$\Sigma o\phi o\varsigma$ – Forward Secure Searchable Encryption

Raphael Bost
Direction Générale de l'Armement - Maîtrise de l'Information
Université de Rennes 1
raphael_bost@alumni.brown.edu

ABSTRACT

Searchable Symmetric Encryption aims at making possible searching over an encrypted database stored on an untrusted server while keeping privacy of both the queries and the data, by allowing some small controlled leakage to the server.

Recent work shows that dynamic schemes – in which the data is efficiently updatable – leaking some information on updated keywords are subject to devastating adaptative attacks breaking the privacy of the queries. The only way to thwart this attack is to design *forward private* schemes whose update procedure does not leak if a newly inserted element matches previous search queries.

This work proposes $\Sigma o\phi o\varsigma$ as a forward private SSE scheme with performance similar to existing less secure schemes, and that is conceptually simpler (and also more efficient) than previous forward private constructions. In particular, it only relies on trapdoor permutations and does not use an ORAM-like construction. We also explain why $\Sigma o\phi o\varsigma$ is an optimal point of the security/performance tradeoff for SSE.

Finally, an implementation and evaluation results demonstrate its practical efficiency.

CCS Concepts

•**Security and privacy** → **Privacy-preserving protocols; Security protocols; Management and querying of encrypted data;**

Keywords

Searchable Symmetric Encryption; Forward Privacy; Provable Security; Implementation

1. INTRODUCTION

Being able to efficiently and privately query an encrypted database stored on an untrusted server is critical for secure cloud applications. Ideally, we would like to use protocols leaking absolutely no information to the server. Such solutions can be constructed using powerful techniques such as multi-party computation, fully homomorphic encryption, or oblivious RAM, but, due to the genericity of these, they are impractical: they are slower than a naïve approach consisting in downloading the entire database and do the search locally [Nav15].

Searchable Symmetric Encryption (SSE) [SWP00], which is a class of structured encryption [CK10] for search structures such as search indexes or search trees, provides a solution to this problem by trading leakage for efficiency. This leakage comes from the use of (often symmetric) deterministic encryption that will enable the server to easily find matches between encrypted tokens, without having to run expensive protocols or computations.

In previous works, leakage ranges from fully revealing the keyword occurence pattern in each document (including, the keyword order), as of legacy-compatible constructions such as CipherCloud [Cip] or skyhigh [sky], to query-revealed occurence patterns, for reversed-index-based schemes derived from [CGKO06].

Islam *et al.* [IKK12] and Cash *et al.* [CGPR15] studied the real-world consequences of the leakage of SSE schemes and showed that even small leakage can be leveraged by a passive attacker to reveal the client's queries, leading to *leakage-abuse attacks*. For larger leakage, the authors of [CGPR15] show that full plaintext recovery of the encrypted database is possible.

Their work was improved by the one of Zhang *et al.* in the case of dynamic databases, where the attacker can inject new documents in the database, in [ZKP16]. In this paper, the authors showed a devastating adaptive attack which reveals the content of a past query by inserting as few as 10 new documents. This attack can be run on almost all existing SSE schemes because the server can learn that the newly added document matches a previous search query.

This work underlines the need for SSE constructions which do no leak this information, also called *forward private* constructions [SPS14]. To our knowledge, besides the ORAM-based ones such as TWORAM [GMP15], the only existing forward private schemes are by Chang and Mitzenmacher [CM05], and by Stefanov *et al.* [SPS14], but both suffer from inefficiencies. The former has a large bandwith requirement from the client to the server for search queries, and a very large server storage, while the latter uses non trivial ORAM-related techniques, which induces a large bandwith overhead on updates, despite supporting efficient deletions.

Until now, no SSE construction offered both the security requirements necessary to thwart those leakage-abuse attacks, and achieved optimal update efficiency.

CCS'16, October 24 - 28, 2016, Vienna, Austria

ACM ISBN 978-1-4503-4139-4/16/10. . . $15.00

DOI: http://dx.doi.org/10.1145/2976749.2978303

Scheme	Computation		Communication		Client	Forward	M.A.
	Search	Update	Search	Update	Storage	Private	
Previous works							
[CJJ+14]	$O(a_w)$	$O(1)$	$O(n_w)$	$O(1)$	$O(1)$	✗	✗
[BFP16]	$O(a_w + \log W)$	$O(\log W)$	$O(n_w + \log W)$	$O(\log W)$	$O(1)$	✗	✓
[SPS14]	$O\left(\min\begin{cases} a_w + \log N \\ n_w \log^3 N \end{cases}\right)$	$O(\log^2 N)$	$O(n_w + \log N)$	$O(\log N)$	$O(N^\alpha)$	✓	✗
[SPS14] ([BFP16])	$O\left(\min\begin{cases} a_w + \log^2 N \\ n_w \log^3 N \end{cases}\right)$	$O(\log^2 N)$	$O(n_w + \log N)$	$O(\log N)$	$O(N^\alpha)$	✓	✓[1]
[GMP15]	$\widetilde{O}(a_w \log N + \log^3 N)$	$\widetilde{O}(\log^2 N)$	$\widetilde{O}\left(\begin{array}{c} a_w \log N \\ + \log^3 N \end{array}\right)$	$\widetilde{O}(\log^3 N)$	$O(1)$	✓	✗
This work							
$\Sigma o\phi o\varsigma$	$O(a_w)$	$O(1)$	$O(n_w)$	$O(1)$	$O(W \log D)$	✓	✗
$\Sigma o\phi o\varsigma$-ε	$O(a_w)$	$O(1)$	$O(n_w)$	$O(1)$	$O(W(\log D + \lambda))$	✓	✓

Table 1 – Comparison with existing SSE schemes. N is the number of entries (*i.e.* the number of keyword/document pairs) in the database, while W is the number of distinct keywords, and D the number of documents. n_w is the size of the search result set for keyword w, and a_w is the number of times the queried keyword w was historically *added* to the database. In particular, in all works except [SPS14], deletions are not optimally supported: the search is not linear in the number of matching documents, but in the number of inserted documents matching the query. M.A. stands for 'Malicious Adversary'. We omitted the polynomial dependency in the security parameter λ for both computation and communication complexity. The $\widetilde{O}$ notation hides the $\log \log N$ factors. Update complexities are given per updated document/keyword pair. We only considered schemes whose server's storage complexity is optimal ($O(N)$).

Contributions. We construct a forward private SSE scheme with optimal search and update complexity, for both computational and communicational complexity, called $\Sigma o\phi o\varsigma$.[2]

It has both the security guarantees of forward private constructions [SPS14], and the asymptotic efficiency of constructions with more leakage [CJJ+14]. Our scheme uses simple cryptographic tools (only pseudo-random functions and trapdoor permutations, and does not rely on ORAM ideas), and is very easy to understand. A comparison of $\Sigma o\phi o\varsigma$ with previous works is given in Table 1.

We give a full proof of security against honest-but-curious adversaries, and show how it can be easily extended to security against malicious adversaries, without server-side modifications of the construction ($\Sigma o\phi o\varsigma$-ε).

Finally, we provide an open source implementation of our scheme, and show that, even on persistent storage, it is very efficient in practice, for both searches and updates.

2. RELATED WORK

SSE has been introduced by Song *et al.* [SWP00], with a solution whose search time is linear in the size of the database. Chang and Mitzenmacher [CM05] later proposed a construction whose search time is also linear, but that achieves forward privacy.

Ten years ago, Curtmola *et al.* [CGKO06] were the first to formally consider leakage, and designed, in the static setting, the first index-based SSE construction achieving sublinear search complexity. The first dynamic and sublinear scheme was introduced by Kamara *et al.* [KPR12], but was leaking the hashes of keywords contained by the updated documents. Kamara and Papamanthou [KP13] later improved this construction, and reduced the leakage at the cost of increased space complexity on the server. Still, this construction is not forward private.

another line of work presented index based schemes supporting boolean queries [CJJ+13], optimized for very large databases and dynamism [CJJ+14], or in a three-party setting [JJK+13], some of whose features were also studied in [PKV+14].

Forward privacy was for the first time explicitely considered by Stefanov *et al.* in [SPS14]. The authors built in their paper a dynamic, sublinear scheme achieving forward privacy.

Verifiable SSE schemes (*i.e.* secure against an active adversary) have been studied by Kurosawa and Ohtaki [KO12], and efficient and dynamic constructions have been recently proposed by Bost *et al.* [BFP16], including a forward private one derived from [SPS14].

Recently, Cash and Tessaro [CT14] demonstrated a lower bound on the storage locality of searchable encryption: it is not possible to achieve both optimal server storage and optimal locality. Their lower bound was shown tight by Asharov *et al.* [ANSS16].

SSE was extended by Chase and Kamara [CK10] to arbitrarily-structured data, such as graphs, matrices or labeled data. New applications of structured encryptions arose recently, such as approximate shortest distance queries on an encrypted graph [MKNK15].

SSE can be constructed from Oblivious RAM [GO96], either in a black box way, or using specific ORAM subcomponents, in a grey-box way, as in [GMP15]. Unfortunately, ORAM induces large bandwith overhead, large client storage and/or, multiple roundtrips. This makes use of ORAM for SSE unrealistic, as showed in the study by Naveed [Nav15].

[1] There is a flaw in the original paper [SPS14], concerning security against malicious adversaries, but it can be fixed without large modifications of the original construction (*cf.* [BFP16]).

[2] $\Sigma o\phi o\varsigma$ (pronounce 'sophos') stands for Scalable Optimal FOrward Secure Searchable Encryption. It also refers to the ancient greek for 'wise': we strongly believe that it is wise to use a forward private SSE scheme.

Islam *et al.* [IKK12] studied the consequences of leakage in searchable encryption. Their attacks were improved by Cash *et al.* [CGPR15], and lately by Zhang *et al.* [ZKP16], who demonstrated the need for forward privacy for SSE.

3. PRELIMINARIES

In the paper, λ is the security parameter and $\mathrm{negl}(\lambda)$ denotes a negligible function in the security parameter. Our construction uses pseudo-random functions (PRF) and (keyed) hash functions, for which we use the standard security definitions [Gol04].

Unless specified explicitely, the symmetric keys are strings of λ bits, and the key generation algorithm uniformly samples a key in $\{0,1\}^\lambda$. We only consider (possibly probabilistic) algorithms and protocols running in time polynomial in the security parameter λ. In particular, adversaries are probabilistic polynomial time algorithms.

For a finite set X, $x \overset{\$}{\leftarrow} X$ means that x was uniformly sampled from X.

3.1 Trapdoor Permutations

A trapdoor permutation (TDP) π is a permutation over a set $\mathcal{D}$ such that, using a public key PK, π can be easily evaluated, but the inverse π^{-1} can be efficiently computed only with the secret SK.

More formally, π with the key generation algorithm KeyGen is a trapdoor permutation if for every adversary $\mathcal{A}$

$$\mathbf{Adv}^{\mathsf{OW}}_{\pi,\mathcal{A}}(\lambda) \leq \mathrm{negl}(\lambda)$$

where

$$\mathbf{Adv}^{\mathsf{OW}}_{\pi,\mathcal{A}}(\lambda) = \Pr[y \overset{\$}{\leftarrow} \mathcal{M}, (\mathsf{SK},\mathsf{PK}) \leftarrow \mathsf{KeyGen}(1^\lambda),$$
$$x \leftarrow \mathcal{A}(1^\lambda, \mathsf{PK}, y) : \pi_{\mathsf{PK}}(x) = y]$$

(π is one-way) while for every $x \in \mathcal{D}$

$$\pi_{\mathsf{PK}}(\pi^{-1}_{\mathsf{SK}}(x)) = x \text{ and } \pi^{-1}_{\mathsf{SK}}(\pi_{\mathsf{PK}}(x)) = x$$

and $\pi_{\mathsf{PK}}(.)$ and $\pi^{-1}_{\mathsf{SK}}(.)$ can be computed in polynomial time. In the paper we also use the notation $\pi^{(c)}_{\mathsf{PK}}(x)$ (resp. $\pi^{(-c)}_{\mathsf{SK}}(x)$) for the iterated application of π_{PK} (resp. π^{-1}_{SK}) c times.

3.2 Symmetric Searchable Encryption

A *database* $\mathsf{DB} = (\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^D$ is a tuple of index/keyword-set pairs with $\mathsf{ind}_i \in \{0,1\}^\ell$ 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 D to be the number of documents in DB, $W = |\mathsf{W}|$ the total number of keywords , and $N = \sum_{i=1}^D |\mathsf{W}_i|$ to be the number of document/keyword pairs (we identify documents with their indices). We denote by $\mathsf{DB}(w)$ 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.
- $\mathsf{Search}(K, q, \sigma; \mathsf{EDB}) = (\mathsf{Search}_C(K, q, \sigma), \mathsf{Search}_S(\mathsf{EDB}))$ is a protocol between the client with input the key K, its state σ, and a search query q, and the server with input EDB. For single-keyword search schemes (as the ones considered in this paper), a search query is restricted to a unique 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 index ind and a set W of keywords, and the server with input EDB. The update operations are taken from the set $\{\mathsf{add}, \mathsf{del}\}$, meaning, respectively, the addition and the deletion of a document/keyword pair.

3.3 Security of SSE

SSE schemes must achieve two security properties: correctness and confidentiality.

Correctness. The correctness of an SSE scheme is a basic property: the search protocol must return the correct result for every query, except with negligible probability.

Confidentiality. The confidentiality definition of an SSE scheme uses the real world versus ideal world formalization [CGKO06, KPR12, CJJ+14]. It is parametrized by a *leakage function* $\mathcal{L} = (\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}})$ describing what the protocol leaks to the adversary, and formalized as a stateful algorithm. The definition ensures that the scheme does not reveal any information beyond the ones that can be inferred from the leakage function.

More precisely, we define two games, SSEReal, and SSEIdeal, in which the adversary A chooses a database DB, and is given back EDB generated using $\mathsf{Setup}(\mathsf{DB})$ in the real case, or $S(\mathcal{L}^{\mathsf{Stp}}(\mathsf{DB}))$ in the ideal case. Then, he repeatedly performs search and update queries with an input in and receives the transcripts generated running the $\mathsf{Search}(q)$ (resp. $\mathsf{Update}(\mathsf{op}, \mathsf{in})$) protocol in the real game, or the simulator $S(\mathcal{L}^{\mathsf{Srch}}(q))$ (resp. $S(\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, \mathsf{in}))$) in the ideal game. Eventually, A outputs a bit. We say that the scheme is $\mathcal{L}$-*adaptively-secure* if for all adversary A, there exists an efficient simulator S such that the adversary A cannot distinguish an execution of the real game from an execution of the ideal game with non-negligible probability.

Common Leakage. Many SSE schemes leak the repetition of tokens sent by the client to the server, and hence, the repetition of queried keywords. Similarly to previous works [CGKO06], when this leakage is limited to search queries (for example in static schemes), we call it the *search pattern*. When repetition of updated keywords also leaks, we call it the *query pattern*.

More formally, the leakage function $\mathcal{L}$ will keep as state the *query list* Q: the list of all queries issued so far, and whose entries are (i, w) for a search query on keyword w, or $(i, \mathsf{op}, \mathsf{in})$ for an op update query with input in. The integer i is a timestamp, initially set to 0, and which is incremented at each query. The search and query patterns, respectively denoted $\mathsf{sp}(x)$ and $\mathsf{qp}(x)$, are then defined as

$$\mathsf{sp}(x) = \{j : (j, x) \in Q\} \text{ (only matches search queries)}$$
$$\mathsf{qp}(x) = \{j : (j, x) \in Q \text{ or } (j, \mathsf{op}, \mathsf{in}) \in Q \text{ and } x \text{ appears in } \mathsf{in}\}$$

In the paper, we also introduce the notation $\mathsf{HistDB}(w)$: it is the list of documents historically added to DB matching keyword w, in the order of insertion. In particular, it includes documents that have been added and later deleted, or even documents that have been added twice.

Finally, we use the notation $\mathsf{Hist}(w)$, the *history* of keyword w. It lists all the modifications made to $\mathsf{DB}(w)$ over the time. It consists in a tuple whose first element is $\mathsf{DB}_0(w)$, the set of documents indices matching w at setup, and whose second element is the list $\mathsf{UpHist}(w)$ of updates of documents matching w, called the *update history*. For example, consider two documents D_{ind_1} and D_{ind_2} matching w. Suppose that we first added D_{ind_1} at the second update, then D_{ind_2} at the 5-th update, and finally that D_{ind_1} got removed during the 14-th update. Then $\mathsf{UpHist}(w) = [(2, \mathsf{add}, \mathsf{ind}_1), (5, \mathsf{add}, \mathsf{ind}_2), (14, \mathsf{del}, \mathsf{ind}_1)]$. These can be also formally defined using the query list.

4. FORWARD PRIVACY

4.1 Definition

Forward privacy is a strong property of the SSE leakage function of dynamic SSE schemes. Informally, it means that an update does not leak any information about the updated keywords. In particular, the server cannot learn that the updated document matches a keyword we previously queried. More formally, we give the following definition of forward privacy.

Definition 1. A $\mathcal{L}$-adaptively-secure SSE scheme Σ is forward private if the update leakage function $\mathcal{L}^{\mathsf{Updt}}$ can be written as

$$\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, \mathsf{in}) = \mathcal{L}'(\mathsf{op}, \{(\mathsf{ind}_i, \mu_i)\})$$

where $\{(\mathsf{ind}_i, \mu_i)\}$ is the set of modified documents paired with the number μ_i of modified keywords for the updated document ind_i.

The definition given here slightly extends the informal definition of forward privacy of Stefanov *et al.* [SPS14] which focused only on *added* documents rather than on updated (*i.e.* both added and deleted) documents. They also describe *backward privacy*, a property that prevents searches to be performed over deleted documents.

The only existing schemes supporting both forward and backward privacy are based on Oblivious RAM, but are not particularly efficient, while efficient forward private-only schemes have been designed [CM05, SPS14], but suffer from a big bandwith or storage blowup.

4.2 The Need for Forward Privacy

Islam *et al.* [IKK12] and Cash *et al.* [CGPR15] studied the real-world consequences of the leakage of SSE schemes and showed that even small leakage can be leveraged by an attacker to reveal the client's queries. Their work not only considered static databases but also dynamic databases, in a setting where the server could submit new documents to be added in the database in a way that would help him to break query privacy. Zhang *et al.* [ZKP16] improved this *file injection attack* and showed that it can be devastating.

They actually describe both a non-adaptive and an adaptive attack. The adaptive attack is very effective but only applies to schemes that are *not* forward private. This attack, reveals a previously search keyword w by submitting $\log 2T$ new documents if the adversary has partial knowledge of the database, or using $W/T + \log T$ new documents if he hasn't. T is the *threshold parameter*, a public parameter used to avoid non adaptative attacks, and that needs to be kept small to be an efficient counter measure (but not too small, to avoid a large efficiency loss). In [ZKP16], the authors chose $T = 200$ for their experiments, and in this case, the privacy of a query can be broken by an adversary posting less than 10 new documents.

Even on a functional point of view, forward privacy is important. Indeed, a forward private scheme allows for an online build of the encrypted database. In most other SSE constructions, one must first go through an indexing step: the setup phase needs an inverted index whose construction takes both time and space. Forward private schemes avoid this source of inefficiency.

4.3 Constraints Induced by Forward Privacy

Unfortunately, forward privacy has various downsides in terms of efficiency. This section explains that dynamic SSE schemes have to make compromises between storage efficiency, locality, and security.

On Storage. First, let us focus on storage. A desirable property of dynamic databases is space reclamation upon entry deletion. For searchable encryption, it implies that, when a document/keyword pair is removed from the encrypted index, the logical location of the pair can be marked as empty. Now, suppose an adversary successively submits a search query for a keyword w, and then a delete query for pair (ind, w). If the Search protocol does not modify the encrypted database (almost all existing schemes do not), and if the scheme reclaims space, the adversary will learn that the updated keyword was just searched. The previous example actually shows that the locations of the encrypted pairs before and after the search query must be completely unrelated for forward private, space-reclaiming scheme, yielding constructions with security properties close to the ones of resizable ORAM [MMBC15]. In particular, one cannot really hope for extremely efficient scheme with these properties.

On Locality. In [CT14], Cash and Tessaro studied the problem of memory accesses locality in SSE. In particular, they showed that one cannot achieve even minimal locality without increasing the size of the encrypted database beyond $\Theta(N)$. Their lower bound holds for static schemes, and so applies for dynamic schemes too, but, to our knowledge, locality of dynamic SSE has not been specifically studied.

Here we claim that, for dynamic schemes, locality and forward-privacy are two irreconciliable notions. First, it is worth noticing that caring about memory locality makes sense only if the scheme is already efficient in terms of disk accesses. For example, if a scheme systematically updates the encrypted database every time a search is performed (*e.g.* the ORAM-based scheme [GMP15]), the rewriting cost will be much higher than the efficiency improvements due to increased locality. So without loss of generality, we consider schemes whose Search protocol does not modify a large portion of EDB. In this case, forward privacy implies that, for an updated keyword w, the location of the newly inserted tokens is unrelated to the location of already existing tokens matching w. So, if the whole set $\mathsf{DB}(w)$ is not somehow, at least partially, re-written during the update, no locality op-

$$\begin{tikzcd}
ST_0(w) \arrow[r, bend left, "\pi_{\mathsf{SK}}^{-1}"] \arrow[dd, "H_{K_w}"'] & ST_1(w) \arrow[l, "\pi_{\mathsf{PK}}"] \arrow[dd, "H_{K_w}"'] & & ST_c(w) \arrow[ll, dashed, "\pi_{\mathsf{PK}}^{(c-1)}"] \arrow[r, bend left, "\pi_{\mathsf{SK}}^{-1}"] \arrow[dd, "H_{K_w}"'] & ST_{c+1}(w) \arrow[l, "\pi_{\mathsf{PK}}"] \arrow[dd, "H_{K_w}"'] \\
 & & \cdots & & \\
UT_0(w) & UT_1(w) & \cdots & UT_c(w) & UT_{c+1}(w)
\end{tikzcd}$$

Figure 1 – Relations among tokens. Operations in red can only be done by the client, using the secret key SK.

timization is possible for keyword w. Said otherwise, if you want both forward privacy and locality during search, you have to do some rather large modifications of the encrypted database, either during searches or updates.

Finally, we will in see in the evaluation of our implementation that non-locality is not a critical performance issue with modern SSDs.

5. THE $\Sigma o\phi o\varsigma$ CONSTRUCTION

In this section, we construct a forward secure SSE scheme, whose Search and Update protocols are performed in a single roundtrip, but at the cost of $O(W(\log D + \log|\mathcal{M}|)$ storage on the client side. We also first consider a scheme that only supports additions, not deletions. We will then describe how to turn this basic construction into an SSE constructions supporting both additions and deletions, with reduced client storage, and also how to make it secure against malicious adversaries.

5.1 General ideas

In an inverted index scheme (such as [CGKO06] and derived works), we are usually considering for each keyword w an indexed list of matching documents $(\mathsf{ind}_0, \cdots, \mathsf{ind}_{n_w})$. Every element ind_c of this list is then encrypted and stored at a (logical) location derived from w and c. We call this location $UT_c(w)$. When the client wants to add a document matching w, he computes a new location $UT_{n_w+1}(w)$, encrypts the document index as e, and sends $(UT_{n_w+1}(w), e)$ to the server (this explains our notation UT, for update token).

When the client performs a search query on w, he will issue a search token that will allow the server to recompute the update tokens, and hence the locations of the entries matching w. In general, we want the update tokens for a given w to be unlinkable until a search token $ST(w)$ is issued. In our case, the search token generated by the client will depend on the number n_w of matching entries, and we want that the search token $ST_c(w)$ generated for c results to be unlinkable to the update tokens $UT_i(w)$ for $i > c$, *i.e.* the update tokens that will be issued for keyword w in the future. In particular, it implies that the server could not generate $ST_i(w)$ from $ST_c(w)$ when $i > c$.

To do so, we could make the client generate all the search token using a PRF evaluation $F(w, i)$, and send them to the server. However this solution is not satisfactory: the client needs to send $O(n_w)$ tokens to the server, which can be a problem on constrained devices. In this work, we propose another solution, based on trapdoor permutations: from $ST_i(w)$, the server will be able to compute $ST_{i-1}(w)$ using the a public key, but only the client will be able to construct $ST_{i+1}(w)$.

Figure 1 gives the relations among the tokens and formalizes our idea for token generation. $ST_{i+1}(w)$ is generated from $ST_i(w)$ by applying the inverse of a one-way trapdoor permutation π: only the client will be able to perform this operation, while the server, given the public key PK will do the opposite, namely compute from a search token $ST_c(w)$ all the tokens $ST_i(w)$ for $0 \leq i < c$. Finally, the update tokens are derived from the search tokens, using a keyed hash function. In particular, it is crucial that H is pre-image resistant for the security of the scheme. We will actually show the security of this construction when H is modeled by a random oracle.

5.2 Basic Construction

Algorithm 1 gives the formal description of our basic forward private scheme, $\Sigma o\phi o\varsigma$-B. It follows the idea of the previous section, in particular for token generation. Also, the only updates $\Sigma o\phi o\varsigma$-B supports are insertions of new keyword/document pairs.

In the pseudo code, π is a trapdoor permutation, F is a PRF, H_1 and H_2 are keyed hash functions, whose outputs are, respectively, μ and ℓ bits long. On the client side, $\mathbf{W}$ maps every inserted keyword to its current search token $ST_c(w)$ and to a counter $c = n_w - 1$. Every time a new document matching w is inserted, $\mathbf{W}[w]$ gets 'incremented': the client generates the new search token $ST_{c+1}(w) = \pi^{-1}(ST_c(w))$ and stores it in $\mathbf{W}$. If w did not match any documents, a new $ST_0(w)$ is randomly picked and put in $\mathbf{W}$. Finally, the entries locations, *i.e.* the update tokens, are derived from the search tokens by a keyed hash function.

$\Sigma o\phi o\varsigma$' setup algorithm does not take a database as input: as stated in Section 4.2, encryption can be performed online with a forward private scheme, without loss of security.

Correctness. The correctness of $\Sigma o\phi o\varsigma$-B is quite straightforward. The only issue is collision among the update tokens $UT_c(w)$, generated from H_1 with input (K_w, ST_c). We can reduce the correctness to the collision resistance of H_1. In particular, we need to choose μ such that $N^2/2^\mu$ is negligible in the security parameter, so in practice, we will set $\mu = \lambda + 2\log N_{\max}$, where $N_{\max}$ is the maximum number of pairs the database can support.

Complexity. The scheme's computational complexity is optimal: $O(n_w)$ for a search query, $O(1)$ for an update. Both Search and Update are single round, and their performance will not be more affected by network latency than regular

Algorithm 1 $\Sigma o \phi o \varsigma$-B: Forward private SSE scheme with large client storage.

```
Setup()
$K_S \xleftarrow{\$} \{0,1\}^\lambda$
$(\mathsf{SK}, \mathsf{PK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$\mathbf{W}, \mathbf{T} \leftarrow$ empty map
return $((\mathbf{T}, \mathsf{PK}), (\mathbf{K_S}, \mathsf{SK}), \mathbf{W})$

Search($w, \sigma$; EDB)
    Client:
$K_w \leftarrow F_{K_S}(w)$
$(ST_c, c) \leftarrow \mathbf{W}[w]$                          ▷ $c = n_w - 1$
if $(ST_c, c) = \perp$
    return $\emptyset$
Send $(K_w, ST_c, c)$ to the server.
    Server:
for $i = c$ to $0$ do
    $UT_i \leftarrow H_1(K_w, ST_i)$
    $e \leftarrow \mathbf{T}[UT_i]$
    $\mathsf{ind} \leftarrow e \oplus H_2(K_w, ST_i)$
    Output each $\mathsf{ind}$
    $ST_{i-1} \leftarrow \pi_{\mathsf{PK}}(ST_i)$
end for

Update(add, $w$, ind, $\sigma$; EDB)
    Client:
$K_w \leftarrow F(K_S, w)$
$(ST_c, c) \leftarrow \mathbf{W}[w]$
if $(ST_c, c) = \perp$ then
    $ST_0 \xleftarrow{\$} \mathcal{M}$, $c \leftarrow -1$
else
    $ST_{c+1} \leftarrow \pi_{\mathsf{SK}}^{-1}(ST_c)$
end if
$\mathbf{W}[w] \leftarrow (ST_{c+1}, c+1)$
$UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
$e \leftarrow \mathsf{ind} \oplus H_2(K_w, ST_{c+1})$
Send $(UT_{c+1}, e)$ to the server.
    Server:
$\mathbf{T}[UT_{c+1}] \leftarrow e$
```

insecure protocols.

Bandwith is also (almost) optimal. The Search protocol uses a single $\log |\mathcal{M}| = \text{poly}(\lambda)$ bits token per query. The Update protocol sends a $\mu + \ell$ bits token per updated document/keyword pair, representing a $\lambda + \log N_{\max}$ bits increase compared to the smallest possible update token size of an unencrypted database.

The client's storage is $O(W(\log |\mathcal{M}| + \log D))$: an element of $\mathcal{M}$ is stored for every keyword, with the counter $c < D$.

5.3 Security

The adaptive security of $\Sigma o \phi o \varsigma$-B can be proven in the Random Oracle Model, and relies on the one-wayness of the TDP π and on the pseudo-randomness of F. We will only give a sketch of the proof here, and postpone to Appendix A the full proof.

THEOREM 1 (ADAPTIVE SECURITY OF $\Sigma o \phi o \varsigma$-B). *Define* $\mathcal{L}_\Sigma = (\mathcal{L}_\Sigma^{\mathsf{Srch}}, \mathcal{L}_\Sigma^{\mathsf{Updt}})$ *as,*

$$\mathcal{L}_\Sigma^{\mathsf{Srch}}(w) = (\mathsf{sp}(w), \mathsf{Hist}(w))$$

$$\mathcal{L}_\Sigma^{\mathsf{Updt}}(\mathsf{add}, w, \mathsf{ind}) = \perp.$$

$\Sigma o \phi o \varsigma$-B is $\mathcal{L}_\Sigma$-adaptively-secure.

Hence, $\Sigma o \phi o \varsigma$ thwarts all the devastating adaptive file-injection attacks of [ZKP16] – but not the non-adaptive ones.

The proof works by constructing successive indistinguishable hybrids, where H_1 and H_2 are modeled as random oracles, the first hybrid being the real-world game, and the laste the ideal-world game. We first replace F by a random function, *i.e.* we randomly pick the keys K_w.

In the second hybrid, we will replace all the strings generated by the random oracles in the Update protocol by randomly chosen strings. The game will then program the random oracles during the Search protocol so that the result produced by the server matches the real results: H_1 is set to map the i-th search token for w to the update token produced randomly when w was updated for the i-th time. H_2 is programmed in a similar manner to produce the right keystream used to hide the ind values. We show that if the first and second hybrids are *not* indistinguishable, it means that the adversary was able to invert the trapdoor permutation without knowing the secret key.

Finally, we construct an hybrid that only needs to know the repetition of search queries and the history to produce search tokens indistinguishable from the previous hybrid. As this hybrid only needs the output of the leakage function to run, it means that we have a simulator that produces indistinguishable transcripts from the real security game.

Note that $\mathcal{L}_\Sigma$ uses Hist and not DB because the simulator needs to know exactly when documents matching w were inserted in the database in order to correctly simulate the real protocol. In Appendix, Theorem 4 gives a more formal statement, with an explicit distinguishing advantage.

5.4 Derived Constructions

Deletion Support. Although $\Sigma o \phi o \varsigma$-B does not support deletions, this is easy to fix by 'duplicating' the data structure: we will use one instance of $\Sigma o \phi o \varsigma$-B for insertions, and the other for deletions. When searching w, the server will compute and return the difference between the indices matching w in both instances.

The leakage stays the same: we can separate the elements of $\mathsf{HistDB}(w)$ in two sublists $\mathsf{HistDB}_{\mathsf{add}}(w)$ and $\mathsf{HistDB}_{\mathsf{del}}(w)$, according to their operation (add or del), to build the leakage functions of each instance of $\Sigma o \phi o \varsigma$-B. The only difference with the original scheme would be that it leaks the operation $\mathsf{op} = \mathsf{add}$ or del, but we can use the same map $\mathbf{T}$ for both instances, which would hide the actual operation performed during the update.

Batch Updates. We can also slightly modify the update protocol to support batch updates on documents: when we want to update document with index ind on the keywords list $\mathbf{w}$, we successively select keywords in $\mathbf{w}$ in random order, and run the original Update protocol with input ind and the selected keyword.

Again, the leakage function remains identical. We just have to slightly update the security proof: when the simulator programs the random oracles on (K_w, ST_i), instead of targeting exactly the only token produced at the i-th update on w, it will pick a one of the random tokens produced during the update that modified $\mathsf{DB}(w)$ for the i-th time.

5.5 Reducing Client-side Storage

We saw that the client's storage is $O(W(\log |\mathcal{M}| + \log D))$.

This can be a problem on constrained devices, especially when $\mathcal{M}$ is big, which is the case for both RSA and Rabin's Squaring trapdoor permutations: $\mathcal{M} = \mathbb{Z}_N^*$, where N is, for a reasonable level of security, a 2048 bits integer.

But there is a workaround to reduce storage to $O(W \cdot \log D)$ at the expense of additional computations. The idea is to pseudo-randomly generate $ST_0(w)$ from w (or a unique identifier $i_w \in \mathbb{N}$ of w). When $\mathcal{M} = \mathbb{Z}_N^*$, this is quite easy to do from a PRF $G : \mathbb{N} \to \{0,1\}^{\lambda+\log N}$ by taking $ST_0(w) \leftarrow G(i_w) \bmod N$.

When $ST_c(w)$ is needed, we recompute it from $ST_0(w)$. However, this will be very computationally expensive if have to iteratively compute $\pi_{\mathsf{SK}}^{(-c)}$ by iterating π_{SK}^{-1} c times. Fortunately, this is not the case for common trapdoor permutations π, and in particular for RSA: if (p, q, d) and (N, e) are respectively the secret and the public keys, $y = RSA_{\mathsf{SK}}^{(-i)}(x)$ can be computed as follows.

$$f \leftarrow d^i \bmod (p-1)(q-1)$$
$$y \leftarrow x^f \bmod N$$

We can also use the Chinese remainders technique to improve the performances of this computation.

Then, the client has only to store an identifier i_w for every w, together with the counter c. When he needs to, he can easily recompute the $ST_c(w)$ from i_w and c, with a small complexity overhead (essentially the cost of a private-key operation, when RSA is the chosen TDP). The client's storage reduces to $O(W \log D)$ asymptotically (as $c < D$).

We call this version $\Sigma o\phi o\varsigma$ (without the -B), and it is the one that we implemented. The proof of security of $\Sigma o\phi o\varsigma$-B can easily be ported to $\Sigma o\phi o\varsigma$:

Theorem 2. *$\Sigma o\phi o\varsigma$ is $\mathcal{L}_\Sigma$-adaptively-secure.*

5.6 Security against Malicious Adversaries

Theorem 2 states security against passive adversaries only. However, $\Sigma o\phi o\varsigma$ can easily be turned in a verifiable SSE scheme, as defined by Bost *et al.* [BFP16]. We can do so by storing a hash of $\mathsf{DB}(w)$ for every w on the client, and during a search, the stored hash value is checked to match the hash recomputed from the server's results. To allow for easy incremental updates, we can use a set hashing function to hash $\mathsf{DB}(w)$, as explained in [BFP16]. The asymptotic complexity of this verifiable version of $\Sigma o\phi o\varsigma$ – that we call $\Sigma o\phi o\varsigma$-ε[3] – and the client's storage will increase to $O(W(\log D + \lambda))$ From the collision resistance of the set hash function, we can show that $\Sigma o\phi o\varsigma$-ε is secure against malicious adversaries.

Theorem 3. *$\Sigma o\phi o\varsigma$-ε is $\mathcal{L}_\Sigma$-adaptively-secure against malicious adversaries.*

5.7 Comparison with Other Constructions

If we limit this scheme to only support modification of full documents (*i.e.* it is not allowed to add/delete a keyword to a document already in the database, but we can add or delete an entire document), our scheme has the same functionalities than the SPS scheme [SPS14], whose leakage function is $\mathcal{L}_{\mathsf{SPS}}$, with $\mathcal{L}_{\mathsf{SPS}}^{\mathsf{Srch}}(w) = (\mathsf{sp}(w), \mathsf{DB}(w), \mathsf{HistDB}(w))$, and $\mathcal{L}_{\mathsf{SPS}}^{\mathsf{Updt}}(\mathsf{op}, \mathsf{ind}, \mathbf{w}) = (\mathsf{op}, \mathsf{ind}, |\mathbf{w}|)$. $\mathcal{L}_\Sigma$ and $\mathcal{L}_{\mathsf{SPS}}$ might look incomparable, but actually we can construct $\mathcal{L}_\Sigma$ from $\mathcal{L}_{\mathsf{SPS}}$ rather easily: we just reconstruct $\mathsf{Hist}(w)$ from $\mathsf{HistDB}(w)$ and the timestamps t generated at every update. We associate these timestamps to the leaked information $(\mathsf{op}, \mathsf{ind}, |\mathbf{w}|)$, and for each matching document in $\mathsf{HistDB}(w)$, we can retrieve the associated update operation, and when it happened, and hence recompute $\mathsf{Hist}(w)$.

This shows that $\Sigma o\phi o\varsigma$ is $\mathcal{L}_{\mathsf{SPS}}$-adaptively-secure. We can similarly show that SPS is $\mathcal{L}_\Sigma$-adaptively-secure by adapting the proof of [SPS14]. Hence, SPS and $\Sigma o\phi o\varsigma$ have exactly the same security guarantees. However $\Sigma o\phi o\varsigma$ is much more efficient in terms of bandwith usage and update complexity. Namely, SPS's updates trigger $O(\log^2 N)$ work and $O(\log N)$ bandwith usage per update, using standard, yet not trivial de-amortization techniques. $\Sigma o\phi o\varsigma$ bandwith and computational overhead for an update is constant.

Also, SPS needs $O(N^a)$ client memory for $0 < a < 1$ as working storage to run the oblivious sort algorithm needed during the updates, while $\Sigma o\phi o\varsigma$ stores $O(W \log D)$ on the client – and we expect $W \ll N$.

As presented in Table 1, $\Sigma o\phi o\varsigma$ has the same asymptotic complexity as the most efficient dynamic SSE schemes. Moreover, we saw in Section 4.3 that we cannot really hope for something better in terms of features (space reclamation) or storage locality, for a forward private scheme. This justifies the optimality claim made higher, in the abstract, and in $\Sigma o\phi o\varsigma$' name itself.

6. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We implemented $\Sigma o\phi o\varsigma$ in C/C++, using a bit more than 8000 lines of code. For the trapdoor permutation, we used RSA. Our RSA implementation uses OpenSSL's BigNum library. The PRF F and the keyed hash function H are instantiated using HMAC, and we chose Blake2b as the underlying hash function.

Aside from cryptographic components, gRPC [Goo16] was chosen for the RPC machinery, and the map $\mathbf{T}$ were stored using RocksDB [Fac16].

$\mathbf{W}$ is implemented as a memory-mapped disk-resident – and hence persistent – hash table (with support of multiple collisions in buckets, and an overflow stash to improve the data structure's load).

We ran our experiments on a desktop computer with a single Intel Core i7 4790K 4.00GHz CPU (with 8 logical cores), 16GB of RAM, a 250 GB Samsung 850 EVO SSD, running on OS X.10. Our code is Open Source [Bos16].

Parameters. Cryptographic keys are 128 bits long for symmetric primitives, and we chose to use 2048 bits RSA keys, with a public exponent fixed to 3.

Our system can easily be scaled to support databases with up to 2^{64} keyword/document pairs, but to avoid unnecessary overhead for our experiments using much smaller database, we fixed $N \le 2^{48}$, $W \le 2^{48}$, and the maximum matching documents per keyword $n_{\max} \le 2^{32}$. We set μ, the length of the update tokens, to 128 bits, leading to an incorrectness probability of 2^{-32} for the maximum size database. Similarly, on the client side, the keywords are hashed to obtain the 128 bits identifiers i_w (*cf.* Section 5.5).

6.1 Evaluation

We evaluated the performance of $\Sigma o\phi o\varsigma$ using 4 data sets of increasing size and also the English Wikipedia. The char-

[3] ε stands for $\varepsilon\pi\alpha\lambda\eta\theta\varepsilon\upsilon\omega$: I verify.

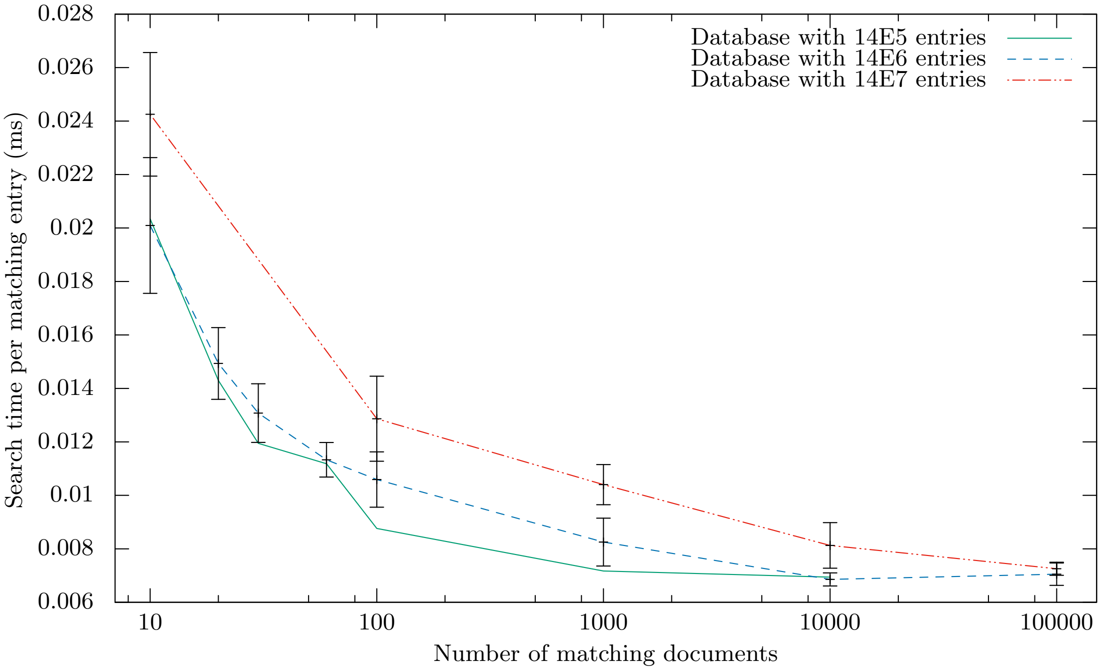

Figure 2 – $\Sigma o\phi o\varsigma$ search time per matched document (with standard deviation). Timings do not include the RPC.

		Storage size	
W	N	Client (load)	EDB
23,356	1.4E6	572 kB (0.91)	64.0 MB
213,349	14E6	4.81 MB (1.00)	512 MB
2,113,333	140E6	64.2 MB (0.64)	5.25 GB
4,599,490	139E6	128 MB (0.91)	5.25 GB

Table 2 – Evaluation databases size. The last database is the English Wikipedia.

acteristics of the databases are given in Table 2, as well as the size of the encrypted database, and the size of the client's state (*i.e.* the size of **W**).

We also report the load of the hash table storing **W**. It can be optimized if we know in advance the number of distinct keywords in the database. This is what we did for the first and second database (but not the third).

Search. Figure 2 presents our measurements for the search algorithm, on the server side, without accounting for RPC costs – we focus on the core of the algorithm. The timings given here are means of the elapsed search time per found document, taken over between 1000 queries (for small queries and large queries on large datasets) to 30 queries (for large queries on small data sets). This allows us to exhibit some real-world side effects in our implementation. We also give standard deviations for data sets of 14E6 and 14E7 keyword/document pairs.

The first thing we can notice in the timings, is that the larger the result set, the faster the search algorithm (again, on a per matching document basis). We explain that by the cost of multi-threading, and by the storage latency: even if the RSA or hash function computations are fully parallelized, adding an index to the result list is not, and accessing the disk induces some wait. Hence, at the beginning of the search algorithm, RSA operations will not be fully interleaved with disk accesses (like they are for a sufficiently large result set), we will pay for the latency induced by mutexes and storage accesses. Also, in the case of small result sets, some one-time costly operations (such as creating threads) are not amortized.

Figure 3 includes the RPCs in the timings, shows that they are far from being negligible, and also confirms our previous supposition: sending a matching document index to the client *via* RPC cannot be done concurrently for a given query, and creates a bottleneck if RPC, disk accesses and RSA computations are not well interleaved.

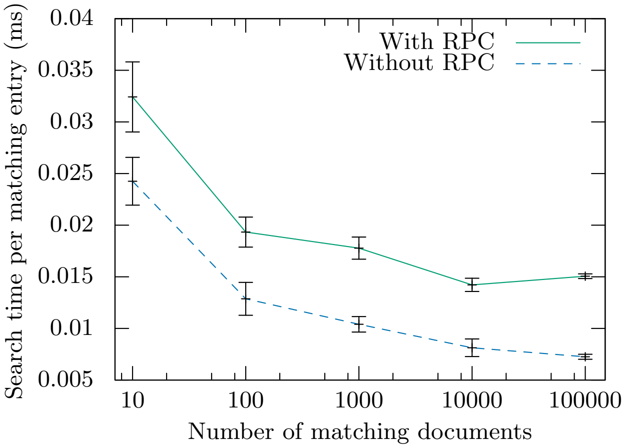

Figure 3 – $\Sigma o\phi o\varsigma$ search time per matched document, including RPC costs.

Update. We report an update throughput of around 4300 keyword/document pairs per second, for both large and small datasets. This evaluation includes all costs (disk accesses, RPC). For updates, the bottleneck is clearly the computation of the new search token used to derive the update token. In particular, we pay for the fact that private key operations are a lot more expensive than public key operations (to a factor 30 on our computer).

6.2 Comparison with Existing Implementations

We can try to compare our implementation with the ones in [CJJ+13], [CJJ+14], and [SPS14], using figures taken directly from the original papers. The comparison will be mostly qualitative as the performance are despite that the evaluation of these papers are measured in extremely different environments. The implementations in [CJJ+13] and [CJJ+14] run on 8 cores server blades connected to a high-end RAID-0/RAID-5 HDD-based storage, while [SPS14] performance is evaluated on an Amazon EC2 `cr1.8xlarge` instance, with 32 cores, entirely in RAM. Ours runs on a desktop computer, with a medium-end SSD.

The first thing we can notice, is that, despite the use of costly asymmetric cryptographic primitives, we achieve higher search throughput with a single basic SSD than the authors of [CJJ+13] (1E7 matches per second for $\Sigma o\phi o\varsigma$, versus 2E4 for large results queries), who use an expensive SAN system. Even using grouping techniques the performances are comparable to ours (*cf.* [CJJ+14] whose search algorithm needs approximately 150 ms for a 1E4 documents match, our implementation only needs 1ms), justifying our claim of Section 4.3, that locality in SSE is not necessary for good performances.

This also quite invalidates the reflexion of Stefanov*et al.* in [SPS14] about the relative cost of RAM vs. Disk: using a single PCI Express SSD, we can achieve up to 430,000 random reads per seconds [Int] with very low latency, and this can be further improved using several SSDs in RAID. For $\Sigma o\phi o\varsigma$, we claim that, once implemented on such powerful systems (more CPU cores, higher end storage), search performance will increase accordingly.

However, this cannot really be done for the updates as we would like (computationally) weak clients to also perform well when updating the database. This would require an improvement of our scheme, so as to limit the number of RSA public key operations the client has to perform during updates. We leave this for future work.

Given that disk accesses is a bottleneck of our construction – as well as for the [CJJ+14] construction – it is quite hard to compare $\Sigma o\phi o\varsigma$ with SPS whose encrypted database is RAM-resident. We can only conjecture that, once implemented on non-volatile storage, SPS throughput, both for search and updates, will be comparable to ours. One could however argue that, in presence of deletions, SPS's sublinear search algorithm will imply better performance than $\Sigma o\phi o\varsigma$ linear one, but that would be the case only if the non-deleted documents represent a very small fraction of the overall inserted documents: $\Sigma o\phi o\varsigma$ makes exactly $2a_w - n_w$ storage accesses, while SPS makes at the very least $\frac{\log^2 N}{2}$ accesses per non deleted matching document – this being a conservative lower bound. As storage accesses are the practical bottleneck for both schemes, SPS will perform better only if less than 0.64% of the inserted documents have not been deleted if $N > 2^{25}$. Also, as SPS's search uses binary searches, all theses accesses cannot be fully parallelized unlike in $\Sigma o\phi o\varsigma$.

Finally, another nice feature of our scheme is that $\Sigma o\phi o\varsigma$' server storage is 8 times smaller than the one in [SPS14], and 2.5 times than the scheme of [CJJ+14], reducing the financial cost of SSE.

Acknowledgements

The author thanks Pierre-Alain Fouque and David Pointcheval for their help on clarifying the proof of security, and Brice Minaud and anonymous reviewers for helpful comments.

This work was supported in part by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013 Grant Agreement no. 339563 – CryptoCloud)

7. REFERENCES

[ANSS16] Asharov, G., Naor, M., Segev, G., and Shahaf, I. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In: 48th ACM STOC. ACM Press (2016).

[BFP16] Bost, R., Fouque, P.A., and Pointcheval, D. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. Cryptology ePrint Archive, Report 2016/062 (2016). http://eprint.iacr.org/2016/062.

[Bos16] Bost, R. Implementation of $\Sigma o\phi o\varsigma$ (2016). URL https://gitlab.com/sse/sophos.

[CGKO06] Curtmola, R., Garay, J.A., Kamara, S., and Ostrovsky, R. Searchable symmetric encryption: improved definitions and efficient constructions. In: A. Juels, R.N. Wright, and S. Vimercati (eds.), ACM CCS 06, pp. 79–88. ACM Press (Oct. / Nov. 2006).

[CGPR15] Cash, D., Grubbs, P., Perry, J., and Ristenpart, T. Leakage-abuse attacks against searchable encryption. In: I. Ray, N. Li, and C. Kruegel: (eds.), ACM CCS 15, pp. 668–679. ACM Press (Oct. 2015).

[Cip] CipherCloud. Cloud data encryption. URL http://www.ciphercloud.com/technologies/encryption/.

[CJJ+13] Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., and Steiner, M. Highly-scalable searchable symmetric encryption with support for boolean queries. In: R. Canetti and J.A. Garay (eds.), CRYPTO 2013, Part I, *LNCS*, vol. 8042, pp. 353–373. Springer, Heidelberg (Aug. 2013).

[CJJ+14] Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., and Steiner, M. Dynamic searchable encryption in very-large databases: Data structures and implementation. In: NDSS 2014. The Internet Society (Feb. 2014).

[CK10] Chase, M. and Kamara, S. Structured encryption and controlled disclosure. In: M. Abe (ed.), ASIACRYPT 2010, *LNCS*, vol. 6477, pp. 577–594. Springer, Heidelberg (Dec. 2010).

[CM05] Chang, Y.C. and Mitzenmacher, M. Privacy preserving keyword searches on remote encrypted data. In: J. Ioannidis, A. Keromytis, and M. Yung (eds.), ACNS 05, *LNCS*, vol. 3531, pp. 442–455. Springer, Heidelberg (Jun. 2005).

[CT14] Cash, D. and Tessaro, S. The locality of searchable symmetric encryption. In: P.Q. Nguyen and E. Oswald (eds.), EUROCRYPT 2014, *LNCS*, vol. 8441, pp. 351–368. Springer, Heidelberg (May 2014).

[Fac16] Facebook, Inc. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. (June 2016). URL http://rocksdb.org.

[GMP15] Garg, S., Mohassel, P., and Papamanthou, C. TWORAM: Round-optimal oblivious RAM with applications to searchable encryption. Cryptology ePrint Archive, Report 2015/1010 (2015). http://eprint.iacr.org/2015/1010.

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

[Gol04] Goldreich, O. Foundations of cryptography. Cambridge University Press (2004).

[Goo16] Google, Inc. gRPC: A high performance, open source, general RPC framework that puts mobile and HTTP/2 first. (April 2016). URL http://www.grpc.io/.

[IKK12] Islam, M.S., Kuzu, M., and Kantarcioglu, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: NDSS 2012. The Internet Society (Feb. 2012).

[Int] Intel. Intel® SSD 750 Series: Performance Unleashed. URL http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-750-series.html.

[JJK$^+$13] Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., and Steiner, M. Outsourced symmetric private information retrieval. In: A.R. Sadeghi, V.D. Gligor, and M. Yung (eds.), ACM CCS 13, pp. 875–888. ACM Press (Nov. 2013).

[KO12] Kurosawa, K. and Ohtaki, Y. UC-secure searchable symmetric encryption. In: A.D. Keromytis (ed.), FC 2012, *LNCS*, vol. 7397, pp. 285–298. Springer, Heidelberg (Feb. / Mar. 2012).

[KP13] Kamara, S. and Papamanthou, C. Parallel and dynamic searchable symmetric encryption. In: A.R. Sadeghi (ed.), FC 2013, *LNCS*, vol. 7859, pp. 258–274. Springer, Heidelberg (Apr. 2013).

[KPR12] Kamara, S., Papamanthou, C., and Roeder, T. Dynamic searchable symmetric encryption. In: T. Yu, G. Danezis, and V.D. Gligor (eds.), ACM CCS 12, pp. 965–976. ACM Press (Oct. 2012).

[MKNK15] Meng, X., Kamara, S., Nissim, K., and Kollios, G. GRECS: Graph encryption for approximate shortest distance queries. In: I. Ray, N. Li, and C. Kruegel: (eds.), ACM CCS 15, pp. 504–517. ACM Press (Oct. 2015).

[MMBC15] Moataz, T., Mayberry, T., Blass, E.O., and Chan, A.H. Resizable tree-based oblivious RAM. In: R. Böhme and T. Okamoto (eds.), FC 2015, *LNCS*, vol. 8975, pp. 147–167. Springer, Heidelberg (Jan. 2015).

[Nav15] Naveed, M. The fallacy of composition of oblivious RAM and searchable encryption. Cryptology ePrint Archive, Report 2015/668 (2015). http://eprint.iacr.org/2015/668.

[PKV$^+$14] Pappas, V., Krell, F., Vo, B., Kolesnikov, V., Malkin, T., Choi, S.G., George, W., Keromytis, A.D., and Bellovin, S. Blind seer: A scalable private DBMS. In: 2014 IEEE Symposium on Security and Privacy, pp. 359–374. IEEE Computer Society Press (May 2014).

[sky] skyhigh. Cloud security and enablement. URL https://www.skyhighnetworks.com.

[SPS14] Stefanov, E., Papamanthou, C., and Shi, E. Practical dynamic searchable encryption with small leakage. In: NDSS 2014. The Internet Society (Feb. 2014).

[SWP00] Song, D.X., Wagner, D., and Perrig, A. Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, pp. 44–55. IEEE Computer Society Press (May 2000).

[ZKP16] Zhang, Y., Katz, J., and Papamanthou, C. All your queries are belong to us: The power of file-injection attacks on searchable encryption. Cryptology ePrint Archive, Report 2016/172 (2016). http://eprint.iacr.org/2016/172.

APPENDIX

A. PROOF OF $\Sigma o\phi o\varsigma$ CONFIDENTIALITY

Let us first restate precisely the security claim for $\Sigma o\phi o\varsigma$.

Theorem 4. *Let π be a one-way trapdoor permutation, F a PRF, and H_1 and H_2 to hash functions modeled as a random oracle outputting respectively μ and λ bits. Define $\mathcal{L}_\Sigma = (\mathcal{L}_\Sigma^{\mathsf{Srch}}, \mathcal{L}_\Sigma^{\mathsf{Updt}})$ as*

$$\mathcal{L}_\Sigma^{\mathsf{Srch}}(w) = (\mathsf{sp}(w), \mathsf{Hist}(w))$$
$$\mathcal{L}_\Sigma^{\mathsf{Updt}}(\mathsf{add}, w, \mathsf{ind}) = \perp.$$

Then $\Sigma o\phi o\varsigma$-B is $\mathcal{L}_\Sigma$-adaptively-secure.

Proof. We are going to derive several games from the real world game $\mathrm{SSE}\textsc{Real}_A^{\Sigma o\phi o\varsigma}(\lambda)$.

Game G_0. G_0 is exactly the real world SSE security game $\mathrm{SSE}\textsc{Real}$.

$$\mathbb{P}[\mathrm{SSE}\textsc{Real}_A^{\Sigma o\phi o\varsigma}(\lambda) = 1] = \mathbb{P}[G_0 = 1]$$

Game G_1. Instead of calling F when generating K_w, G_1 picks a new random key when it is confronted to a new w, and stores it in a table `Key` so it can be reused next time w is queried. If an adversary is able to distinguish between G_0 and G_1, we can then build a reduction able to distinguish between F and a truly random function. More formally, there exists an efficient adversary B_1 such that

$$\mathbb{P}[G_0 = 1] - \mathbb{P}[G_1 = 1] \leq \mathbf{Adv}_{F,B_1}^{\mathtt{prf}}(\lambda).$$

Algorithm 2 Games G_2 and $\widetilde{G_2}$ Boxed code is included in $\widetilde{G_2}$ only.

```
Setup()
 1: $(\mathsf{SK}, \mathsf{PK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
 2: $\mathbf{W}, \mathbf{T} \leftarrow$ empty map
 3: $\mathsf{bad} \leftarrow \mathsf{false}$
 4: return $((\mathbf{T}, \mathsf{PK}), (\mathbf{K_S}, \mathsf{SK}), \mathbf{W})$
Search($w, \sigma$; EDB)
    Client:
 1: $K_w \leftarrow \mathtt{Key}[w]$
 2: $(ST_0, \ldots, ST_c, c) \leftarrow \mathbf{W}[w]$
 3: if $(ST_0, \ldots, ST_c, c) = \perp$
 4:     return $\emptyset$
 5: $(\mathsf{ind}_0, \ldots, \mathsf{ind}_c) \leftarrow \mathsf{HistDB}(w)$    ▷ Ordered from the order of updates
 6: for $i = 0$ to $c$ do
 7:     $\mathtt{H}_1(K_w, ST_i) \leftarrow \mathtt{UT}[w, i]$
 8: end for
 9: Send $(K_w, ST_c, c)$ to the server.
    Server:
10: for $i = c$ to $0$ do
11:     $UT_i \leftarrow H_1(K_w, ST_i)$
12:     $e \leftarrow \mathbf{T}[UT_i]$
13:     $\mathsf{ind} \leftarrow e \oplus H_2(K_w, ST_i)$
14:     Output each ind
15:     $ST_{i-1} \leftarrow \pi_{\mathsf{PK}}(ST_i)$
16: end for
Update(add, $w$, ind, $\sigma$; EDB)
    Client:
 1: $K_w \leftarrow \mathtt{Key}[w]$
 2: $(ST_0, \ldots, ST_c, c) \leftarrow \mathbf{W}[w]$
 3: if $(ST_0, \ldots, ST_c, c) = \perp$ then
 4:     $ST_0 \xleftarrow{\$} \mathcal{M}$, $c \leftarrow -1$
 5: else
 6:     $ST_{c+1} \leftarrow \pi_{\mathsf{SK}}^{-1}(ST_c)$
 7: end if
 8: $\mathbf{W}[w] \leftarrow (ST_0, \ldots, ST_{c+1}, c+1)$
 9: $UT_{c+1} \leftarrow \{0,1\}^\mu$
10: if $\mathtt{H}_1(K_w, ST_{c+1}) \neq \perp$ then
11:     $\boxed{\mathsf{bad} \leftarrow \mathsf{true},\ UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})}$
12: end if
13: $\mathtt{UT}[w, c+1] \leftarrow UT_{c+1}$
14: $d \leftarrow \mathsf{ind} \oplus H_2(K_w, ST_{c+1})$
15: Send $(UT_{c+1}, d)$ to the server.
    Server:
16: $\mathbf{T}[UT_{c+1}] \leftarrow e$
$H_1(k, st)$
 1: $v \leftarrow \mathtt{H}_1(k, st)$
 2: if $v = \perp$ then
 3:     $v \xleftarrow{\$} \{0,1\}^\lambda$
 4:     if $\exists w, c$ s.t $st = ST_c \in \mathbf{W}[w]$ then
 5:         $\boxed{\mathsf{bad} \leftarrow \mathsf{true},\ v \leftarrow \mathtt{UT}[w, c]}$
 6:     end if
 7:     $\mathtt{H}_1(k, st) \leftarrow v$
 8: end if
 9: return $v$
```

Game G_2. In G_2, in the Update protocol, instead of calling H_1 to generate the update tokens UT, we pick random strings. Then, during the Search protocol, the random oracle H_1 is programmed so that $H_1(K_w, ST_c(w)) = UT_c(w)$.

Algorithm 2 formally describes G_2, and also introduces an intermediate game $\widetilde{G_2}$, by including the additional boxed lines. In the pseudo-code, we explicit the calls to the random oracle H_1, and keep track of the transcripts *via* the table $\mathtt{H}_1$. In particular, we can see that we explicitly program the RO during Search at line 7. Note that we use the following convention for the table Key: if an entry is accessed for the first time, it is first randomly chosen and then returned. Also G_2 and $\widetilde{G_2}$ make some bookkeeping of the search tokens ST_c, instead of recomputing all of them in Search.

The point of $\widetilde{G_2}$ is to ensure consistency of H_1's transcript: in $\widetilde{G_2}$, H_1 is never programmed to two different values for the same input by Search' line 7. Instead of immediately generating the UT derived from the c-th ST for keyword w from H_1, $\widetilde{G_2}$ randomly either choses them if (K_w, ST_{c+1}) does not already appear in H_1's transcript, or, if this is already the case, sets UT_{c+1} to the already chosen value $\mathtt{H}_1[K_w, UT_{c+1}]$. Then, $\widetilde{G_2}$ lazily programs the RO when needed by the Search protocol (line 7) or by an adversary's query (line 5 of H_1), so that it's outputs are consistent with the chosen values of the UT's.

Because of this, H_1's outputs in $\widetilde{G_2}$ and G_1 are perfectly indistinguishable, and so are the games:

$$\mathbb{P}[\widetilde{G_2} = 1] = \mathbb{P}[G_1 = 1].$$

The games $\widetilde{G_2}$ and G_2 are also perfectly identical unless the flag bad is set to true, and we can apply the *identical-until-bad* technique to bound the distinguishing advantage between $\widetilde{G_2}$ and G_2:

$$\mathbb{P}[\widetilde{G_2} = 1] - \mathbb{P}[G_2 = 1] \leq \mathbb{P}[\mathsf{bad} \text{ is set to } \mathsf{true} \text{ in } \widetilde{G_2}].$$

Intuitively, we can see that, if bad is set to true, we can break the one-wayness of the TDP. More formally, we are going to construct a reduction B_2 from a distinguisher $\mathcal{A}$ inserting N keyword/document pairs in the database, using a technique similar to the Schnorr's signatures proofs. We note $\max_{w \in W} n_w = n_{\max} = \mathrm{poly}(\lambda)$ the maximum number of documents matching a keyword. B_2 will take as input a public key PK and a challenge $y \in \mathcal{M}$, and will ouput x such that $\pi_{\mathsf{PK}}(x) = y$.

As for Schnorr's signatures proofs, B_2 first guesses the pair (w^*, c^*) for which bad will be set to true for the first time, by querying H_1 on (K_{w^*}, ST_{c^*}) (*i.e.* by pre-computing $\mathtt{UT}[w^*, c^*]$), among the N possible pairs. For all keyword $w \in W \setminus \{w^*\}$, B_2 pre-computes $ST_i(w)$ as follows:

$$ST_{n_{\max}}(w) \xleftarrow{\$} \mathcal{M}$$
$$ST_i(w) \leftarrow \pi_{\mathsf{PK}}(ST_{i+1}(w)) \text{ for } 0 \leq i < n_{\max}$$

Similarly, for w^*, B_2 generates the search tokens from the challenge y:

$$ST_{c^*-1}(w^*) \leftarrow y$$
$$ST_i(w^*) \leftarrow \pi_{\mathsf{PK}}(ST_{i+1}(w^*)) \text{ for } 0 \leq i < c^*$$

It is essential to see that the distribution of the search tokens remains unchanged from game G_2. So, to return a pre-image of y, the reduction B_2 will find the value x by evaluating $\pi_{\mathsf{PK}}(r)$ for all (K_{w^*}, r) in the RO's transcript and check if $\pi_{\mathsf{PK}}(r) = y$. Hence, for a fixed pair (w^*, c^*), if G_2 sets bad to true because of an adversary's query on (K_{w^*}, ST_{c^*}), B_2 is able to invert π without the secret key:

$$\mathbb{P}\left[\mathsf{bad} \text{ is set to } \mathsf{true} \text{ by forging } \mathtt{UT}[w^*, c^*]\right] = \mathbf{Adv}^{\mathsf{OW}}_{\pi, B_2}(\lambda).$$

Guessing the pair (w^*, c^*) implies a N loss in the advantage of the reduction, and

$$\begin{aligned}\mathbb{P}[G_1 = 1] - \mathbb{P}[G_2 = 1] &= \mathbb{P}[\widetilde{G_2} = 1] - \mathbb{P}[G_2 = 1] \\ &\leq N \cdot \mathbf{Adv}^{\mathsf{OW}}_{\pi, B_2}(\lambda).\end{aligned}$$

Game G_3. Game G_3 does exactly what game G_2 did for H_1, but for H_2. The exact same argument can be reused, giving that there is an adversary B_3 such that

$$\mathbb{P}[G_2 = 1] - \mathbb{P}[G_3 = 1] \leq N \cdot \mathbf{Adv}^{\mathsf{OW}}_{\pi, B_3}(\lambda).$$

Note that we can consider that $B_2 = B_3$ without loss of generality: we could have started with H_2 instead of H_1 and the reduction would have been the same.

Game G_4. In game G_4, as defined by Algorithm 3, the game keeps track of the randomly generated string UT and d differently than before, but the transcript output by Search and Update are stricly identical, and the random oracles are also programmed identically. In Algorithm 3, we removed the now useless code for the H_1 oracle. Also note that we got rid of the server's part is the protocols: these are single roundtrip protocols and the removed lines do not influence the client's transcript.

We still have to show that G_3 and G_4 are indistinguishable. For Update, this is immediate as we are already outputting fresh random strings for each update in G_3. In Search, G_4 generates the search token from ST_0 by iterating π^{-1}_{SK} instead of using an already computed and stored token. As for $\mathtt{Key}$, if an entry of $\mathbf{W}$ if an entry is accessed for the first time, the game randomly picks it in $\mathcal{M}$.

Finally, instead of directly mapping the pairs (w, i) (a keyword and the i-th update to this keyword) to the values picked for $\mathtt{UT}$ and $\mathtt{e}$, we use the intermediate table $\mathtt{Updates}$ which implicitly maps (w, i) to the global update count. Hence,

$$\mathbb{P}[G_3 = 1] - \mathbb{P}[G_4 = 1] = 0.$$

Algorithm 3 Game G_4.

```
Setup()
  $(\mathsf{SK}, \mathsf{PK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
  $\mathbf{W}, \mathbf{T} \leftarrow$ empty map
  $u \leftarrow 0$
  $\mathtt{Updates} \leftarrow$ empty map
  return $((\mathbf{T}, \mathsf{PK}), (\mathbf{K_S}, \mathsf{SK}), \mathbf{W})$
Search($w, \sigma; \mathsf{EDB}$)
  Client:
  $K_w \leftarrow \mathtt{Key}[w]$
  $ST_0 \leftarrow \mathbf{W}[w]$
  $[(u_0, \mathsf{ind}_0), \ldots, (u_c, \mathsf{ind}_c)] \leftarrow \mathtt{Updates}[w]$
  if $c = 0$
    return $\emptyset$
  for $i = 0$ to $c$ do
    Program $H_1$ s.t. $H_1(K_w, ST_i) \leftarrow \mathtt{UT}[u_i]$
    Program $H_2$ s.t. $H_2(K_w, ST_i) \leftarrow \mathtt{e}[u_i] \oplus \mathsf{ind}_i$
    $ST_{i+1} \leftarrow \pi^{-1}_{\mathsf{SK}}(ST_i)$
  end for
  Send $(K_w, ST_c, c)$ to the server.
Update(add, $w$, ind, $\sigma; \mathsf{EDB}$)
  Client:
  Append $(u, \mathsf{ind})$ to $\mathtt{Updates}[w]$
  $\mathtt{UT}[u] \xleftarrow{\$} \{0,1\}^\mu$
  $\mathtt{e}[u] \xleftarrow{\$} \{0,1\}^\lambda$
  Send $(\mathtt{UT}[u], \mathtt{e}[u])$ to the server.
  $u \leftarrow u + 1$
```

The Simulator. We can – almost straightforwardly – cut the code of game G_4 in two independent parts: the leakage and the simulator. The simulator is described in Algorithm 4, and the leakage function is $\mathcal{L}_\Sigma$. G_4 and $\text{SSEIdeal}_{S, \mathcal{L}_\Sigma}$ are identical games, the only difference being that, instead of the keyword w, S uses the counter $\overline{w} = \min \mathsf{sp}(w)$ uniquely mapped from w using the leakage function. Hence,

$$\mathbb{P}[G_4 = 1] - \mathbb{P}[\text{SSEIdeal}^{\Sigma o\phi o\varsigma}_{A, S, \mathcal{L}_\Sigma}(\lambda) = 1] = 0.$$

Algorithm 4 Simulator S.

```
S.Setup()
  $(\mathsf{SK}, \mathsf{PK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
  $\mathbf{W}, \mathbf{T} \leftarrow$ empty map
  $u \leftarrow 0$
  return $((\mathbf{T}, \mathsf{PK}), (\mathbf{K_S}, \mathsf{SK}), \mathbf{W})$
S.Search(sp($w$), Hist($w$))
  Client:
  $\overline{w} \leftarrow \min \mathsf{sp}(x)$
  $K_{\overline{w}} \leftarrow \mathtt{Key}[\overline{w}]$
  $ST_0 \leftarrow \mathbf{W}[\overline{w}]$
  Parse $\mathsf{Hist}(w)$ as $[(u_0, \mathsf{add}, \mathsf{ind}_0), \ldots, (u_c, \mathsf{add}, \mathsf{ind}_c)]$
  if $c = 0$
    return $\emptyset$
  for $i = 0$ to $c$ do
    Program $H_1$ s.t. $H_1(K_{\overline{w}}, ST_i) \leftarrow \mathtt{UT}[u_i]$
    Program $H_2$ s.t. $H_2(K_{\overline{w}}, ST_i) \leftarrow \mathtt{e}[u_i] \oplus \mathsf{ind}_i$
    $ST_{i+1} \leftarrow \pi^{-1}_{\mathsf{SK}}(ST_i)$
  end for
  Send $(K_{\overline{w}}, ST_c)$ to the server.
S.Update()
  Client:
  $\mathtt{UT}[u] \xleftarrow{\$} \{0,1\}^\mu$
  $\mathtt{e}[u] \xleftarrow{\$} \{0,1\}^\lambda$
  Send $(\mathtt{UT}[u], \mathtt{e}[u])$ to the server.
  $u \leftarrow u + 1$
```

Conclusion. By combining all the contributions from all the games, there exists 2 adversaries B_1 and B_2 such that

$$\begin{aligned}\mathbb{P}[\text{SSEReal}^{\Sigma o\phi o\varsigma}_{A}(\lambda) = 1] &- \mathbb{P}[\text{SSEIdeal}^{\Sigma o\phi o\varsigma}_{A, S, \mathcal{L}_\Sigma}(\lambda) = 1] \\ &\leq \mathbf{Adv}^{\mathtt{prf}}_{F, B_1}(\lambda) + 2N \cdot \mathbf{Adv}^{\mathsf{OW}}_{\pi, B_2}(\lambda).\end{aligned}$$

We conclude by stating that π is a one-way function, and that F is a PRF. □