makes use of cryptographic hash functions for verifications. The optimality of such constructions has been studied by Tamassia and Triandopoulos [TT05], who showed that one cannot do better than having $\Omega(\log n)$ verification and update computational complexity to authenticate a dictionary with n entries through hashing (Theorem 6 in [TT05]). Using the same reduction as in Theorem 2, we can show that if we rely on hashing for verification purpose in SSE, both Search and Update protocols cannot be run in less than $\Omega(\log |W|)$ time. Taking into account the search operation's lower bound for general SSE, we actually have that the minimal search complexity is $\Omega(\min(m, \log |W|))$.

An other well-known technique in authenticated data structures is cryptographic accumulators. In particular, it has been used in several contributions for verifiable hash tables [CL02, STSY01]. The most efficient construction [PTT09] performs constant (expected) query time (for both proof generation and verification). However, the expected update time on the server side is $O(n^{\varepsilon})$ for a fixed $0 < \varepsilon < 1$ (it is constant on the client side), and we cannot reach the general lower bound of Section 3.2 using cryptographic accumulators. For practical considerations, we must also consider the fact that the cryptographic operations needed for accumulators (exponentiation of large integers or bilinear maps) are a lot more computationally expensive than hashing, and despite a better asymptotical complexity, they might not compare favorably with hashing-based techniques.

4 Construction of Verifiable Dynamic SSE: General Ideas and Tools

When verifying the results of a search query, one has to check two points: first that every result matches the query, and then that every matching result has been returned. This crucial fact was already emphasized in [SPS14].

If the first point is relatively easy to ensure for static databases, things become more complicated when considering dynamic databases. We must indeed prevent *replay attacks*, where the server returns a result that *used to* belong to the database but that no longer does (the document-keyword pair was deleted). If using MACs over the document-keyword pairs would have been enough for static SSE, this is no longer secure on its own for the dynamic case. Somehow, the client has to store some kind of digest of the outsourced database. For efficiency, we want this digest to be much smaller than the database.

The second point, is no less important, *e.g.* when a search query has no matching result. In this case, how can we prove that there are actually no matches to the query? Similarly, we must prevent the server to return an empty list of results when there are actual matches. This last remark was not taken into account in [SPS14], and the modification the authors present to make their scheme secure against malicious adversaries does not prevent this attack.

4.1 Verifiable Hash Tables

A central component of our verifiable SSE construction is *verifiable hash tables*. It implements the functionality of a regular (static) hash table, but also provides a proof that, when querying a key in the hash table, the returned element is the right one, and that there are no associated element when querying a key that is not present in the hash table.

More formally, a verifiable hash table is a tuple of algorithms $\Theta = (\mathsf{VHTSetup}, \mathsf{VHTGet}, \mathsf{VHTVerify}, \mathsf{VHTUpdate}, \mathsf{VHTRefresh})$:

- $\mathsf{VHTSetup}(T)$ takes as input a hash table T and outputs $(K_{\mathsf{VHT}}, \mathsf{VHT}, \sigma_{\mathsf{VHT}})$ where K_{VHT} is a private key, VHT is the verifiable hash table data structure (possibly including a public key), and σ_{VHT} the client's state.
- $\mathsf{VHTUpdate}(T, \mathsf{VHT}, u)$ takes as input a hash table T together with its verifiable data structure VHT, and updates all of these according to the update operation u coming from a predefined update set (*e.g.* replacement of a value in the table, deletion of a key, ...). It outputs the new verifiable hash table VHT' and an update proof π. Its complexity for a table of size n is denoted $T^{\mathrm{prf}}_{\mathsf{up}}(n)$.
- $\mathsf{VHTRefresh}(K_{\mathsf{VHT}}, \sigma_{\mathsf{VHT}}, \pi, u)$ refreshes the digest σ_{VHT} (*i.e.* the client's state) according to the update u, using the proof π generated by a previous $\mathsf{VHTUpdate}$ call. Its complexity for a table of size n is denoted $T^{\mathrm{chk}}_{\mathsf{up}}(n)$.
- $\mathsf{VHTGet}(T, \mathsf{VHT}, \mathsf{hkey})$ outputs the tuple (v, π) where v is the value associated to hkey in T, and π a proof. For a table of size n, its complexity is denoted $T^{\mathrm{prf}}_{\in}(n)$ when $v \neq \perp$ (hkey matches a value), and $T^{\mathrm{prf}}_{\perp}(n)$ when $v = \perp$.
- $\mathsf{VHTVerify}(K_{\mathsf{VHT}}, \sigma_{\mathsf{VHT}}, \mathsf{hkey}, v, \pi)$ returns `ACCEPT` or `REJECT`. For a table of size n, its complexity is denoted $T^{\mathrm{chk}}_{\in}(n)$ when $v \neq \perp$, and $T^{\mathrm{chk}}_{\perp}(n)$ when $v = \perp$.

A verifiable hash table must have two properties: completeness (VHTGet should return the value associated to hkey in T together with a valid proof) and soundness (it is hard for the server, without σ_{VHT}, to forge a valid proof, even if he saw a polynomial number of valid proofs, and tried to corrupt the client's state). These properties are formalized by games VHTCOMP and VHTSOUND, as described in Figure 7 in Appendix B. We will also use *static* verifiable hash tables. In this case, $\mathsf{VHTUpdate}$ and $\mathsf{VHTRefresh}$ are not implemented, as well as the Update procedures from the security games defined above.

Algorithm 1 Instantiation of verifiable hash table

$\mathsf{VHTSetup}(T)$

$K \leftarrow \mathcal{K}$
Initialize a empty table VHT of size $|T|$
$i \leftarrow 0$
for all $(key, v) \in T$ in ascending lexicographic order over key **do**
  $s \leftarrow \mathsf{MAC}_K(key, v, i)$
  Add (key, v, i, s) to VHT.
  $i{+}{+}$
end for
return (K, VHT)

$\mathsf{VHTGet}(T, x)$

if $T[x] \neq \perp$ **then**
  $(v, i, s) \leftarrow \mathsf{VHT}[x]$
  return $(v, (i, s))$
else
  Using dichotomic search, locate x in T:
  Find i such that $key_i < x < key_{i+1}$
  where $(v_i, i, s_i) = \mathsf{VHT}[key_i]$
  and $(v_{i+1}, i+1, s_{i+1}) = \mathsf{VHT}[key_{i+1}]$
  return $(\perp, (i, key_i, v_i, s_i, key_{i+1}, v_{i+1}, s_{i+1}))$
end if

$\mathsf{VHTVerify}(K, x, v, \pi)$

if $v \neq \perp$ **then**
  Parse π as (i, s)
  return $\mathsf{Verify}_K((x, v, i), s)$
else
  Parse π as $(i, key_-, v_-, s_-, key_+, v_+, s_+)$
  if $key_- < x < key_+$, $\mathsf{Verify}_K((key_-, v_-, i), s_-)$ and $\mathsf{Verify}_K((key_+, v_+, i+1), s_+)$ **then**
    return `ACCEPT`
  else
    return `REJECT`
  end if
end if

We emphasize that we do not directly need any form of confidentiality on the table content nor on the queries. We will actually start from a secure SSE scheme (in this case security means confidentiality) and turn it into a verifiable SSE scheme. The confidentiality of our scheme will be inherited from the confidentiality of the original scheme. The verifiable hash table that will be put on top of the SSE scheme will ensure soundness. Finally, we have to make sure that correctness of the original scheme correctly transfers to the verifiable scheme.

Construction of verifiable hash table has been extensively studied in the literature, in particular in the case of dynamic tables. For now, we suppose that we have access to an implementation of verifiable hash tables.

4.2 Example Construction of a Static VHT

We give an example instantiation of a static VHT in Algorithm 1. The idea of this construction is to sort the elements of the input table T according to their key, and place them in a table at their sorted position and MAC the element together with the position and key. To answer a query with key x, we look for the corresponding entry in T. If there is a matching element, the server returns it, together with the MAC. Verification will just consist of verifying the MAC. If there is no matching element, the server finds the key position in the table using dichotomic search, and returns the position and the neighboring elements together with the associated MACs. This construction is inherently static: if we were able to modify the content of a cell, even if we had MACed the new value as in the setup phase, an adversary would be able to run a replay attack trivially (by submitting a previously seen entry for a given key).

Soundness. The completeness of our instantiation is trivial. The soundness is a bit more involved and relies on the CMA security of the underlying MAC. In Appendix C, we show that for every adversary A of

the VHTSOUND game, there exists an adversary B such that

$$\mathbf{Adv}_{\mathcal{A}}^{\text{VHTSOUND}}(\lambda) \leq \mathbf{Adv}_{\mathcal{B}}^{\text{CMA}}(\lambda).$$

Time Complexity. The verification is always completed in constant time (at most two MAC verifications and two comparisons). On the other side, the proof generation complexity depends on whether the queried key has an associated element. If this is the case, the proof is generated in constant time (access to a hash table). Otherwise, the prover has to run a dichotomic search among n elements, needing $\lceil \log n \rceil$ comparisons.

5 Verifying SSE Optimally

In this section, we describe a solution for verifiable symmetric searchable encryption that matches the lower bound of Section 3. The construction described later in the section must be seen as a generic solution to the verification problem with SSE: we make use of an SSE instantiation Π as a black-box and do not rely on its construction. In other words, the data structures built in this section can be used on top (or more exactly besides) of any SSE scheme and turn it in a VSSE scheme with similar confidentiality guarantees and (additively) logarithmic overhead.

The general principle of this construction is to have a verifiable hash table, whose keys are the database's keywords, and which stores *digests* of an index set. More precisely, we will have a hash table T such that for any $w \in W$, $T[w] = \mathcal{H}(\mathsf{DB}(w))$ where $\{\mathsf{ind}_1, \ldots, \mathsf{ind}_m\} = \mathsf{DB}(w)$, and $\mathcal{H}$ a hash function defined over sets of strings. The main issue here is to make updates fast: we don't want to re-hash the full set $\mathsf{DB}(w)$ when updating the database for keyword w. The hash function $\mathcal{H}$ must have a certain level of homomorphism: we need a set hashing function.

5.1 (Multi)set Hashing

Multiset hashing was introduced by Clarke *et al.* [CDVD+03] and is based on the framework proposed by Bellare and Micciancio [BM97] for incremental hashing.

The constructions work essentially as follows: if H is a regular (*i.e.* non incremental) hash function, $\mathcal{H}(x_1^{m_1}, \ldots, x_n^{m_n})$ is defined as $\sum m_i \cdot H(x_i)$ ($x_i^{m_i}$ represents the element x_i with multiplicity m_i). We want multiset hash functions to be secure in the sense of collision resistance. We refer to [CDVD+03] for the full formal definitions and security requirements for multiset hashing, but `MSet-Mu-Hash` is defined as follows:

$$\begin{aligned} \mathcal{H}(M) : \mathbb{S}^{\mathbb{Z}} &\to \mathbb{F}_q \\ M &\mapsto \Pi_{x \in S} H(x)^{M_x} \end{aligned}$$

where $H : \mathbb{S} \to \mathbb{F}_q$ is a hash function from the set $\mathbb{S}$ to the field $\mathbb{F}_q$. Clarke *et al.* show that $\mathcal{H}$ is collision resistant as long as the discrete log assumption holds in $\mathbb{F}_q$ when H is modeled as a random oracle.

Theorem 3 (Theorem 2 of [CDVD+03])**.** *If the discrete log assumption holds in $\mathbb{F}_q$, and H is a (non-programable) random oracle, the multiset hash function $\mathcal{H}$ is collision resistant.*

This construction clearly fits our functional needs: if $S \subset \mathbb{S}$ is a set, we can easily compute (*i.e.* in constant time) $\mathcal{H}(S \cup \{x\})$ (resp. $\mathcal{H}(S \setminus \{x\})$) from $\mathcal{H}(S)$ and $H(x)$ for $x \in \mathbb{S}$ (resp. $x \in S$) – or even from x if we have access to H – as $\mathcal{H}(S \cup \{x\}) = \mathcal{H}(S) \cdot H(x)$ (resp. $\mathcal{H}(S \setminus \{x\}) = \mathcal{H}(S) \cdot H(x)^{-1}$). As explained before, this will be very helpful to efficiently update the SSE verification data structure.

For the sake of generality, we extend the multiset hashing definition of [CDVD+03], and define a set hashing function as quadruple of probabilistic polynomial algorithms $(\mathcal{H}, \equiv_{\mathcal{H}}, +_{\mathcal{H}}, -_{\mathcal{H}})$ such that $\mathcal{H}$ maps sets included in the superset $\mathbb{S}$, for all $S \subset \mathbb{S}$,

- $\mathcal{H}(S) \equiv_{\mathcal{H}} \mathcal{H}(S)$ (comparability)
- $\forall x \in \mathbb{S} \setminus S, \mathcal{H}(S \cup \{x\}) \equiv_{\mathcal{H}} \mathcal{H}(S) +_{\mathcal{H}} \mathcal{H}(\{x\})$ (insertion incrementality)
- $\forall x \in S, \mathcal{H}(S \setminus \{x\}) \equiv_{\mathcal{H}} \mathcal{H}(S) -_{\mathcal{H}} \mathcal{H}(\{x\})$ (deletion incrementality)

We can also define set hashing resistance as follows ($A^{\mathcal{H}}$ means that A has oracle access to $(\mathcal{H}, \equiv_{\mathcal{H}}, +_{\mathcal{H}}, -_{\mathcal{H}})$).

Definition 4. *A set hash function $\mathcal{H}$ is* collision resistant *if for all adversary A, $\mathbf{AdvColl}^{\mathcal{H}}_A(\lambda)$ is negligible in λ, where*

$$\mathbf{AdvColl}^{\mathcal{H}}_A(\lambda) = \mathbb{P}[(S, M) \leftarrow A^{\mathcal{H}}(\lambda) : S \neq M \text{ and } \mathcal{H}(S) \equiv_{\mathcal{H}} \mathcal{H}(M)]$$

5.2 Optimal Verifiable Construction

Let Π be a dynamic SSE scheme, Θ a verifiable hash table instantiation, and $\mathcal{H}$ a set hashing function. We define $\mathsf{GSV}_{\Pi,\Theta,\mathcal{H}}$ (for Generic SSE Verification) as in Algorithm 2.

Correctness. The correctness of GSV is straightforward given the correctness of Π and completeness of Θ. Using an hybrid proof, we can very easily prove the following proposition.

Proposition 4. *If Π is a dynamic SSE scheme, Θ a verifiable hash table instantiation, and $\mathcal{H}$ a set hashing function, then for every adversary A, there exists adversaries B and B' such that*

$$\mathbf{AdvCor}^{\mathsf{SSE},\mathsf{GSV}_{\Pi,\Theta,\mathcal{H}}}_A(\lambda) \leq \mathbf{AdvCor}^{\mathsf{SSE},\Pi}_B(\lambda) + \mathbf{AdvComp}^{\mathsf{VHT},\Theta}_{B'}(\lambda)$$

Soundness. As expected, the soundness of the GSV scheme directly relies on the soundness of Θ and the collision resistance of $\mathcal{H}$. More formally, we have the following proposition.

Proposition 5. *If Π is a dynamic SSE scheme, Θ a verifiable hash table instantiation, and $\mathcal{H}$ a set hashing function, then for every adversary A, there exists adversaries B, C, and D such that*

$$\mathbf{AdvSnd}^{\mathsf{SSE},\mathsf{GSV}_{\Pi,\Theta,\mathcal{H}}}_A(\lambda) \leq \mathbf{AdvCor}^{\mathsf{SSE},\Pi}_B(\lambda) + \mathbf{AdvColl}^{\mathcal{H}}_C(\lambda) + \mathbf{AdvSnd}^{\mathsf{VHT},\Theta}_D(\lambda)$$

The idea behind the proof is to use hybrids, in which we successively replace the (sometimes incorrect) SSE scheme by direct calls to the database, the calls to the verifiable hash table by direct call to T and the set hashing function by a one-to-one representation of the sets.

Confidentiality. Confidentiality of the composite scheme that is GSV is a little trickier: the informations stored in the verifiable hash table, or the keys, give some informations to the server even if these were previously hidden by the underlying SSE scheme Π. For example, the server will immediately learn the number of distinct keywords in the database from the size of the table. Also, he will learn from the search and update queries the repetition of searched/updated keywords (*cf.* the query pattern in Section 2.2.3).

Algorithm 2 GSV construction

$\underline{\mathsf{Setup}(\mathsf{DB})}$

$(K_\Pi, \mathsf{EDB}_\Pi) \leftarrow \Pi.\mathsf{Setup}(\mathsf{DB})$
Initialize T to an empty hash table
$K_T, K_S \xleftarrow{\$} \{0,1\}^\lambda$
for all $w \in W$ **do**
  $\mathsf{wtag} \leftarrow F(K_T, w)$
  $K_e \leftarrow F(K_S, w)$
  $T[\mathsf{wtag}] \leftarrow \mathcal{H}(F_{K_e}(\mathsf{DB}(w)))$
    ▷ *Apply F_{K_e} to all the elements of $\mathsf{DB}(w)$, and hash the resulting set*
end for
$(\mathsf{VHT}, \sigma_{\mathsf{VHT}}) \leftarrow \mathsf{VHTSetup}(T)$
return $((\mathsf{EDB}_\Pi, T, \mathsf{VHT}), (K_\Pi, K_T, K_S), \sigma_{\mathsf{VHT}})$

$\underline{\mathsf{Search}(K, \sigma, w; \mathsf{EDB})}$

Run $\Pi.\mathsf{Search}(K_\Pi, w; \mathsf{EDB}_\Pi)$. The client gets the result set V

Client:
$\mathsf{wtag} \leftarrow F(K_T, w)$
$K_e \leftarrow F(K_S, w)$
Compute $\mathcal{H}(F_{K_e}(V))$
Send wtag to the server

Server:
$(h, \pi) \leftarrow \mathsf{VHTGet}(T, \mathsf{VHT}, \mathsf{wtag})$
Send (h, π) to the client

Client:
$\mathsf{VHTVerify}(\sigma_{\mathsf{VHT}}, \mathsf{wtag}, h, \pi)$
if not $h \equiv_{\mathcal{H}} \mathcal{H}(F_{K_e}(V))$
  return REJECT
return V

$\underline{\mathsf{Update}(K, \sigma, \mathsf{op}, \mathsf{in}; \mathsf{EDB})}$

Run $\Pi.\mathsf{Update}(K_\Pi, \mathsf{op}, \mathsf{in}; \mathsf{EDB})$
For all modified pairs (w, ind) run the following
Client:
$\mathsf{wtag} \leftarrow F(K_T, w)$
$K_e \leftarrow F(K_S, w)$
$h' \leftarrow \mathcal{H}(F_{K_e}(\{\mathsf{ind}\}))$
Send $(\mathsf{wtag}, h', \mathsf{op})$ to the server

Server:
$h \leftarrow T[\mathsf{wtag}]$
if $\mathsf{op} = \mathsf{add}$ or edit^+
  $\widetilde{h} \leftarrow h +_{\mathcal{H}} h'$
if $\mathsf{op} = \mathsf{del}$ or edit^-
  $\widetilde{h} \leftarrow h -_{\mathcal{H}} h'$
Let u be the replacement of h by $\widetilde{h}$ in $T[\mathsf{wtag}]$
$(\mathsf{VHT}, \pi) \leftarrow \mathsf{VHTUpdate}(T, \mathsf{VHT}, u)$
Send $(\pi, h, \widetilde{h})$ to the client

Client:
$\sigma_{\mathsf{VHT}} \leftarrow \mathsf{VHTRefresh}(\sigma_{\mathsf{VHT}}, u)$
return σ_{VHT}

Proposition 6. *If Π is a $\mathcal{L}_\Pi$-adaptively-secure dynamic SSE scheme, Θ a verifiable hash table instantiation, and $\mathcal{H}$ a set hashing function, then for every adversary A, there exist an adversary B and simulators S and S_Π such that*

$$\mathbf{AdvConf}^{\mathsf{SSE},\mathsf{GSV}_{\Pi,\Theta,\mathcal{H}}}_{A,S,\mathcal{L}_{\mathsf{GSV}}}(\lambda) \leq \mathbf{AdvConf}^{\mathsf{SSE},\Pi}_{B,S_\Pi,\mathcal{L}_\Pi}(\lambda)$$

with $\mathcal{L}_{\mathsf{GSV}}(\mathsf{DB}) = (\mathcal{L}_\Pi(\mathsf{DB}), |W|)$ for the setup, $\mathcal{L}_{\mathsf{GSV}}(\mathsf{DB}, w) = (\mathcal{L}_\Pi(\mathsf{DB}, q), \overline{w})$ for a search query, and $\mathcal{L}_{\mathsf{GSV}}(\mathsf{DB}, \mathsf{op}, \mathsf{in}) = (\mathcal{L}_\Pi(\mathsf{DB}, \mathsf{op}, \mathsf{in}), \mathsf{op}, \overline{w})$ of an update query.

The confidentiality of the Π is considered under the definition of Section 2.2.3, *i.e.* against an active adversary, which is formally different from the previous confidentiality definitions which considered only passive adversaries. However, we can see that, when Π has single round Search and Update protocols, these are equivalent (the adversary cannot mess with the first and single message the client sends).

It is also crucial to see that GSV cannot be forward secure, independently of the chosen SSE scheme: the update part of the leakage function always return the query pattern.

Computational Complexity. To evaluate the computational complexity of the construction, we will use the dynamic SSE construction $\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$ of Cash *et al.* [CJJ+14] which has search complexity linear in the

Table 2 – Computational complexity of GSV with $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$ as (non verifiable) SSE scheme, in function of the VHT instantiation used. The update complexities are given for a single modified keyword/document pair.

VHT Instantiation		Search		Update	
		Server	Client	Server	Client
General VHT		$O(T_{\in}^{\mathrm{prf}}(\lvert W\rvert)+m)$ or $O(T_{\perp}^{\mathrm{prf}}(\lvert W\rvert))$	$O(T_{\in}^{\mathrm{chk}}(\lvert W\rvert)+m)$ or $O(T_{\perp}^{\mathrm{chk}}(\lvert W\rvert))$	$O(T_{\mathsf{up}}^{\mathrm{prf}}(\lvert W\rvert))$	$O(T_{\mathsf{up}}^{\mathrm{chk}}(\lvert W\rvert))$
Hash Tree [TT05]		$O(m+\log\lvert W\rvert)$	$O(m+\log\lvert W\rvert)$	$O(\log\lvert W\rvert)$	$O(\log\lvert W\rvert)$
Accumulators [PTT09]	Pairing	$O(m+1/\varepsilon)$	$O(m+1/\varepsilon)$	$O(\lvert W\rvert^{\varepsilon})$	$O(1/\varepsilon)$
	RSA	$O(m+\lvert W\rvert^{\varepsilon})$	$O(m+1/\varepsilon)$	$O(1/\varepsilon)$	$O(1/\varepsilon)$

number of results and constant update time.

The computational complexity of Search and Update queries in GSV is given in Table 2, as a general result (first row), and for two possible VHT instantiations, hash-based [TT05] or cryptographic accumulators-based [PTT09]. In the first case, we achieve logarithmic complexity in both accessing and updating (including the verification), and reach the lower bound of Section 3.3. The second case achieves optimal search time (for both the client and the server), linear in the number of results, as both access and verification of the VHT is done in constant time, but needs $O(|W|^{\varepsilon})$ update time, where $0 < \varepsilon < 1$ is a fixed constant (for the server only, the client having constant update time). The we could also use the VHT instantiation of [PTT09] which reverses the complexity for searches and update, and end up with an optimal update complexity.

Hence, depending on VHT implementations, the GSV construction achieves optimality in two senses: first in a general way, showing that the lower bound of Section 3 is tight, then for the Search query only, and finally for the Update query only, showing that we can have a VSSE scheme with no (asymptotic) verification overhead for either search or update queries. We emphasize that the accumulator-based instantiation might be interesting in practice for very large databases with a few updates, as in the case of $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$.[1]

A more practice-oriented evaluation is done in Appendix G.1. In particular, we estimate the computational overhead due to set hashing verification to a few milliseconds for a query matching 10 000 results.

We emphasize that both the search complexity and update complexity are better than one of Kurosawa and Ohtaki [KO13]: *e.g.* their search algorithm takes time $O(d)$ (d is the number of documents in the database).

6 Verification of SPS

SPS relies on a hierarchical structure to ensure forward security. As we think that this is a key security property, the modifications we make must not destroy forward privacy.

We can see SPS as levels of static SSE which are rebuilt upon modification of the database. Hence, we don't really need some dynamic verifiable data structures, static ones will suffice.

6.1 Remembering SPS

The SPS construction [SPS14] consists of $L+1$ levels $\mathbf{T}_0, \ldots, \mathbf{T}_L$, where $L = \lfloor \log N \rfloor$, and each level $\mathbf{T}_\ell$ is of size 2^ℓ. Each level can be seen as a map between keywords and lists of tuples of the form $(\mathsf{ind}, \mathsf{op}, cnt)$, where ind is a document index, $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ the operation associated with the tuple (addition or deletion)

[1] $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$ supports updates using a dynamic extension under the form of a dictionary stored by the server, but also as a counter table stored by the client. Both grow linearly with the number of distinct updated keywords, and thus do not support many updates.

and cnt a unique counter. These tuples are encoded in a way that their content is hidden unless given a token allowing decryption. This token will be unique per keyword w and per level ℓ, and with such a token, the server will not learn anything else than the content of the tuples associated with the keyword w at level ℓ. They are actually stored in a hash table, that allows a constant-time lookup, for a specific tuple (w, op, cnt), providing either the document ind or $\perp$ as output. This conceptual lookup operation is denoted $\Gamma_\ell[w, \mathsf{op}, cnt]$. Moreover, in a level ℓ, all the tuples $(w, \mathsf{ind}, \mathsf{op}, cnt)$ are lexicographically sorted based on key $(w, \mathsf{ind}, \mathsf{op})$.

For dynamism, the update operation exploits the leveled structure: for a new entry (addition or deletion), it is either inserted at level 0, in the unique cell of $\mathbf{T}_0$, or in the first empty level $\mathbf{T}_\ell$ with all the entries from the lower levels $\mathbf{T}_0, \ldots, \mathbf{T}_{\ell-1}$: the $2^\ell - 1$ entries in the lower levels and the new entry are indeed merged and re-ordered in $\mathbf{T}_\ell$ in the lexicographic ordering based on $(w, \mathsf{ind}, \mathsf{op})$, using an oblivious sort. Once the tuples are sorted, one can easily cancel addition-deletion for the same document/keyword pair, just replacing the two entries by $\perp$. The previous levels are all emptied.

To perform a search with keyword w, for each level, the server is provided the token for w, and performs lookups to find all matching entry in the levels. He does that for both add and del operations, simplifying inserted-then-deleted document/keyword pairs from the result set. However, such a data structure can lead to a linear-time search, in the worse case.

They thus improved their approach with an additional information in the tuples. The main drawback with the above tuples is the del operations that have to be looked for in a different level before returning a document that has been added and might have been deleted later. Also an extra information is added to del entries, such that, at level ℓ, given an add tuple with no matching del entry, the server can efficiently find the next add tuple with no matching del entry. This extra information is the field ℓ^*, called the *target level*, and the conceptual lookup operation now return $\Gamma_\ell[w, \mathsf{op}, cnt] = (\ell^*, \mathsf{ind})$ or $\perp$. In a deletion tuple, ℓ^* is the level where the associated addition tuple is stored. In an addition tuple, we just set $\ell^* = \ell$. If each level is now lexicographic ordered based on $(w, \ell^*, \mathsf{ind}, \mathsf{op})$, a fast procedure allows to find an addition tuple without deletion, leading to a complexity in $O(m \log^3 N)$. It can find and skip holes (a series of addition tuples that have been deleted later) in each level. Unfortunately, as is, these methods do not provide verifiable outputs to the client. Also note that SPS only supports add and del update operations, not edit^+ nor edit^-.

Stefanov *et al.* briefly described how to make there construction secure in the malicious model. However, although the modifications they propose are essential in the malicious model, they are not enough to guarantee security against active adversaries.

6.2 Quick Cryptanalysis of Ideas of [SPS14] in the Malicious Model

In [SPS14], the authors claim that adding a timestamp to the entries (as we did), MACing these, returning to the client the MACs with each non deleted entry and the holes proofs (*i.e.* the del entries at the edge of each hole component, together with their MAC) is enough. The client then would have to check the size of the holes and check that they match the distance between two add entries (as $\mathsf{CheckResults}$ does).

Incompletely Scanned Levels. One major modification brought by our protocol $\mathsf{Verif\text{-}SPS}$ is the verification that the search has been performed over all the entries of a specific level. The absence of such proofs (which is the role of non-member proofs in $\mathsf{Verif\text{-}SPS}$) allows trivial attacks where the server omits to return part of the results (*e.g.*, the last add entry of a level), resulting in missing results, or part of the holes (*e.g.* the last hole component of a level), resulting in non matching results.

We will see that the generation of such proofs in $\mathsf{Verif\text{-}SPS}$ is the main modification compared to the

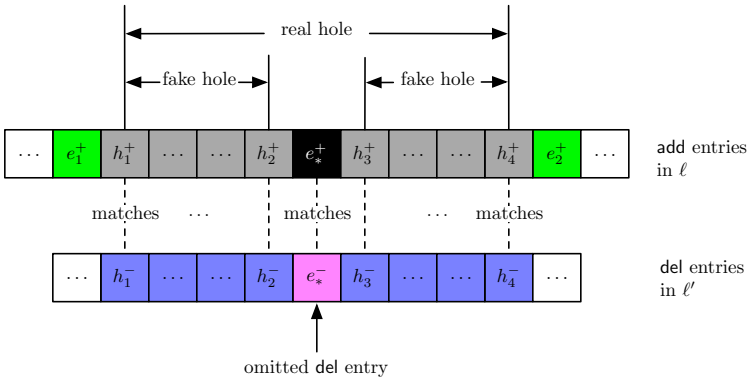

Figure 4 – Illustration of a non-consecutive hole attack. The adversary returns e_*^+ as a (fake) additional result, together with the proofs for non-maximal holes (h_1^-, h_2^-) and (h_3^-, h_4^-).

original SPS scheme, and that this modification implies the logarithmic overhead of the basic verifiable scheme.

Non-Consecutive Holes. Suppose now that the previous issue is (somehow) fixed. Unfortunately, there is still a possible attack in the claimed secure against active adversaries scheme of [SPS14].

In fact, checking the size of the holes is not enough to protect against malicious servers. Say there is, at level ℓ, an add entry e_*^+ in a hole between add entries e_1^+ and e_2^+, and that matches a del entry e_*^-. For the sake of simplicity, we suppose that the del entries matching the hole between e_1^+ and e_2^+ are all at the same level, between entries e_1^- and e_2^-. The server could decide to split the real hole in two, and make the client believe that e_*^+ is a non deleted matching add entry. He will, in some sort, omit the existence of e_*^-.

However, the size of the hole will still be equal to the distance between the returned add entries (e_1^+ and e_*^+ for the first hole, and e_*^+ and e_2^+ for the second one). In Verif-SPS, the attack is thwarted by a procedure CheckAdjacency. It verifies that every hole component is either the last of its level, or immediately followed by an other hole component, and hence that no del entry has been omitted by the server.

6.3 Basic Verifiable Construction

To avoid the server tampering with the data, one could MAC the tuples in the levels (together with a timestamp to avoid replay attacks). This would ensure that the server never returns to the client a tuple that was not in the level. Yet, it would not prevent the server from not returning a matching tuple (either for addition or deletion) for a search query.

Using verifiable hash tables and the particularities of the SPS construction, we show how we can thwart this attack, and more generally how to make SPS verifiable at the extra cost of a verifiable hash table per level.

To give a better insight into our verifiable SSE instantiation, we first give a basic construction, itself based on the basic construction of [SPS14]. The modified construction is described in Algorithms 3, 5, and 4. It uses a static verifiable hash table instantiation $\Theta = (\mathsf{VHTSetup}, \mathsf{VHTGet}, \mathsf{VHTVerify})$. Except for Search, the modifications are highlighted in red.

For all of the algorithms/protocols, except Search, the main modification resides in the addition of a verifiable hash table at every level and the use of authenticated encryption. For example, the Lookup function,

Algorithm 3 Modified construction from [SPS14].

Setup(DB)

1: Client chooses an encryption key esk, $L+1$ random level keys $k_0, \ldots, k_L$ (where $L = \log N$), and a key K_e for the PRF F. The scheme's key is $K = (esk, k_0, \ldots, k_L, K_e)$.
2: Client initializes an empty hierarchical structure D consisting of exponentially growing levels $\mathbf{T}_0, \ldots, \mathbf{T}_L$.
3: Initialize counters $r_0, \ldots, r_L$ to 0.
4: **return** $(K, (r_0, \ldots, r_L), (\mathbf{T}_0, \ldots, \mathbf{T}_L))$

Lookup(token, op, cnt)

1: hkey $\leftarrow H_{\mathsf{token}}(0||\mathsf{op}||cnt)$
2: $(v, \pi) \leftarrow \mathsf{VHTGet}(\mathbf{T}_\ell, \mathsf{VHT}_\ell, \mathsf{hkey})$
3: **if** $v = \perp$ **then return** $(\perp, \pi)$
4: **else**
5: Parse v as (c_1, c_2)
6: $(\ell^*, \widetilde{\mathsf{ind}}) \leftarrow c_1 \oplus H_{\mathsf{token}}(1||\mathsf{op}||cnt)$
7: **return** $(\ell^*, \widetilde{\mathsf{ind}}, v, \pi)$
8: **end if**

EncodeEntry$_{esk,\ell}(w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt)$

1: $\mathsf{token}_\ell \leftarrow F(k_\ell, h(w))$
2: $\mathsf{hkey} \leftarrow H_{\mathsf{token}_\ell}(0||\mathsf{op}||cnt)$
3: $c_1 \leftarrow (\ell^*, \mathsf{ind}) \oplus H_{\mathsf{token}_\ell}(1||\mathsf{op}||cnt)$
4: $c_2 \leftarrow \mathsf{AEnc}(esk, (\ell, r_\ell), (w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt))$
5: **return** $(\mathsf{hkey}, c_1, c_2)$

Update(K, op, ind, W; EDB)

1: **for all** $w \in W$ in random order **do**
2: Compute the target level ℓ^* of $(w, \mathsf{ind}, \mathsf{op})$
3: **if** $\mathbf{T}_0$ is empty **then**
4: Select a fresh key k_0, r_0++
5: $\mathbf{T}_0 \leftarrow \mathsf{EncodeEntry}_{esk,0}(w, \ell^*, \mathsf{ind}, \mathsf{op})$
6: **else**
7: Let $\mathbf{T}_\ell$ denote the first empty level.
8: r_ℓ++
9: Rebuild$(\ell, (w, \mathsf{ind}, \mathsf{op}))$.
10: **end if**
11: **end for**

in addition to retrieving the entry associated with the search token token, operation op, and counter cnt, will also return a proof that the returned value is correct.

The modifications brought to the Rebuild algorithm (Algorithm 4) are only a bit more complicated to analyze. First, the use of authenticated encryption instead of encryption without integrity verification ensures that the server only sends valid entries to the client. However, it does not prevent the server from sending the entries in the wrong order, or to send the same entry twice. To avoid such attacks, we need to ensure that the oblivious sort algorithm is secure, even in the presence of a malicious server. But we also have to make sure that the server cannot replay ciphertexts generated in the past to fool the client. So, we add to the additional data a per-level timestamp r_ℓ, incremented every time a level is rebuilt, as well as a flag, set to 0 or 1, to distinguish the ciphertexts produced before and after the first call to o-sort. Finally, the authenticating data structure for the verifiable hash table is initialized (line 28).

Oblivious Sorting with a Malicious Server. Usually, the problem of oblivious sorting is considered in the semi-honest setting, were the server only tries to infer informations from the messages he sees, but sticks to the protocol's execution.

A solution to enable resistance against active adversaries would be to rely on checkable memory techniques, but as we saw in Section 3.2, these are costly an would incur a logarithmic overhead. We could use less general verification techniques using the fact that, in the case of oblivious sorting, the client always knows which memory locations were modified at what time, as I/Os are independent from the sorted values. The client could use this property and add metadata – a timestamp and the memory location – to each memory block used by the algorithm. Every time a block is processed, the client will check that these data are consistent with the algorithm state. For example, when using sorting networks [GM11], we can MAC the location in the array and an integer accounting the last comparison executed on the item (in a sorting network, we can assign to each comparison a unique integer) to the probabilistic encryption of the item (used to ensure the obliviousness of the protocol). Every time an item is used by the algorithm, the client checks

Algorithm 4 Modified Rebuild protocol of the SPS construction.

Rebuild$(\ell, (w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt))$.

Let $\mathsf{entry}^* \leftarrow \mathsf{EncodeEntry}_{esk,K_e,k_0}(w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt)$
Let $\hat{\mathbf{B}} \leftarrow \{\mathsf{entry}^*\} \cup \mathbf{T}_0 \cup \ldots \mathbf{T}_{\ell-1}$.
for all $\mathsf{entry} = (\mathsf{hkey}, c_1, c_2) \in \hat{\mathbf{B}}$, with original level ℓ' **do**
  $(w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt) \hookleftarrow \mathsf{ADec}(esk, (\ell', r_{\ell'}), c_2)$
  Overwrite entry with $\mathsf{AEnc}(esk, (\ell, r_\ell, 0), (w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt))$
end for
$\hat{\mathbf{B}} \leftarrow \mathsf{o\text{-}sort}(\hat{\mathbf{B}})$, based on the lexicographic sorting key $(w, \ell^*, \mathsf{ind}, \mathsf{op})$.
for all $e = \mathsf{AEnc}(esk, (\ell, r_\ell, 0), (w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt))$ in $\hat{\mathbf{B}}$ in sorted order **do**
  if e marks the start of a new word w, for an operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ **then**
    Set $cnt_{\mathsf{op},w} \leftarrow 0$
    $e \leftarrow \mathsf{AEnc}(esk, (\ell, r_\ell, 1), (w, \ell^*, \mathsf{ind}, \mathsf{op}, 0))$
  else if e and its adjacent entry are add and del operations for the same $(w, \ell^*, \mathsf{ind})$ pair **then**
    Suppress the entries by updating both entries with $\mathsf{AEnc}(esk, (\ell, r_\ell, 1), \perp)$
  else Update e in $\hat{\mathbf{B}}$:
    $e \leftarrow \mathsf{AEnc}(esk, (\ell, r_\ell, 1), (w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt_{\mathsf{op},w}{}^{++}))$
  end if
end for
Randomly permute $\hat{\mathbf{B}} \leftarrow \mathsf{o\text{-}sort}(\hat{\mathbf{B}})$, based on hkey.
Select a new level key k_ℓ.
for all $\mathsf{entry} \in \hat{\mathbf{B}}$ **do**
  $(w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt) \hookleftarrow \mathsf{ADec}(esk, (\ell, r_\ell, 1), \mathsf{entry})$
  if $\mathsf{op} = \mathsf{add}$ **then**
    $\ell^* \leftarrow \ell$
  end if
  $(\mathsf{hkey}, c_1, c_2) \hookleftarrow \mathsf{EncodeEntry}_{esk,K_e,k_\ell}(w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt)$
  Add $(\mathsf{hkey}, c_1, c_2)$ to $\mathbf{T}_\ell$
end for
$(\mathsf{VHT}_\ell, \sigma_{\mathsf{VHT}_\ell}) \leftarrow \mathsf{VHTSetup}(\mathbf{T}_\ell)$

this MAC and ensures that it is not an element forged by the adversary. Using authenticated encryption with additional data, we could actually do both MAC and encryption at once, and the additional cost (compared to pure encryption) would be negligible using modern AEAD schemes [RBB03]. For now, we suppose that o-sort is secure against active adversaries.

Verifiable Search. The idea in the simple search algorithm is to make the server send to the client *all* the entries matching the searched keyword at every level ℓ, both for $\mathsf{op} = \mathsf{add}$ and $\mathsf{op} = \mathsf{del}$, make the client perform the eliminations among these entries.

To verify that the server sent all the entries corresponding to keyword w with operation op at level ℓ, we use the fact that the counters used to compute the entries' hkey are consecutive. So, if there are c add entries at level ℓ, for all $0 \leq i < c$, $\Gamma_\ell[w, \mathsf{add}, i] \neq \perp$, and $\Gamma_\ell[w, \mathsf{add}, c] = \perp$. Consecutiveness of the $cnt_{\mathsf{op},w}$ values, is ensured by the soundness of the o-sort algorithm.

Finally, as we check that the entries have the right value (and in particular that the last one is $\perp$), if the protocol does not abort, we are sure that $\widetilde{\mathcal{I}}_\ell^+$ contains all the (w, add) entries at level ℓ, and that $\widetilde{\mathcal{I}}_\ell^-$ contains all the (w, del) entries at level ℓ.

Time Complexity. The complexity of this construction is very similar to the one of the original SPS scheme. We can indeed show that, if the VHT is instantiated using the construction of Section 4.2, search has complexity $O(\alpha + \log^2 N)$ where α is the number of entries matching the searched keyword in the database, and update has $O(\log^2 N)$ amortized, $O(N \log N)$ worst-case update time. The full computational complexity accounting is done in Appendix F.

Algorithm 5 Simple Search algorithm. Every time that the result of a client's computation is REJECT, he immediately stops and returns REJECT.

Search$(K, w, \sigma;$ EDB$)$

Client: On input (K, w), it computes tokens for each level.
$\mathsf{tks} \leftarrow \{\mathsf{token}_\ell = F(k_\ell, h(w)), \quad \ell = 0, \ldots, L\}$
The client sends tks to the server.
Server:
for $\ell = L$ **to** 0 **do**
  Initialize $\widetilde{\mathcal{I}}_\ell^+$ and $\widetilde{\mathcal{I}}_\ell^-$ to empty lists.
  $cnt \leftarrow 0$
  repeat
    $(\ell^*, \mathsf{ind}, entry, \pi) \leftarrow \mathsf{Lookup}(\mathsf{token}_\ell, \mathsf{add}, cnt{+}{+})$
    $\widetilde{\mathcal{I}}_\ell^+[cnt] \leftarrow (entry, \pi)$
  until $entry = \perp$
  $cnt \leftarrow 0$
  repeat
    $(\ell^*, \mathsf{ind}, entry, \pi) \leftarrow \mathsf{Lookup}(\mathsf{token}_\ell, \mathsf{del}, cnt{+}{+})$
    $\widetilde{\mathcal{I}}_\ell^-[cnt] \leftarrow (entry, \pi)$
  until $entry = \perp$
  Send $\{(\widetilde{\mathcal{I}}_\ell^+, \widetilde{\mathcal{I}}_\ell^-)\}_\ell$ to the client.
end for
Client: Let $\mathcal{I} \leftarrow \emptyset$
for $\ell = L$ **to** 0 **do**
  for $i = 0$ **to** $|\widetilde{\mathcal{I}}_\ell^+| - 2$ **do** ▷ *Check that all the elements (but the last) of the list are valid elements*
    Parse $\widetilde{\mathcal{I}}_\ell^+[i]$ as $(entry, \pi)$
    $\mathsf{hkey} \leftarrow H_{\mathsf{token}_\ell}(0||\mathsf{add}||i)$
    if $\mathsf{VHTVerify}(\sigma_{\mathsf{VHT}_\ell}, \mathsf{hkey}, entry, \pi) = \mathtt{REJECT}$ or $entry = \perp$ **then**
      return REJECT
    else
      $(\ell^*, \mathsf{ind}) \leftarrow entry.c_1 \oplus H_{\mathsf{token}_\ell}(1||\mathsf{add}||i)$
      $\mathcal{I} \leftarrow \mathcal{I} \cup \{\mathsf{ind}\}$
    end if
  end for
  Parse $\widetilde{\mathcal{I}}_\ell^+[|\widetilde{\mathcal{I}}_\ell^+| - 1]$ as $(entry, \pi)$ ▷ *Check that the last element of the list is* $\perp$
  $\mathsf{hkey} \leftarrow H_{\mathsf{token}_\ell}(0||\mathsf{add}||(|\widetilde{\mathcal{I}}_\ell^+| - 1))$
  $\mathsf{VHTVerify}(\sigma_{\mathsf{VHT}_\ell}, \mathsf{hkey}, \perp, \pi)$
  Do the same as before, but with del instead of add, running the loop over $\widetilde{\mathcal{I}}_\ell^-$ instead of $\widetilde{\mathcal{I}}_\ell^+$, and removing elements to $\mathcal{I}$ instead of adding them.
end for
return $\mathcal{I}$

6.4 Verifiable Sublinear Construction

In [SPS14], the authors explain that to make their sublinear algorithm secure against malicious adversaries, one must give a proof for each returned add entry in the form of a MAC, and a proof for each hole as the del entries at the edge of each deletion region, together with a MAC for each of these entries.

Unfortunately, the verification algorithm is not further detailed and, as we have seen in Section 6.2, providing these proof elements is not enough: there are still lots of way for the server to cheat. One of these is by just not returning the add entries for a level ℓ. Or just return part of them, as in the simple search algorithm. Similarly, the server could also omit to return one or more holes with a specific target level.

As in the simple search algorithm, the main threat the client has to protect against is not receiving all the elements (either the add entries or the holes) necessary to compute the search result. Again, we heavily rely on the structure of the levels and on how the entries' counter increases. For example, for each level ℓ, the server will give a proof that for a counter cnt_{max}, the entry $\Gamma_\ell[w, \mathsf{add}, cnt_{\mathsf{max}}]$ is $\perp$, ensuring that there is no add entry matching w with counter $cnt > cnt_{\mathsf{max}}$. We proceed similarly for del entries.

The verification of the sublinear Search algorithm is not as direct as the verification of basic Search, and we will present all the procedures needed for the generation of proofs and their verification. For the sake of readability, in the algorithms, as before, we suppose that once a call returned REJECT, the calling procedure immediately stops and returns REJECT itself. We also omit the parameters K, w, σ and EDB in the procedure signatures.

Algorithm 6 Sublinear Search algorithm.

$\mathsf{Search}(K, w, \sigma; \mathsf{EDB})$

1: *Client*: On input (K, w), it computes tokens for each level.
2: $\mathsf{tks} \leftarrow \{\mathsf{token}_\ell = F(k_\ell, h(w)), \ell = 0, \ldots, L\}$
3: The client sends tks to the server.
4: *Server*:
5: Initialize AllHoles as an empty hash table of lists.
6: **for** $\ell = 0$ **to** L **do**
7: $\quad \mathcal{I}_\ell \leftarrow \mathsf{ProcessLevel}(\ell, \mathsf{token}_\ell)$
8: **end for**
9: $\Pi_{\mathsf{Holes}} \leftarrow \mathsf{ProveHoles}(\mathsf{AllHoles})$
10: Send $(\mathcal{I}_0, \ldots, \mathcal{I}_L, \Pi_{\mathsf{Holes}})$ to the client.
11: *Client*:
12: $\mathsf{AllHoles} \leftarrow \mathsf{VerifyHoles}(\Pi_{\mathsf{Holes}})$
13: $\mathcal{I} \leftarrow \mathsf{CheckResults}(\mathcal{I}_0, \ldots, \mathcal{I}_L, \mathsf{AllHoles})$
14: **return** $\mathcal{I}$

6.4.1 Verification of the Sublinear Search Algorithm.

The Search algorithm, as described by Algorithm 6, calls four sub-procedures:

- ProcessLevel, as in the original SPS scheme, finds all the tuples $(w, \ell, \mathsf{ind}, \mathsf{add})$ stored at level ℓ such that there is no corresponding entry $(w, \ell, \mathsf{ind}, \mathsf{del})$ in lower levels $\ell' < \ell$. To do so, it has to find the holes with target level ℓ, so, for verification purposes, it also returns these holes and adds them to the hash table AllHoles that tracks all the holes ever found.
- ProveHoles takes as input all the holes found by the calls to ProcessLevel and returns a proof for these, *i.e.* a proof that they are *maximal* and *consecutive* (we will explain these notions later).
- VerifyHoles verifies the proofs produced by ProveHoles and returns the corresponding holes.
- CheckResults checks the consistency of the results returned by ProcessLevel and produces the result set $\mathcal{I}$.

The ProcessLevel **Algorithm.** ProcessLevel is the key procedure described in Algorithm 7. It goes through the add entries for keyword w at level ℓ and looks for those that were not deleted by a subsequent deletion. If such a deletion happened, a corresponding $(w, \ell, \mathsf{ind}, \mathsf{del})$ entry will be in some lower level $\ell' < \ell$ (note that such an entry cannot be stored at level ℓ because it would have been "simplified" by the Rebuild algorithm). If this search for deletions were done on an entry-by-entry basis, it would be very inefficient, and the server would need to go through all the add entries at level ℓ and the search complexity would not be better than in the simple Search protocol.

To avoid this, if ProcessLevel found an entry that was deleted in the lower levels, it will directly jump to the next add entry not deleted. This is where SkipHole enters: it finds the largest collection of successive add entries at level ℓ (starting at the current entry) for which we can find a matching collection of del entries in lower levels (line 1). These entries are called a *hole*.

For this, it uses DeletedSum. DeletedSum finds in every level $\ell' < \ell$ the largest region of successive del entries whose target level is ℓ and document's index comprised between ind and ind$'$. It returns the sum of the size of these regions and their limits *i.e.* the smallest (resp. biggest) counter cnt_x (resp. cnt_y) such that $\Gamma_{\ell'}[w, \mathsf{del}, cnt_x] = (\ell, \mathsf{ind}_x)$ and $\mathsf{ind}_x \geq \mathsf{ind}$ (resp. $\Gamma_{\ell'}[w, \mathsf{del}, cnt_y] = (\ell, \mathsf{ind}_y)$ and $\mathsf{ind}_y \leq \mathsf{ind}'$). We refer to these limits as *hole components*.

Algorithm 7 The ProcessLevel algorithm (and auxiliary procedures).

ProcessLevel$(\ell, \mathsf{token}_\ell)$

$cnt \leftarrow 0$
Initialize $\mathcal{I}_\ell$ as an empty list.
$(\ell, \mathsf{ind}, entry, \pi)$
  $\leftarrow$ Lookup$(\mathsf{token}_\ell, \mathsf{add}, cnt{+}{+})$
while $entry \neq \perp$ **do**
  if $(w, \ell, \mathsf{ind}, \mathsf{del})$ is not found in any lower levels **then** ▷ *Through a binary search for each lower level*
    Append $(entry, \pi)$ to $\mathcal{I}_\ell$
  else
    Call $(cnt, \mathsf{Hole}) \leftarrow$ SkipHole$(\ell, \mathsf{token}_\ell, \mathsf{ind})$
    Append Hole to $\mathcal{I}_\ell$
  end if
  $(\ell, \mathsf{ind}, entry, \pi)$
    $\leftarrow$ Lookup$(\mathsf{token}_\ell, \mathsf{add}, cnt{+}{+})$
end while
Append $(entry = \perp, \pi)$ to $\mathcal{I}_\ell$
  ▷ *Show that we reached the last* add *element in level* ℓ.
return $\mathcal{I}_\ell$

AppendHole(AllHoles, Hole)

for all $(\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y) \in \mathsf{Hole}$ **do**
  Append (ℓ, cnt_x, cnt_y) to AllHoles$[\ell']$
end for

SkipHole$(\ell, \mathsf{token}_\ell, \mathsf{ind})$

Through binary search, compute the maximum identifier $\mathsf{ind}' \geq \mathsf{ind}$ in level ℓ s.t. $\mathsf{count}_{\ell,\ell,w,\mathsf{add}}(\mathsf{ind}', \mathsf{ind}) + 1 = sum$, with $(sum, \mathsf{Hole}) \leftarrow$ DelSum$(\ell, \mathsf{ind}, \mathsf{ind}')$
AppendHole(AllHoles, Hole)
  ▷ *Track all the generated holes.*
return the corresponding value cnt for ind' and Hole.

DeletedSum$(\ell, \mathsf{ind}, \mathsf{ind}')$

$sum \leftarrow 0$, Hole $\leftarrow$ empty list
for $\ell' = 0$ **to** $\ell - 1$ **do**
  Find the largest region $[(\ell, \mathsf{ind}_x), (\ell, \mathsf{ind}_y)]$ that falls within the range $[(\ell, \mathsf{ind}), (\ell, \mathsf{ind}')]$
  if such a region is found **then**
    Let cnt_x and cnt_y be such that $(\ell, \mathsf{ind}_x) = \Gamma_\ell[w, \mathsf{del}, cnt_x]$ and $(\ell, \mathsf{ind}_y) = \Gamma_\ell[w, \mathsf{del}, cnt_y]$
    $sum \leftarrow sum + cnt_y - cnt_x + 1$
    Append $(\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$ to Hole
  end if
end for
return (sum, Hole).

Finally, for the sake of verifiability, we need to keep track of these holes. So we add them to a hash table of lists AllHoles. AllHoles$[\ell']$ stores all the hole components at level ℓ'. Note that, as we process levels increasingly, the target level ℓ of components added to AllHoles$[\ell']$ increases similarly, and as among levels, entries are processed with increasing counter, counters cnt_x and cnt_y also increase. Actually, if $(\ell^1, cnt_x^1, cnt_y^1)$ and $(\ell^2, cnt_x^2, cnt_y^2)$ are two successive entries in AllHoles$[\ell']$, $\ell^1 \leq \ell^2$ and $cnt_x^1 \leq cnt_y^1 < cnt_x^2 \leq cnt_y^2$. Hence, the server does not have to go through sorting to get a sorted list (which is important for computational complexity).

In the end, ProcessLevel returns the lists $\mathcal{I}_\ell$ whose elements are either add entries for w in level ℓ (together with some membership proof) or holes for w with target level ℓ. The last element of this list is always the $\perp$ entry with a VHT proof corresponding to the last counter value of the main loop (the smallest cnt for which Lookup returns $\perp$).

Proving and Verifying Holes. The server has to prove to the client that he did not cheat when producing the holes. To do so, we could first use the verifiable hash table to show that the hole components limits are genuine del entries, which is what ProveHoles does. Unfortunately, this is not enough: for example the server could return two holes for which the intersection of components at a level ℓ is not empty, which never happens if the search protocol is run fairly. He could also intentionally omit a del entry. Fortunately, we can use some very interesting properties of the holes to avoid any falsification of the results by the server, which

Algorithm 8 Prove and Verify Holes.

ProveHoles(AllHoles)

1: Initialize Π_{Holes} as an empty table of lists.
2: **for** $\ell = 0$ **to** L **do**
3: $cnt_{last} \leftarrow -1$
4: **for all** $(\ell', cnt_x, cnt_y, \cdot, \cdot)$ in AllHoles$[\ell]$ in increasing order **do**
5: $(entry_x, \pi_x)$ $\leftarrow$ Lookup$(\mathsf{token}_\ell, \mathsf{del}, cnt_x)$
6: $(entry_y, \pi_y)$ $\leftarrow$ Lookup$(\mathsf{token}_\ell, \mathsf{del}, cnt_y)$
7: Append $((cnt_x, entry_x, \pi_x), (cnt_y, entry_y, \pi_y))$ (in that order) to $\Pi_{\mathsf{Holes}}[\ell]$
8: $cnt_{last} \leftarrow cnt_y$
9: **end for**
10: $(entry_{last}, \pi_{last})$ $\leftarrow$ Lookup$(\mathsf{token}_\ell, \mathsf{del}, cnt_{last} + 1)$ ▷ $entry_{last} = \perp$
11: Append π_{last} to $\Pi_{\mathsf{Holes}}[\ell]$
12: **end for**
13: **return** Π_{Holes}

CheckAdjacency(AllHoles$[\ell]$)

1: $cnt \leftarrow -1, \ell^* \leftarrow 0$
2: **for all** $(\ell', cnt_x, cnt_y, ., .) \in$ AllHoles$[\ell]$ in increasing order **do**
3: **if** $\ell^* > \ell'$, $cnt_x \neq cnt + 1$ or $cnt_x > cnt_y$
4: **return** REJECT
5: $cnt \leftarrow cnt_y, \ell^* \leftarrow \ell'$
6: **end for**
7: **return** ACCEPT

VerifyHoles(Π_{Holes})

1: Initialize AllHoles as an empty table of lists.
2: **for** $\ell = 0$ **to** L **do**
3: $cnt_{last} \leftarrow -1$
4: **for all** $((cnt_x, entry_x, \pi_x), (cnt_y, entry_y, \pi_y)) \in \Pi_{\mathsf{Holes}}[\ell]$ **do**
5: $\mathsf{hkey}_x \leftarrow H_{\mathsf{token}_\ell}(0||\mathsf{del}||cnt_x)$
6: $\mathsf{hkey}_y \leftarrow H_{\mathsf{token}_\ell}(0||\mathsf{del}||cnt_y)$
7: VHTVerify$(\sigma_{\mathsf{VHT}_\ell}, \mathsf{hkey}_x, entry_x, \pi_x)$
8: VHTVerify$(\sigma_{\mathsf{VHT}_\ell}, \mathsf{hkey}_y, entry_y, \pi_y)$
9: Parse $entry_x$ as $(l_x^*, \mathsf{ind}_x, c_{2x})$
10: Parse $entry_y$ as $(l_y^*, \mathsf{ind}_y, c_{2y})$
11: **if** $l_x^* \neq l_y^*$ **return** REJECT
12: Append $(l_x^*, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$ to AllHoles$[\ell]$
13: $cnt_{last} \leftarrow cnt_y$
14: **end for**
15: hkey_{last} $\leftarrow H_{\mathsf{token}_\ell}(0||\mathsf{del}||cnt_{last} + 1)$
16: VHTVerify$(\sigma_{\mathsf{VHT}_\ell}, \mathsf{hkey}_{last}, \perp, \pi_{last})$
17: CheckAdjacency(AllHoles$[\ell]$)
18: **end for**
19: **return** AllHoles

we use in ProveHoles and VerifyHoles (Algorithm 8).

First, let us fix a level ℓ. We will consider only hole components or del entries whose level is ℓ. For such a del entry, there exists an add entry in a higher level ℓ'. Hence it should belong to a hole component with target level ℓ'. Said otherwise, every del entries belong to a hole, and if the client wants to check that the server rightly took into account every del entries, he should check that the hole components at level ℓ span all its del entries. This is exactly what CheckAdjacency does, using the fact that hole components are *consecutive*.

In the previous paragraph, we noticed that the elements in AllHoles$[\ell]$ have components increasing in a very particular way. Let $(\ell^1, cnt_x^1, cnt_y^1)$ be an element of AllHoles$[\ell]$ The del entry with counter value cnt_y^1+1 must be either $\perp$, or a del entry with target level $\ell^* \geq \ell^1$ (because of the sorting key used in Rebuild). In the latter case, it necessarily is the start of a new hole component at level ℓ. Hence, if $(\ell^1, cnt_x^1, cnt_y^1)$ and $(\ell^2, cnt_x^2, cnt_y^2)$ are two successive entries in AllHoles$[\ell]$, we must have $\ell^1 \leq \ell^2$ and $cnt_x^1 \leq cnt_y^1 = (cnt_x^2 - 1) < cnt_y^2$. This condition is checked at line 3 of CheckAdjacency. Testing that the server reached the last del entries of level ℓ is done by verifying a non membership proof for counter $cnt_{last} + 1$ at line 16, where cnt_{last} is the highest counter value encountered among the hole components of level ℓ. The proof was produced by the server at line 10.

Algorithm 9 CheckResults algorithm.

CheckResults($\mathcal{I}_0, \ldots, \mathcal{I}_L$, AllHoles)

1: $\mathcal{I} \leftarrow \emptyset$
2: $\mathsf{ind}_{last} \leftarrow -1$
3: $cnt \leftarrow 0$
4: **for** $\ell = 0$ **to** L **do**
5: **for all** $e \in \mathcal{I}_\ell$ **do**
6: **if** e is $(entry, \pi)$ **then**
7: $\mathsf{hkey}_x \leftarrow H_{\mathsf{token}_\ell}(0||\mathsf{add}||cnt)$
8: VHTVerify$(\sigma_{\mathsf{VHT}_\ell}, \mathsf{hkey}, entry, \pi)$
9: $(w, \ell, \mathsf{ind}, \mathsf{add}, cnt)$
$\leftarrow$ ADec$(esk, (\ell, r_\ell), c_2)$
▷ *As entry has been verified, we know that* ADec *returns keyword* w*, target level* ℓ *and operation* add.
10: **if** $\mathsf{ind}_{last} \geq \mathsf{ind}$ **return** REJECT
11: $\mathsf{ind}_{last} \leftarrow \mathsf{ind}$
12: $cnt{+}{+}$
13: $\mathcal{I} \leftarrow \mathcal{I} \cup \{\mathsf{ind}\}$
14: **else if** e is a hole Hole **then**
15: **if** previous entry was a hole
16: **return** REJECT
17: $sum \leftarrow 0$
18: $\mathsf{ind}_{\mathsf{max}} \leftarrow \mathsf{ind}_{last}$
19: **for all** $h \in$ Hole **do**
20: Parse h as $(\ell', \ell^*, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$
21: **if** $\ell^* \neq \ell$, $h \notin$ AllHoles$[\ell']$
or $\mathsf{ind}_x \leq \mathsf{ind}_{last}$
22: **return** REJECT
23: $\mathsf{ind}_{\mathsf{max}} \leftarrow \mathsf{max}(\mathsf{ind}_{\mathsf{max}}, \mathsf{ind}_y)$
24: $sum \leftarrow sum + cnt_y - cnt_x + 1$
25: **end for**
26: $\mathsf{ind}_{last} \leftarrow \mathsf{ind}_{\mathsf{max}}$
27: $cnt \leftarrow cnt + sum + 1$
28: **else** ▷ e *is the proof* π_{last}
29: $\mathsf{hkey}_{last} \leftarrow H_{\mathsf{token}_\ell}(0||\mathsf{add}||cnt)$
30: VHTVerify$(\sigma_{\mathsf{VHT}_\ell}, \mathsf{hkey}, \perp, \pi_{last})$
31: **if** e is not the last element of $\mathcal{I}_\ell$
32: **return** REJECT
33: **end if** ▷ *Test value of* e
34: **end for** ▷ *Loop over* $\mathcal{I}_\ell$
35: **end for** ▷ *Loop over the levels*
36: **return** $\mathcal{I}$

The CheckResults **Algorithm.** The last thing the client has to do is to check the consistency of the result lists $\mathcal{I}_\ell$ and holes. This is what CheckResults achieves, as described in Algorithm 9. It verifies three points: first that every add entry in $\mathcal{I}_\ell$ has no matching del entry among the holes, then that the holes in $\mathcal{I}_\ell$ are verified holes, and finally that all the add entries at level ℓ where considered when searching.

First, one must notice that, if the Search protocol is fairly executed by the server, one cannot find in $\mathcal{I}_\ell$ two consecutive holes. Otherwise, it would say that the first hole is not maximal, unlike what SkipHole ensures. Hence, in CheckResults (line 16), we check that there are not two consecutive holes entries in $\mathcal{I}_\ell$. In the following, we will suppose that a hole is immediately followed by an add entry in lists $\mathcal{I}_\ell$.

One of the key thing to check is that the document's index of successively considered entries strictly increases: in ProcessLevel, the server adds to $\mathcal{I}_\ell$ either add entries retrieved using an incrementing counter or holes whose components limits indices must be larger than the previous add entry and smaller than the next one, by construction of SkipHole. This condition is checked at lines 10 and 21.

CheckResults would also reject when holes are incorrect, *i.e.* when their target level is not ℓ, or if they were not registered in AllHoles. Note that if a hole h is in AllHoles, it has been proved correct by a previous ProveHole call, and we do not have to show that its components were correctly formed and that their limits were genuine entries.

The algorithm also checks that the add entries are correct. In particular, the counter value used to retrieve the hkey is not provided by the server, but recomputed by the client: it is incremented when the current element e is an add entry, and, if e is a hole, increased by its width. By doing so, and relying on the increasing document indices, we ensure that the holes width is correct, *i.e.* that the width of a hole is equal to the difference of counter values of neighboring add entries. Figure 5 illustrates this point.

Merging the VHT in the Level Table and De-Amortization. If one uses Section 4.2 as the VHT instantiation Θ, one would notice that the contents of the hash tables, *i.e.* the entries created by EncodeEntry, are authenticated ciphertexts (at least for the member c_2) – essentially a ciphertext associated with a MAC – which are then MACed with their position in the table. We could avoid this second MAC by adding the position in the table to the associated data of the authenticated encryption, and including this position as a member of the entry. All the algorithms described previously would not change, except that the VHTVerify calls would be replaced by the verification that the decryption procedure ADec did not return `REJECT` when verifying a membership proof.

Moreover, this modification (including the position index in the associated data) would essentially be transparent to the oblivious sort algorithm. Hence, we could apply the de-amortization as in [SPS14], without caring about any extra cost induced by the verified hash table. Using de-amortization, the average case complexity becomes the worst-case complexity: the update operation takes $O(\log^2 N)$ time and induces $O(\log N)$ bandwidth, in the worst case.

If we were to use an other instantiation for Θ, we would have to make sure we can use de-amortization on this instantiation to avoid unacceptable worst-case complexity.

Time Complexity. As we reuse the same Rebuild algorithm as the one of the simple construction, we refer to Section 6.3 and claim that Update take $O(\log^2 N)$ amortized time, and $O(N \log N)$ in the worst case.

For the Search algorithm, we recall that the original complexity of the unverified search was $O(m \log^3 N)$ where m is the number of matching documents. In our case, verification does not increase the search time, and the complexity of Search is still $O(m \log^3 N)$, as shown in Appendix F.

An estimation of the practical complexity is done in Appendix G.2. In particular, we claim that the verification of a query's result on the client side should take a few tens of milliseconds, while the overhead for the server is negligible.

6.4.2 Security of Verif-SPS

Despite the modifications brought to SPS for verifiability, we claim that this new verifiable dynamic SSE scheme Verif-SPS is correct, sound and $\mathcal{L}_{\mathsf{SPS}}$-adaptively-secure ($\mathcal{L}_{\mathsf{SPS}}$ is the leakage function of the original SPS scheme).

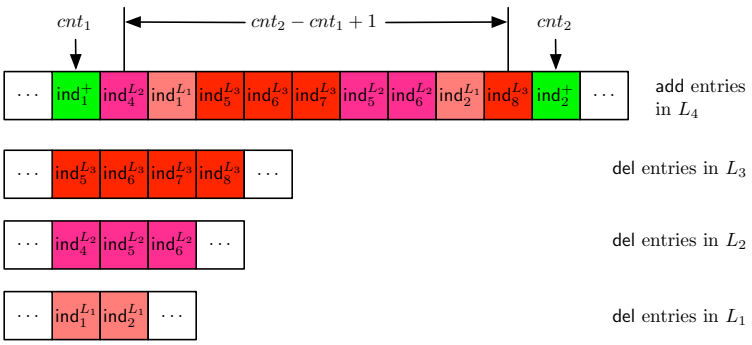

Figure 5 – Example of what CheckResults checks. In this example, $\mathcal{I}_{L_4}$ contains the following sequence: add entry with counter cnt_1 and index ind_1^+, a hole whose components are at levels L_3, L_2 and L_1, and an other add entry with counter cnt_2 and index ind_2^+. The upper row represents the add entries for keyword w at level L_4. In red, pink and purple are the ones with a matching entry in the hole. The green entries are the add entries in $\mathcal{I}_{L_4}$

Correctness follows from the correctness of SPS and from the completeness of the verifiable hash table Θ. Using an hybrid argument, we can show that for every adversary A there exist two adversaries B and B' such that

$$\mathbf{AdvCor}_{A}^{\mathsf{SSE},\mathsf{Verif\text{-}SPS}}(\lambda) \leq \mathbf{AdvCor}_{B}^{\mathsf{SSE},\mathsf{SPS}}(\lambda) + \mathbf{AdvComp}_{B'}^{\mathsf{VHT},\Theta}(\lambda)$$

where B makes the same requests as A to the SSECORR game. In the case of our instantiation of Θ, this gives us $\mathbf{AdvCor}_{A}^{\mathsf{SSE},\mathsf{Verif\text{-}SPS}}(\lambda) \leq \frac{N^2}{2^{\lambda+2}}$ (we might have collision issues with the hash function computing hkey in $\mathsf{EncodeEntry}$, which is modeled as a random oracle).

The soundness relies both on the verifiable hash table and on the authenticated encryption scheme, it is easy to see that using an hybrid argument that will successively suppose the complete soundness of the VHT, and unforgeability of the authenticated encryption scheme, we will be able to prove the soundness of Verif-SPS. Yet, in the proof, we will also have to suppose that SPS is also fully correct. More formally, we have the following theorem, proven in Appendix E.

Theorem 7 (Soundness of Verif-SPS)**.** *For every adversary A, there are adversaries B, C and D such that*

$$\begin{aligned}\mathbb{P}[&\mathrm{SSESOUND}_{A}^{\mathsf{Verif\text{-}SPS}}(\lambda) = 1] \\ &\leq N \cdot \mathbf{AdvSnd}_{B}^{\mathsf{VHT},\Theta}(\lambda) + \mathbf{Adv}_{C}^{\mathsf{Auth},\mathsf{AEnc}}(\lambda) + \mathbf{AdvCor}_{D}^{\mathsf{SSE},\mathsf{SPS}}(\lambda)\end{aligned}$$

Finally, confidentiality directly follows from the confidentiality of SPS. More precisely, as in the protocols (both Search and Update) the tokens/messages sent by the client to the server are (almost) the same as in SPS (we replaced encryption by authenticated encryption with associated data that was already freely available to the adversary) and that the verifiable hash tables only authenticate encrypted entries already present in the levels, we directly have that for every adversary A, simulator S and leakage function $\mathcal{L}$, there exists a simulator S' such that

$$\mathbf{AdvConf}_{A,S',\mathcal{L}}^{\mathsf{SSE},\mathsf{Verif\text{-}SPS}}(\lambda) = \mathbf{AdvConf}_{A,S',\mathcal{L}}^{\mathsf{SSE},\mathsf{SPS}}(\lambda).$$

In particular, as SPS is $\mathcal{L}_{\mathsf{SPS}}$-adaptively-secure, Verif-SPS is also $\mathcal{L}_{\mathsf{SPS}}$-adaptively-secure. We recall that $\mathcal{L}_{\mathsf{SPS}}(\mathsf{DB}) = N$ for the initialization, $\mathcal{L}_{\mathsf{SPS}}(\mathsf{DB}, q) = (\mathsf{DB}(w), \overline{w})$ ($\overline{w}$ is the query pattern) for a search query, and $\mathcal{L}_{\mathsf{SPS}}(\mathsf{DB}, \mathsf{op}, \mathsf{in}) = (\mathsf{op}, \mathsf{ind}, n)$ where n is the number of modified keywords and ind the identifier of the updated document. In particular, an update never reveals that the document matches a previously searched keyword.

Conclusion

In this paper, we presented the first constructions of verifiable symmetric searchable encryption. We showed that, even though such constructions have to respect some lower bounds (and that these lower bounds are – almost – tight), we can build very efficient searchable encryption schemes secure against malicious servers. In particular, we explained how one can make the construction of [SPS14] verifiable at no additional cost for the server, and using only a small amount of computation on the client side.

These efficient and secure constructions open some new questions on SSE and verifiability, in particular about functionalities SSE schemes are likely to achieve – as expressive queries, or a three-party setting where the data owner and the client are separated – while ensuring results authenticity.

Acknowledgements

We thank Olya Ohrimenko for her helpful suggestions and remarks, and Mehdi Tibouchi for sharing early results on multiset hashing. We also thank anonymous reviewers for their useful comments.

References

[BEG+91] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *32nd FOCS*, pages 90–99. IEEE Computer Society Press, October 1991.

[BLT14] Andrey Bogdanov, Martin M. Lauridsen, and Elmar Tischhauser. AES-based authenticated encryption modes in parallel high-performance software. Cryptology ePrint Archive, Report 2014/186, 2014. `http://eprint.iacr.org/2014/186`.

[BM97] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 163–192. Springer, Heidelberg, May 1997.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006.

[CDVD+03] Dwaine Clarke, Srinivas Devadas, Marten Van Dijk, Blaise Gassend, and G Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In Chi-Sung Laih, editor, *ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 188–207. Springer, Heidelberg, November / December 2003.

[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 06*, pages 79–88. ACM Press, October / November 2006.

[CGPR15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of ACM Computer and Communications Security*. ACM, 2015.

[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.

[CJJ+14] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*. The Internet Society, February 2014.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, August 2002.

[CT14] David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 351–368. Springer, Heidelberg, May 2014.

[DNRV09] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 503–520. Springer, Heidelberg, March 2009.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part II*, volume 6756 of *LNCS*, pages 576–587. Springer, Heidelberg, July 2011.

[GMOT12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In Yuval Rabani, editor, *23rd SODA*, pages 157–167. ACM-SIAM, January 2012.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[Gol04] Oded Goldreich. *Foundations of cryptography.* Cambridge University Press, 2004.

[KO12] Kaoru Kurosawa and Yasuhiro Ohtaki. UC-secure searchable symmetric encryption. In Angelos D. Keromytis, editor, *FC 2012*, volume 7397 of *LNCS*, pages 285–298. Springer, Heidelberg, February / March 2012.

[KO13] Kaoru Kurosawa and Yasuhiro Ohtaki. How to update documents verifiably in searchable symmetric encryption. In Michel Abdalla, Cristina Nita-Rotaru, and Ricardo Dahab, editors, *CANS 13*, volume 8257 of *LNCS*, pages 309–328. Springer, Heidelberg, November 2013.

[KPR12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 965–976. ACM Press, October 2012.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 719–734. Springer, Heidelberg, May 2013.

[NR05] Moni Naor and Guy N. Rothblum. The complexity of online memory checking. In *46th FOCS*, pages 573–584. IEEE Computer Society Press, October 2005.

[PKV+14] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steve Bellovin. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374. IEEE Computer Society Press, May 2014.

[PT08] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *ICICS 07*, volume 4861 of *LNCS*, pages 1–15. Springer, Heidelberg, December 2008.

[PTT09] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Cryptographic accumulators for authenticated hash tables. Cryptology ePrint Archive, Report 2009/625, 2009. `http://eprint.iacr.org/2009/625`.

[RBB03] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):365–403, 2003.

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS 2014*. The Internet Society, February 2014.

[STSY01] Tomas Sander, Amnon Ta-Shma, and Moti Yung. Blind, auditable membership proofs. In Yair Frankel, editor, *FC 2000*, volume 1962 of *LNCS*, pages 53–71. Springer, Heidelberg, February 2001.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 299–310. ACM Press, November 2013.

[SWP00] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55. IEEE Computer Society Press, May 2000.

[Tib15] Mehdi Tibouchi. Personal communications, 2015.

[TT05] Roberto Tamassia and Nikos Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP 2005*, volume 3580 of *LNCS*, pages 153–165. Springer, Heidelberg, July 2005.

[WHC+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 191–202. ACM Press, November 2014.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 215–226. ACM Press, November 2014.

Appendix

A Chosen Message Attack Security Game

As stated in Section 2.1.3, the Chosen Message Attack (CMA) game formalized in Figure 6 defines the security of a Message Authentication Code: any polynomial adversary A should not win the game with non-negligible advantage $\mathbf{Adv}_A^{\mathsf{MAC}}(\lambda)$.

Definition 5 (CMA security)**.** *Let* $(\mathsf{MAC}, \mathsf{Verif})$ *be a message authentication code, A probabilistic polynomial-time algorithm. We define the advantage of A in the CMA security game as* $\mathbf{Adv}_A^{\mathsf{MAC}}(\lambda)$ *where*

$$\mathbf{Adv}_A^{\mathsf{MAC}}(\lambda) = \mathbb{P}[\mathrm{CMA}_A^{\mathsf{MAC}}(\lambda) = 1].$$

We say that $(\mathsf{MAC}, \mathsf{Verif})$ *is* unforgeable *if for all adversaries A,*

$$\mathbf{Adv}_A^{\mathsf{MAC}}(\lambda) \leq \mathrm{negl}(\lambda)$$

.

$\underline{\mathsf{Init}()}$
$K \xleftarrow{\$} \mathcal{K}$
return K
$\underline{\mathsf{Query}(x)}$
$s \leftarrow \mathsf{MAC}_K(x)$
$Q \leftarrow Q \cup \{(x, s)\}$
return x

$\underline{\mathsf{Challenge}(x', s')}$
if $\mathsf{Verify}_K(x', s') =$ `ACCEPT` and
$(x', s') \notin Q$ **then**
`win` $\leftarrow$ `true`
end if
$\underline{\mathsf{Final}()}$
return `win`

Figure 6 – Chosen-messages attack game CMA for MACs.

B Security definitions for verifiable hash tables

We describe the security games for the verifiable hash tables, as stated in Section 4.1, to define completeness and soundness of these verifiable hash tables.

The VHTComp is relatively straightforward: it simulates a regular execution of a verifiable hash table (the adversary does not try to cheat by providing adversarial input to the VHT functions). The VHTSound is a bit more involved as it allows this behavior: not only the adversary can try to forge a valid proof for an invalid answer (in the Challenge procedure), but he can also try to tamper the client's state by giving him wrong refresh informations (in the Update procedure).

Similarly to the previous definitions, for an adversary A we define the following:

$$\mathbf{AdvComp}_A^{\mathsf{VHT},\Theta}(\lambda) = \mathbb{P}[\mathrm{VHTComp}_A^{\Theta}(\lambda) = 1]$$
$$\mathbf{AdvSnd}_A^{\mathsf{VHT},\Theta}(\lambda) = \mathbb{P}[\mathrm{VHTSound}_A^{\Theta}(\lambda) = 1]$$

Finally, we say that Θ is *correct* if, for all adversaries A, $\mathbf{AdvComp}_A^{\mathsf{VHT},\Theta}(\lambda)$ is negligible, and that Θ is *sound* if, for all adversaries A, $\mathbf{AdvSnd}_A^{\mathsf{VHT},\Theta}(\lambda)$ is negligible.

$\underline{\mathsf{Init}(T)}$
$(K_{\mathsf{VHT}}, \mathsf{VHT}, \sigma_{\mathsf{VHT}}) \leftarrow \mathsf{VHTSetup}(T)$
return $(\mathsf{VHT}, \sigma_{\mathsf{VHT}})$
$\underline{\mathsf{Update}(u)}$
(VHT', π)
$\quad \leftarrowtail \mathsf{VHTUpdate}(T, \mathsf{VHT}, u)$
σ'_{VHT}
$\quad \leftarrowtail \mathsf{VHTRefresh}(K_{\mathsf{VHT}}, \sigma_{\mathsf{VHT}}, \pi, u)$
if $\sigma'_{\mathsf{VHT}} \neq \texttt{REJECT}$ **then**
$T \leftarrow \mathsf{Apply}(T, u)$
$\mathsf{VHT} \leftarrow \mathsf{VHT}'$
$\sigma_{\mathsf{VHT}} \leftarrow \sigma'_{\mathsf{VHT}}$
end if
return $(T, \mathsf{VHT}, \sigma_{\mathsf{VHT}})$
$\underline{\mathsf{Challenge}(\mathsf{hkey})}$
$(v, \pi) \leftarrow \mathsf{VHTGet}(T, \mathsf{VHT}, \mathsf{hkey})$
if $T[\mathsf{hkey}] \neq v$ or
$\quad \mathsf{VHTVerify}(K_{\mathsf{VHT}}, \sigma_{\mathsf{VHT}}, \mathsf{hkey}, v, \pi)$
$\quad\quad = \texttt{REJECT}$ **then**
$\quad \texttt{win} \leftarrow \texttt{true}$
end if
return (v, π)

$\underline{\mathsf{Init}(T)}$
$(K_{\mathsf{VHT}}, \mathsf{VHT}, \sigma_{\mathsf{VHT}}) \leftarrow \mathsf{VHTSetup}(T)$
return $(\mathsf{VHT}, \sigma_{\mathsf{VHT}})$
$\underline{\mathsf{Query}(\mathsf{hkey})}$
$(v, \pi) \leftarrow \mathsf{VHTGet}(T, \mathsf{VHT}, \mathsf{hkey})$
return (v, π)
$\underline{\mathsf{Update}(u)}$
(VHT', π)
$\quad \leftarrowtail \mathsf{VHTUpdate}(T, \mathsf{VHT}, u)$
The game gives (VHT', π) to the adversary and gets back $\widetilde{\pi}$
σ'_{VHT}
$\quad \leftarrowtail \mathsf{VHTRefresh}(K_{\mathsf{VHT}}, \sigma_{\mathsf{VHT}}, \widetilde{\pi}, u)$
if $\sigma'_{\mathsf{VHT}} \neq \texttt{REJECT}$ **then**
$T \leftarrow \mathsf{Apply}(T, u)$
$\mathsf{VHT} \leftarrow \mathsf{VHT}'$
$\sigma_{\mathsf{VHT}} \leftarrow \sigma'_{\mathsf{VHT}}$
end if
return $(T, \mathsf{VHT}, \sigma_{\mathsf{VHT}})$
$\underline{\mathsf{Challenge}(\mathsf{hkey}, v, \pi)}$
if $T[\mathsf{hkey}] \neq v$ and
$\quad \mathsf{VHTVerify}(K_{\mathsf{VHT}}, \sigma_{\mathsf{VHT}}, \mathsf{hkey}, v, \pi)$
$\quad\quad = \texttt{ACCEPT}$ **then**
$\quad \texttt{win} \leftarrow \texttt{true}$
end if

Figure 7 – Completeness (VHTCOMP - left) and soundness (VHTSOUND - right) games for verifiable hash tables.

C Security of the Static Verifiable Hash Table

We show in this section the soundness of the VHT construction of Section 4.2 (Algorithm 1).

Consider the reduction from an adversary $\mathcal{A}$ on the VHT instantiation to the CMA game adversary $\mathcal{B}$: every time the instantiation needs to compute MAC_K, we replace the call by the Query oracle of the CMA game. This only happens during the VHTSetup procedure. When $\mathcal{A}$ produces a forgery (key, v, π), we will forward it to the CMA game and produce a MAC forgery.

If the forgery is such that $v \neq \perp$, π can be parsed as (i, s) and $((key, v, i), s)$ is forwarded to CMA. If $\mathcal{A}$ wins the soundness game with this forgery, it implies that $(v, \pi) \neq \mathsf{VHTGet}(T, \mathsf{VHT}, key)$, hence that $((key, v, i), s)$ is not in the transcript of CMA (CMA's Query is called only during the setup phase). It also implies that VHTVerify accepts, *i.e.* that the MAC verification succeeded, and that the forwarded forgery was a valid unseen forgery.

If the forgery is such that $v = \perp$, π can be parsed as $(i, key_-, v_-, s_-, key_+, v_+, s_+)$. We know that $\pi \neq (i^{real}, key_-^{real}, v_-^{real}, s_-^{real}, key_+^{real}, v_+^{real}, s_+^{real})$ where the *real* values are the one generated by $\mathsf{VHTGet}(T, \mathsf{VHT}, key)$. Let us separate two cases:

- $i \neq i^{real}$: as VHTVerify accepts, we can suppose that (key_-, v_-, i, s_-) and $(key_+, v_+, i+1, s_+)$ are not both entries of VHT. If that were the case, it would mean that the condition $key_- < key < key_+$ is verified. But, by construction, we know that key_-^{real} and key_+^{real} is the only pair of *consecutive* keys verifying this condition, and it would contradicts $i \neq i^{real}$.
  Hence, if (key_-, v_-, i, s_-) is not an entry of VHT, then $((key_-, v_-, i), s_-)$ does not appear in the transcript of the CMA game and is a valid MAC forgery. The same applies if $(key_+, v_+, i+1, s_+)$ is not an entry of VHT.

- $i = i^{real}$, suppose WLOG, that there is a difference on the $-$ components of the proof. Then (key_-, v_-, i, s_-) is a valid unseen (because never computed in VHTSetup) MAC forgery.

From the previous study, we directly have

$$\mathbf{Adv}_{\mathcal{A}}^{\text{VHTSOUND}}(\lambda) \leq \mathbf{Adv}_{\mathcal{B}}^{\text{CMA}}(\lambda).$$

D Proof of VSSE Lower Bounds

We give here the proof of Theorem 2 which is restated below.

Theorem 8. *Let Π be a VSSE scheme with client memory of size $s \leq |W|^{1-\varepsilon}$ for some $\varepsilon > 0$, where w is the number of distinct stored keywords (keywords with at least one matching document). Then, either* Search *queries have minimal computational complexity $\Omega(\mathsf{max}(\frac{\log|W|}{\log\log|W|}, m))$ or* Update *queries have minimal computational complexity $\Omega(\frac{\log|W|}{\log\log|W|})$.*

Proof. Search queries cannot be computed in less than $O(m)$ operations, trivially. So in the following, we will just show that they cannot either be executed in less than $O(\frac{\log|W|}{\log\log|W|})$.

Let us first consider the memory checker $\mathcal{C}$ built over an SSE scheme Π as follows: to read index i, $\mathcal{C}$ calls $\Pi.\mathsf{Search}(i)$ and returns the result, to write value v at index i, $\mathcal{C}$ first calls $\Pi.\mathsf{Search}(i)$ and gets a single value v', runs $\Pi.\mathsf{Update}(\mathsf{del}, i, v')$ and $\Pi.\mathsf{Update}(\mathsf{add}, i, v)$.

In this execution, Π stores exactly one value (document) per index (keyword), so the Search calls will only return a single entry. The size s of private memory $\mathcal{C}$ uses is $s = s' + c$ where s' is the private memory of Π and c a constant independent of $n = |W|$. Hence, if Π is a VSSE scheme with private memory of size $s' \leq n^{1-\varepsilon}$ with $\varepsilon > 0$, $\mathcal{C}$ is using less than $n^{1-\varepsilon'}$ private memory for some $\varepsilon' \geq \varepsilon > 0$.

If both Search and Update had an (amortized) complexity smaller than $O(\frac{\log|W|}{\log\log|W|})$, $\mathcal{C}$ would break the lower bound of Theorem 1. Hence, one of Search or Update has complexity at least $\Omega(\frac{\log|W|}{\log\log|W|})$. □

E Soundness proof of Verif-SPS

We give here the full proof of the soundness of the Verif-SPS scheme. We will use the following, two step, strategy:

1. We create a reduction, using hybrids, to a derivative of the game SSESOUND were the values given by the adversary are 'sound', *i.e.* are either correctly verified or fail verification. For example, a value issued from a verifiable hash table will never be a successful forgery.

2. We show that this game cannot be won by any adversary.

E.1 Reduction

We construct 4 games G_0 to G_3, G_0 being (almost) the original SSESOUND security game. We will go from G_0 to G_3 by successively assuming the soundness of the verifiable hash table Θ, the authenticity of the encryption scheme AEnc, and finally the correctness of $\mathsf{Verif\text{-}SPS}$.

Game G_0 G_0 is exactly the game $\text{SSESOUND}_A^{\mathsf{Verif\text{-}SPS}}$, up to one difference: for every VHT or authenticated encryption verification, if a forgery actually succeeded, G_0 will immediately set the flag `win` to `true`. More formally, each call to $\mathsf{VHTVerify}(\sigma_{\mathsf{VHT}_\ell}, \mathsf{hkey}, entry, \pi)$ is followed by the pseudo-code

if $\mathbf{T}_\ell[\mathsf{hkey}] \neq entry$
  and $\mathsf{VHTVerify}(\sigma_{\mathsf{VHT}_\ell}, \mathsf{hkey}, entry, \pi) = \texttt{ACCEPT}$
 `win` $\leftarrow$ `true`

and the decryptions $tuple \leftarrow \mathsf{ADec}(esk, (\ell, r_\ell), c_2)$ by

if $tuple \neq$ `REJECT` and c_2 is not an encryption of $tuple$
 `win` $\leftarrow$ `true`

The adversary A has more chances to win G_0 than the original SSESOUND game and we have

$$\mathbb{P}[\text{SSESOUND}_A^{\mathsf{Verif\text{-}SPS}}(\lambda) = 1] \leq \mathbb{P}[G_0 = 1]$$

Game G_1 In game G_1, all calls to $\mathsf{VHTVerify}(\sigma_{\mathsf{VHT}_\ell}, \mathsf{hkey}, entry, \pi)$ (including the pseudo code added in G_0) are replaced by the following:

if $\mathbf{T}_\ell[\mathsf{hkey}] \neq entry$
 return `REJECT`
return $entry$

Said otherwise, we suppose that all the $\mathsf{VHTVerify}$ calls perform as expected, rejecting forgery tentatives.

It is important to notice that, at every update, exactly one level is rebuilt. So, let us consider sub-hybrids $G_0^0, G_0^1, \ldots, G_0^N$, such that for all i, in G_0^i, all the tables rebuilt before the i-th update (included) use the upper pseudo-code for verification, and all the other tables rebuilt after this update use the regular verification procedure. We have that, for all $1 \leq i \leq N$, there exists an adversary B_i such that

$$\mathbb{P}[G_0^i = 1] - \mathbb{P}[G_0^i = 1] \leq \mathbf{AdvSnd}_{B_i}^{\mathsf{VHT},\Theta}(\lambda).$$

B_i tries to attack the soundness of the table rebuild at the i-th update. Note that B_i does not make any call to Update in the game VHTSOUND (the tables are static), and the number of calls to$\mathsf{Challenge}$ is upper bounded by the number of tokens sent by A.

Summing the probabilities, as $G_0^0 = G_0$ and $G_0^N = G_1$, we have that there exists an adversary B such that

$$\mathbb{P}[G_1 = 1] - \mathbb{P}[G_0 = 1] \leq N \cdot \mathbf{AdvSnd}_B^{\mathsf{VHT},\Theta}(\lambda).$$

Game G_2 In game G_2, all the tentative forgeries of authenticated ciphertexts will fail: the adversary has to give to the game valid ciphertext or these will be rejected.

Notice that, as we have from G_1 that all the entries issued from the verifiable hash table are valid, in particular, their field c_2 has to be valid too, and hence decrypt correctly, without rejection. It implies that the adversary can win G_1 by providing forged ciphers is in the $\mathsf{Rebuild}$ protocol. As a consequence, in

G_2, we remove all the code added in G_0 after calls to ADec in the Search protocol, without modifying the adversary's success probability. In the rest of the paragraph, we will focus only on the Rebuild protocol.

When running Rebuild, when decrypting a tuple, the game will always know what result he should expect: the algorithms are deterministic, except for the encryption but it does not influence the order the tuples are treated. So, we modify the game G_1 the following way to give G_2: in G_2 all calls to ADec are followed by:

> **if** $(w, \ell^*, \mathsf{ind}, \mathsf{op}, cnt)$ has not been encrypted with the same authenticated data as the one used for decryption
>
> **return** `REJECT`

We also do this in the o-sort protocol, which uses an authenticated encryption scheme (and associated data too).

In G_2, the game accepts to decrypt only ciphertexts that have been generated by the game itself with the additional data that are used for the decryption. We are exactly in the case of the authentication security game for encryption schemes and hence, there exists an adversary C such that

$$\mathbb{P}[G_2 = 1] - \mathbb{P}[G_1 = 1] \leq \mathbf{Adv}_C^{\mathsf{Auth},\mathsf{AEnc}}(\lambda)$$

C sees all the encryptions generated by the Rebuild algorithms (including the o-sort protocol), and tries to forge at most once per Rebuild call (remember that the protocol halts as soon as it gets `REJECT`) *i.e.* at most N times.

We also recall that, as we supposed o-sort to be secure against malicious adversaries, when enumerating the entries of $\hat{\mathbf{B}}$, they come in sorted order, and are the same as the entries of the input table.

Game G_3 In game G_3, we will suppose that all the lookups behave normally, or said otherwise, that when looking for keyword w, with operation op and counter cnt at level ℓ, $\mathsf{Lookup}(\mathsf{token}_\ell, \mathsf{op}, cnt)$ will actually return an entry for keyword w. Still an other way to see it is to say that the schemes behave exactly as expected on the conceptual data structure Γ. In particular, in implies that there is no collision when computing token_ℓ or hkey. This is ensure by the correctness of the original SPS construction, and we have that there exists an adversary D such that

$$\mathbb{P}[G_3 = 1] - \mathbb{P}[G_2 = 1] \leq \mathbf{AdvCor}_D^{\mathsf{SSE},\mathsf{SPS}}(\lambda)$$

Finally, when we sum up the contribution of all the games, we infer that

$$\begin{aligned}\mathbb{P}[\mathbf{SSESound}_A^{\mathsf{Verif\text{-}SPS}}(\lambda) = 1] \leq \mathbb{P}[G_3 = 1] \\ + N \cdot \mathbf{AdvSnd}_B^{\mathsf{VHT},\Theta}(\lambda) \\ + \mathbf{Adv}_C^{\mathsf{Auth},\mathsf{AEnc}}(\lambda) \\ + \mathbf{AdvCor}_D^{\mathsf{SSE},\mathsf{SPS}}(\lambda)\end{aligned}$$

In the next section, we will show that $\mathbb{P}[G_3 = 1] = 0$.

E.2 Soundness of G_3

Without loss of generality, we can suppose that the adversary never gives `REJECT` to the game G_3: when it the case, the current procedure would immediately halt and return `REJECT`, and the flag `win` will not be set

to `true`. As a consequence, in this section, we will suppose that none of the procedures Search or Update return `REJECT`. It implies that, in G_3, the entries given by the server to the client are genuine, and that ciphertexts decrypted by the client were previously encrypted with the same authenticated data.

The first thing we want to prove is that the adversary cannot corrupt the table $\mathbf{T}_\ell$: its content always reflects the conceptual data structure Γ_ℓ.

Proposition 9. *At anytime in the execution of the game, if* $(\ell^*, \mathsf{ind}) = \Gamma_\ell[w, \mathsf{op}, cnt]$ *then, in* G_3, $\mathsf{Lookup}(\mathsf{token}_\ell, \mathsf{op}, cnt) = (\ell^*, \mathsf{ind}, ., .)$ *with* $\mathsf{token}_\ell = F(k_\ell, h(w))$

Proof. Because of the timestamp r_ℓ and the flags used during the rebuilding, ciphertexts generated cannot be replayed. More precisely, all the ciphertexts generated for level ℓ before the reconstruction cannot be reused because of the new value of the timestamp r_ℓ that was incremented upon rebuild. Then, ciphertexts generated before the first call to o-sort cannot be reused in place of the ciphertexts generated between the two calls to o-sort which themselves cannot be reused later: the firsts are encrypted using $(\ell, r_\ell, 0)$ as associated data, while the seconds use $(\ell, r_\ell, 1)$ and the lasts only (ℓ, r_ℓ).

We conclude with the correctness of the oblivious sort algorithm: at the end of the protocol's execution, the entries are always correctly sorted in $\mathbf{T}_\ell$, unless it returned `REJECT`.

□

Now, we can focus on the Search protocol. We only treat the sublinear case, as the soundness of the basic construction is immediate, and we will proceed level per level: the result set is the (disjoint) union of the results produced by each level. For lemmas will be proven:

- $\mathsf{AllHoles}[\ell]$ forms a partition of the del entries in level ℓ. Moreover, in each hole component, entries are consecutive.
- At level ℓ, all the add entries returned by the server in $\mathcal{I}_\ell$ match no del entry in lower levels.
- At level ℓ, every add entry in a hole defined by $\mathcal{I}_\ell$ match a del entry in a lower level.
- At level ℓ, every add entry has been returned by the server, either as a non-deleted entry, or an entry belonging to a hole.

Lemma 10. *Let* ℓ *be a level, and* AllHoles *the hole table sent by the server after a search request with keyword* w*. Let* $((cnt^i_x, entry^i_x, \pi^i_x), (cnt^i_y, entry^i_y, \pi^i_y))_{i=0}^s$ *be the list* $\mathsf{AllHoles}[\ell]$.

Then $(\{\Gamma_\ell[w, \mathsf{del}, cnt]\}_{cnt=cnt^i_x}^{cnt^i_y})_{i=0}^s$ *is a partition of* $\{\Gamma_\ell[w, \mathsf{del}, cnt]\}_{cnt=0}^{cnt_{last}}$ *where* cnt_{last} *is the largest integer such that* $\Gamma_\ell[w, \mathsf{del}, cnt_{last} + 1] \neq \perp$

Proof. This lemma entirely relies on the procedure CheckAdjacency. It checks that the hole components in $\mathsf{AllHoles}[\ell]$ are consecutive: if $(\ell^*, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$ and $(\ell'^*, cnt'_x, cnt'_y, \mathsf{ind}'_x, \mathsf{ind}'_y)$ are successive elements in $\mathsf{AllHoles}[\ell]$, then $cnt_x \leq cnt_y$, $cnt'_x \leq cnt'_y$ and $cnt'_x = cnt_y + 1$. Hence, when CheckAdjacency accepts, we know that the sets $\{\Gamma_\ell[w, \mathsf{del}, cnt]\}_{cnt=cnt^i_x}^{cnt^i_y}$ are pairwise disjoint.

This is ensured for cnt_x starting at 0: if the first element of $\mathsf{AllHoles}[\ell]$ has not $cnt_x = 0$, the test fails. Also, in VerifyHoles, we checked that, if cnt_{last} is the value of cnt_y for the last element of $\mathsf{AllHoles}[\ell]$, it has to be that $\Gamma_\ell[w, \mathsf{del}, cnt_{last}] \neq \perp$ while $\Gamma_\ell[w, \mathsf{del}, cnt_{last} + 1] \neq \perp$. The verification spanned all the possible counter values for del entries at level ℓ. Thus, $(\{\Gamma_\ell[w, \mathsf{del}, cnt]\}_{cnt=cnt^i_x}^{cnt^i_y})_{i=0}^s$ is a partition of $\{\Gamma_\ell[w, \mathsf{del}, cnt]\}_{cnt=0}^{cnt_{last}}$.

□

Lemma 11. *Let ℓ be a level and $\mathcal{I}_\ell$ the result list returned by the server for this level after a search request with keyword w. Let $(entry, \pi)$ be an* add *element of $\mathcal{I}_\ell$, with $entry$ decrypting to $(w, \ell, \mathsf{ind}, \mathsf{add}, cnt)$. If* Search *did not return* REJECT*, there is no* del *entry matching (ind, ℓ) in any level $\ell' < \ell$: for all c, $\ell' < \ell$, $\Gamma_{\ell'}[w, \mathsf{del}, c] \neq (\mathsf{ind}, \ell)$.*

Proof. For this lemma, we use the guaranties offered by CheckResults, and in particular on the fact that the inner variable ind_{last} strictly increases when enumerating the elements of $\mathcal{I}_\ell$.

As a consequence, all the hole components $(\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$ in $\mathcal{I}_\ell$ encountered before $entry$ are such that $\mathsf{ind}_y < \mathsf{ind}$ (CheckResults would have failed otherwise), and all the hole components $(\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y$ in $\mathcal{I}_\ell$ encountered after $entry$ are such that $\mathsf{ind}_x > \mathsf{ind}$.

We conclude using Lemma 10: if there is cnt_{del} and $\ell' < \ell$ such that $\Gamma_{\ell'}[w, \mathsf{del}, c] = (\mathsf{ind}, \ell)$, there is a hole component $(\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$ with $\mathsf{ind}_x \leq \mathsf{ind} \leq \mathsf{ind}_y$.

□

Lemma 12. *Let ℓ be a level and $\mathcal{I}_\ell$ the result list returned by the server for this level after a search request with keyword w. Let* Hole *be a hole in $\mathcal{I}_\ell$, $(entry, \pi)$ and $(entry', \pi')$ the two* add *entries flanking* Hole *in $\mathcal{I}_\ell$. $entry$ (resp. $entry'$) decrypts to $(w, \ell, \mathsf{ind}, \mathsf{add}, cnt)$ (resp. $(w, \ell, \mathsf{ind}', \mathsf{add}', cnt')$). If* Hole *is the first element of $\mathcal{I}_\ell$, we set $cnt = 0$, and if it is the last element of $\mathcal{I}_\ell$, we set cnt to the smallest integer such that $\Gamma_\ell[w, \mathsf{add}, cnt] = \perp$.*

Then, for every $cnt < c < cnt'$, $(\mathsf{ind}_c, \ell) = \Gamma_\ell[w, \mathsf{add}, c]$, there exists $\ell' < \ell$ and c_{del} such that $(\mathsf{ind}_c, \ell) = \Gamma_{\ell'}[w, \mathsf{del}, c_{\mathsf{del}}]$. There also is a hole component $h \in$ Hole, $h = (\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$ with $cnt_x \leq c_{\mathsf{del}} \leq cnt_y$, $\mathsf{ind}_y < \mathsf{ind} < \mathsf{ind}_x$.

Proof. Let us consider the set

$$\Delta = \bigcup_{h \in \mathsf{Hole}} \left\{ \Gamma_{\ell'}[w, \mathsf{del}, cnt] \;\middle|\; \begin{array}{l} h = (\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y) \\ \text{and } cnt_x \leq cnt \leq cnt_y \end{array} \right\}$$

Because we know that the elements of the union are pairwise disjoint, and because we checked the size of the hole by summing the difference $cnt_y - cnt_x + 1$ for every component, we have that

$$|\Delta| = cnt' - cnt - 1$$

which is the number of c such that $cnt < c < cnt'$.

Let $h = (\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$ be a hole component. As we saw earlier, we know that $T_{\ell'}$ reflects the conceptual table $\Gamma_{\ell'}$, for each $cnt_x \leq c \leq cnt_y$, $\Gamma_{\ell'}[w, \mathsf{del}, c] = (\mathsf{ind}_c, \ell)$. We also know that every del entry in $\Gamma_{\ell'}[w, \mathsf{del}, c] = (\mathsf{ind}_c, \ell^*)$ (no condition on c)has a matching entry $\Gamma_{\ell^*}[w, \mathsf{add}, c_{\mathsf{add}}] = (\mathsf{ind}_c, \ell^*)$. Given Lemma 10, as the hole components at level ℓ' form a partition of successive del entries at level ℓ', it must be that for all $cnt_x \leq c \leq cnt_y$, $\mathsf{ind} < \mathsf{ind}_x \leq \mathsf{ind}_c \leq \mathsf{ind}_y < \mathsf{ind}'$, where $\Gamma_{\ell'}[w, \mathsf{del}, c] = (\mathsf{ind}_c, \ell)$.

This gives us that there are $cnt' - cnt - 1$ distinct add entries at level ℓ whose indices are strictly comprised between ind and ind$'$. From $entry$ and $entry'$, we also know that there are exactly $cnt' - cnt - 1$ distinct add entries at level ℓ, $\Gamma_\ell[w, \mathsf{add}, c] = (\mathsf{ind}_c, \ell)$ with $cnt < c < cnt'$, and $\mathsf{ind} < \mathsf{ind}_c < \mathsf{ind}'$. These have to be the same ones, proving the lemma.

□

Lemma 13. *Let ℓ be a level and $\mathcal{I}_\ell$ the result list returned by the server for this level after a search request with keyword w. For every cnt such that $\Gamma_\ell[w, \mathsf{add}, cnt] = (\mathsf{ind}, \ell)$, either there is an entry $entry \in \mathcal{I}_\ell$ decrypting to $(w, \ell, \mathsf{ind}, \mathsf{add}, cnt)$ or there is a hole* Hole *$\in \mathcal{I}_\ell$ with a component $h = (\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$ with $\mathsf{ind}_x \leq \mathsf{ind}$, and $\mathsf{ind} \leq \mathsf{ind}_y$.*

Proof. Let cnt_{last} be the last counter value encountered when enumerating $\mathcal{I}_\ell$: if its last element is an add entry decrypting to $(w, \ell, \mathsf{ind}, \mathsf{add}, cnt)$, $cnt_{last} = cnt$, if its last element is a hole $(\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$, $cnt_{last} = cnt$.

CheckResults ensured that $\Gamma_\ell[w, \mathsf{add}, cnt_{last}] \neq \perp$ and $\Gamma_\ell[w, \mathsf{add}, cnt_{last} + 1] = \perp$. It means that the add entries' count at level ℓ span the interval $[0, cnt_{last}]$.

Let c be in this interval. Suppose that, in $\mathcal{I}_\ell$, there is no entry decrypting to $(w, \ell, \mathsf{ind}_c, \mathsf{add}, c)$. We also know that in $\mathcal{I}_\ell$, the value of the counter of add entries strictly increases. Hence, there is two counters $cnt < c < cnt'$ satisfying the following conditions: there is two add entries $entry$ and $entry'$ in $\mathcal{I}_\ell$ decrypting to $(w, \ell, \mathsf{ind}, \mathsf{add}, cnt)$ and $(w, \ell, \mathsf{ind}', \mathsf{add}, cnt')$, $entry'$ is the add entry following $entry$ in $\mathcal{I}_\ell$.

$entry$ and $entry'$ cannot be immediately successive in $\mathcal{I}_\ell$: if that were the case, we would have $cnt' = cnt + 1$ and one of $entry$ or $entry'$ would decrypt to $(w, \ell, \mathsf{ind}_c, \mathsf{add}, c)$. So, there is the hole Hole between these in $\mathcal{I}_\ell$, and with Lemma 12, we conclude that there is a hole component $h = (\ell', \ell, cnt_x, cnt_y, \mathsf{ind}_x, \mathsf{ind}_y)$ with $\mathsf{ind}_x \leq \mathsf{ind}_c \leq \mathsf{ind}_y$. □

With these lemmas, we can now easily show the perfect soundness of the hybrid of game G_3.

Theorem 7. As before, we proceed level per level. So let ℓ be a level, and cnt such that $\Gamma_\ell[w, \mathsf{add}, cnt] \neq \perp$.

Lemma 13 tells us that there is either a corresponding entry $entry$ or a correspond hole. In case of an add entry, Lemma 11 guaranties us that $\Gamma_\ell[w, \mathsf{add}, cnt]$ has not been deleted in a lower level. In case of a hole, Lemma 12 implies that $\Gamma_\ell[w, \mathsf{add}, cnt]$ as a matching del entry in a lower level.

In the sublinear Search algorithm, the result set is constructed from the add entries of the $\mathcal{I}_\ell$ lists, which correspond to all the $\Gamma_\ell[w, \mathsf{add}, cnt]$ that have not been deleted. If Search did not return `REJECT` in game G_3, it returned the correct result set $\mathsf{DB}(w)$.

$$\mathbb{P}[G_3 = 1] = 0.$$

Combined with the reduction of the previous section

$$\begin{aligned}\mathbb{P}[\text{SSESound}_A^{\mathsf{Verif\text{-}SPS}}(\lambda) = 1] \leq & + N \cdot \mathbf{AdvSnd}_B^{\mathsf{VHT},\Theta}(\lambda) \\ & + \mathbf{Adv}_C^{\mathsf{Auth},\mathsf{AEnc}}(\lambda) \\ & + \mathbf{AdvCor}_D^{\mathsf{SSE},\mathsf{SPS}}(\lambda)\end{aligned}$$

□

F Time Complexity of the Verifiable SPS Constructions

F.1 Complexity of the basic scheme

Let $T^{\text{build}}(n)$ be the complexity of VHTSetup for an input table of size n, $T^{\text{prf}}_{\in}(n)$ (resp. $T^{\text{prf}}_{\perp}(n)$) the complexity of VHTGet, *i.e.* the proving complexity, when queried on a key in the table (resp. a non-member key), and $T^{\text{chk}}_{\in}(n)$ (resp. $T^{\text{chk}}_{\perp}(n)$) the complexity of VHTVerify, *i.e.* the verification complexity, for a proof of an element in the table (resp. an non member element).

The Rebuild (without the VHT) algorithm takes time $O(2^\ell \ell)$ when ℓ is the empty level to be filled, as the bottleneck phase is the oblivious sorting algorithm [GM11]. Computing the VHT induces an extra $T^{\text{build}}(2^\ell)$ cost, as well as using authenticated encryption and the additional sanity checks. Hence, the worst case complexity of Update is $O(N \log N) + T^{\text{build}}(N)$ and $O(\log^2 N) + \sum_{\ell=0}^{L-1} T^{\text{build}}(2^\ell)/N = O(\log^2 N +$

$\frac{\log N}{N} T^{\text{build}}(N))$ amortized time, as each level ℓ is rebuilt every 2^ℓ updates, each rebuild being of $O(\ell \cdot 2^\ell) + T^{\text{build}}(2^\ell)$ computational complexity.

The basic Search algorithm took time $O(\alpha + \log N)$ in the passive adversary setting of [SPS14], where α is the number of times the keyword was added to the database. In our case, it increases to $O(\alpha(T_\in^{\text{prf}}(n) + T_\in^{\text{chk}}(n)) + \log N(T_\perp^{\text{prf}}(n) + T_\perp^{\text{chk}}(n)))$. To be more precise, the server needs $O(\alpha T_\in^{\text{prf}}(n) + \log N)$ to find through all entries with keyword w and get the associated proof, and for each level, it needs to produce two proofs for $\perp$ (one for an add entry, the other for a del entry), which takes $O(\log N \cdot T_\perp^{\text{prf}}(n))$ time total.

Hence, using the instantiation of Section 4.2, we end up with $O(\alpha + \log^2 N)$ search time and $O(\log^2 N)$ amortized update time.

F.2 Complexity of the Sublinear Scheme

As stated in Section 6.4.1, we reuse the same Rebuild algorithm as before, and directly have that Update take $O(\log^2 N + \frac{\log N}{N} T^{\text{build}}(N))$ amortized time, and $O(N \log N) + T^{\text{build}}(N)$ in the worst case, using the conclusions of the previous section. We will focus here on the Search algorithm (Algorithms 6, 7, 8, and 9).

First, let us study ProcessLevel Namely, the server sends to the client L lists $\mathcal{I}_\ell$ containing add entries and holes. There are exactly m such add entries total, and we know that a hole entry must me immediately be followed by an add entry, and there are at most m holes in the lists $\mathcal{I}_\ell$. To ensure an add entry has no associated del entry, the server has to perform an unsuccessful binary search in every level, taking $O(\log^2 N)$ time. To compute a hole, SkipHole performs a binary search on the add entries of the current level ℓ, an hence calls DeletedSum $O(\log N)$ times. Then, DeletedSum goes through all the lower levels and itself performs a binary search, taking $O(\log^2 N)$. Finally, for each level, ProcessLevel produces a proof for a key not present in the VHT, taking $O(\log N)$. If we sum all these contributions, ProcessLevel takes time $O(m \log^3 N)$.

ProveHoles enumerates the hole components. As there are at most m holes, there are at most $m \log N$ hole components. For each of these components, ProveHoles generate a VHT proof in constant time (components limits are real entries in the level). It also generates a "not found" proof for every level in $O(\log N)$. ProveHoles takes time $O(m \log N \cdot T_\in^{\text{prf}}(N) + \log N \cdot T_\perp^{\text{prf}}(N))$.

On the verifier/client side, VerifyHoles verifies $O(m \log N)$ of these proofs and calls CheckAdjacency, which itself goes through all the $O(m \log N)$ hole components. VerifyHoles also verifies the $O(\log N)$ non-membership proofs in $O(T_\perp^{\text{chk}}(N))$ time. Finally, CheckResults also uses the hole components and $\mathcal{I}_\ell$'s add entries sequentially, each of them being processed in constant time.

The total verification time is $O(m \log N \cdot T_\in^{\text{chk}}(N) + \log N \cdot T_\perp^{\text{chk}}(N))$, and the total time complexity of the search algorithm is

$$\begin{aligned} O(&m \log^3 N \\ &+ m \log N(T_\in^{\text{prf}}(N) + T_\in^{\text{chk}}(N)) \\ &+ \log N(T_\perp^{\text{prf}}(N) + T_\perp^{\text{chk}}(N))). \end{aligned}$$

Once instantiated with Section 4.2, it gives us a search complexity of $O(m \log^3 N)$.

G Practical Evaluation of Verifiable SSE

G.1 Minimal Practical Overhead of Search Queries

As we have seen in Section 3, and with constructions of Section 5, to verify the results of a search query, the client will have to go through all the returned results and check against these a proof. Given the presented

construction, we can legitimately claim that the work that has to be performed by the client is comparable to the computation of a hash or a MAC over the result set.

To estimate the practical overhead, we suppose that a query generating 10 000 results has been processed by the server. Each result's index is represented over λ bits (here we take $\lambda = 128$), and the byte string representing the result set spans over $16 \cdot 10^4$ bytes. Using PMAC [RBB03], this string can be processed at around 1.6 cycle per byte [BLT14], *i.e.* $25.6 \cdot 10^4$ cycles in total. For a 2 GHz CPU, this will be done in around 0.1 ms.

However, the GSV construction needs a set hashing function, which is more computationally expensive than a MAC or a regular hash function. Recent implementations of set hashing gives that one can hash about 1.5 million elements per second on a modern 2 GHz CPU [Tib15], which would give us a verification time of about 7 ms for a 10 000 results query.

G.2 Practical Evaluation of Verifiable SPS

In this section, we estimate the practical verification costs implied by the algorithms of Sections 6.3 and 6.4.

The Rebuild protocol (and the Update calls) remained almost unchanged, except that authenticated encryption is used instead of regular randomized encryption. In practice, as the authenticated data is quite small: authenticated data comprises at most three integers of size roughly $\log N$, while the block cipher used in the AEAD scheme takes usually blocks of 128 bits as input (*e.g.* for AES), making packing of these three integers in a single block possible even for large data bases (up to 2^{42} additions/deletions of keyword-document pairs). The cost of this additional block cipher call will be negligible, and with most of the modern AEAD schemes, and it can even be parallelized [RBB03].

For the search, let us focus first on the server side. When instantiated with the construction of Section 4.2, proving the validity of a found entry is free as the proof for such entries is the MAC of the entry and its position in the hash table, MAC which is stored with the entry itself. The proving cost comes only from the binary search needed by the proof construction for non member keys. However, we claim that this cost is negligible compared the search cost: the server has to construct $2L$ of those proofs (one per level and per operation – add and del), performing (at most) $\log N$ comparisons for each of them giving $2\log^2 N$ total comparisons, while the SkipHole function performs $\log^3 N$ comparisons per found result due to the unparallelizable and interleaved binary searches.

On the verifier (client) side, Search has to verify at most $m \cdot L$ proofs for keys in the tables and $2L$ proofs for keys not in the tables. The total number of MACs to compute and compare is at most $(m+4)L$ (recall that this is the worst case, when every search result is followed by a hole), each MAC being done over the tuple of data $(w, \ell, \ell^*, r_\ell, \mathsf{ind}, \mathsf{op}, cnt)$, the position in the table and the IV used for the encryption, *i.e.* a string of about $4\log N + 2\log\log N + 2\lambda + 1$ bits. For $\lambda = 128$ and $N < 2^{40}$ ($L = 40$) as taken before, the string to be MACed spreads over 429 bits, and will need 4 128-bits block cipher calls to be processed, and the total number of additional block cipher calls due to verification is about $4 \cdot L = 160$ per matching document on an very large data base. Suppose a search keyword matched 10 000 documents. Search would have to verify around 400 000 authenticated encryptions of 54 bytes. A modern desktop class CPU running at 2GHz can compute AES-OCB3 at around 1.6 cycle per byte [BLT14], and thus run all the verifications in around 17 ms. Finally, the CheckResults function enumerates all the results and hole components, performing at most $m \log N$ comparisons and additions over $\log N$ bits, operations a lot less expensive than AES evaluation.

From this rough evaluation, we show that the verification cost of SPS is quite low: a matter of milliseconds of verification time on the client side and two additional binary searches per level, which are easily parallelizable, and whose cost is significantly less than the search processing cost. Finally, to give a comparison, the (unverified) SPS evaluation of [SPS14] gives a throughput of around 9000 per second for a 1000

times smaller database than the one taken for our estimation, while our evaluation shows a limitation (on the client side only) of 67 queries per second in the worst case. For databases with 2^{30} entries (the size of the largest database in [SPS14]), the verification time for 10 000 results lowers to 11 ms, at a throughput of 90 queries per second.