ORAM Based Forward Privacy Preserving Dynamic Searchable Symmetric Encryption Schemes

Panagiotis Rizomiliotis
Dep. of Information and Communication Systems Engineering
University of the Aegean
Karlovassi, GR 83200
Samos, Greece
prizomil@aegean.gr

Stefanos Gritzalis
Dep. of Information and Communication Systems Engineering
University of the Aegean
Karlovassi, GR 83200
Samos, Greece
sgritz@aegean.gr

ABSTRACT

In the cloud era, as more and more businesses and individuals have their data hosted by an untrusted storage service provider, data privacy has become an important concern. In this context, *searchable symmetric encryption (SSE)* has gained a lot of attention. An SSE scheme aims to protect the privacy of the outsourced data by supporting, at the same time, outsourced search computation. However, the design of an efficient dynamic SSE (DSSE) has been shown to be a challenging task.

In this paper, we present two efficient DSSEs that leak a limited amount of information. Both our schemes make a limited use of ORAM algorithms to achieve forward privacy and to minimize the overhead that ORAMs introduce, at the same time. To the best of our knowledge, there is only one other DSSE scheme that offers efficiently forward privacy. Our schemes are parallizable and significantly improve the search and update complexity, as well as the memory access locality.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*

General Terms

Security

Keywords

Searchable Encryption, Storage Outsourcing, ORAM

1. INTRODUCTION

In the cloud era, data outsourcing is becoming the dominant storage model. As businesses and individuals have their data hosted by an untrusted storage service provider, data privacy has become an important concern. One straightforward solution is to upload all data encrypted using one of the well studied symmetric encryption techniques. However, in this case the client has to download all its data and decrypt them in order to perform even simple computations, like data search. Clearly, such a solution is not practical.

In this context, *searchable symmetric encryption (SSE)* has gained a lot of attention. SSE aims to protect the privacy of user's outsourced data by supporting, at the same time, outsourced search computations. With a SSE scheme the client outsources its data encrypted, while she can perform keyword search queries without revealing the secret key or downloading the entire data set and searching herself. The client sends a search token to the server and the server performs the search operation without knowing the encryption key.

A predefined set of keyword search queries can be executed on the collection of the encrypted documents. The vast majority of the proposed schemes have a setup phase where an encrypted index is computed for the specific collection of documents, i.e. each keyword is related to a precomputed set of file identifiers. If the index remains unaltered after this phase, then the SSE scheme is called *static SSE*. If additions and deletions are supported, then it is a *dynamic SSE (DSSE)* scheme.

In terms of security, SSE schemes leak information. This leakage can be minimized using primitives, like the Oblivious RAM (ORAM) algorithms. However, these solutions are very costly. The last two years ORAM proposals and implementations are getting more practical, but there is still a performance gap to be filled. In order to make SSE schemes practical, we tolerate some more leakage. SSE leakage contains at minimum the search pattern and the access patten. By search pattern we refer to the hashes of the keywords, i.e. it is leaked when the same keyword is searched, while the access pattern is the matching document identifiers of a keyword search.

In the case of DSSEs, some more information leakage is expected. More precisely, the access pattern includes also the document identifiers that have been added or deleted. Two security notions are relevant to DSSE's security, namely the *forward* and *backward privacy*. By forward privacy we mean that, when a new keyword and file identifier pair is added, the server does not learn anything about this pair. In backward privacy, queries cannot leak the file identifiers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CCSW'15, October 16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3825-7/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2808425.2808429.

of deleted documents. Designing an efficient DSSE scheme that possesses both these security properties has been shown to be a very difficult task. To the best of our knowledge there is no DSSE solution that offers backward privacy, while only one DSSE scheme exists that is practical and supports forward privacy ([19]).

Our contribution. In this paper, we introduce two efficient DSSE schemes that achieve forward privacy with very small leakage. Our schemes use simple structures, like dictionaries, to store the inverted index for keyword (i.e. the files per keyword) and the index for file (i.e. the keywords per file), and more sophisticated structures, like the ORAM algorithms, to store the client's state.

ORAM is a cryptographic primitive designed to conceal access patterns, when a client with small (and preferably constant) local memory executes a sequence of reads and writes to remotely stored data. In this paper, we use ORAM as a black box and, thus, any proposal can be applied. For the performance evaluation, we consider two of the most efficient ORAMs presented in [12] and [18]. While it is generally believed that ORAMs are still very costly primitives, we show how to make a limited use of an ORAM algorithm in order to achieve forward security and, at the same time, to minimize the overhead that this construction introduces. Both our schemes can be seen as an extension of a very efficient and scalable static SSE proposed by Cash et al. ([2]). Our scheme has the following characteristics:

- *Leakage.* We present two schemes that follow the same philosophy, the 'simple' and the 'extended' DSSE schemes. Both of them leak the access and the search pattern after each search operation for a keyword. Also, on file update, both schemes offer forward privacy (but not backward privacy) and the only information leaked is the number of keywords per added file. Each file that it is added, even if it was previously deleted from the DSSE, it is treated as a completely new file with new file identifier. Finally, the two schemes differ on the information leaked at the setup phase. Both of them leak the number of keywords and files (can be deduced from the size of the structures and can be hidden with padding). However, the 'simple' DSSE leaks also the number of keywords per file. The 'extended' DSSE hides this information.

- *Efficiency.* The search complexity of both of our schemes is $O(\max(\log^2(|W|)/\gamma, |I_\omega|))$, where $|W|$ is the number of keywords, $|I_\omega|$ the number of files that contain the keyword ω and γ depends on the ORAM scheme that we use. The update complexity is, in the worst case, $O(\max(|W_f|\log^2(|W|)/\gamma, \log^2(|F|)/\gamma)$, where $|W_f|$ is the number of keywords that the file f contains and $|F|$ is total number of files. We compare our scheme with the DSSE of Stefanov et al. ([19]), i.e. the only other practical DSSE that offers forward privacy. Their scheme's search and update complexity depends on the total number N of keyword and file identifier pairs, while our proposal has the nice feature to be independent of N.

- *Parallelization and locality.* Both our schemes are fully parallizable, i.e. the Search and Update operations can be performed with parallelism. Also, we have decided to divide the information into two categories: leaked and not leaked. Thus, the leaked data can be treated differently and are stored in contiguous area of memory positions optimizing locality (i.e. we replace dictionary reads with array reads).

- *File index.* Usually, the DSSE schemes do not support the storage of the index for file. However, at the same time, the keywords per file is expected as input when a file is deleted and the schemes do not demonstrate how such information is available, with very few exceptions ([10]). Our scheme maintains the index for file efficiently and parallelism is possible.

Paper organization. In Section 2, background and comparison with prior work are presented. In Section 3, the necessary definitions and notation are provided, while in Section 4 we introduce the simplest of our schemes. The extended and more secure version appears in Section 5. Illustrative examples are given for the better comprehension of the designs. In Section 6 and in Section 7, the complexity and the security of the schemes are analyzed, respectively. Finally, Section 8 concludes the paper.

2. RELATED WORK

2.1 Background

In this section, we present the most important DSSE solutions. Several efficient static SSE schemes have been proposed, but their presentation is out of the scope of our work.

The problem of searching (symmetrically) encrypted data that can be modified can be solved with minimum information leakage by using either the Private Information Retrievabilty (PIR) protocols ([4], [16], [21]) or ORAM algorithms ([7], [14], [8], [12]). However, such solutions are not attractive. PIRs have linear search complexity, while ORAMs still require high communication overhead.

In [17], Song et al. presented an SSE that supports insertions/deletions of files. Their scheme is a semantically secure encryption scheme that leaks access patterns and it is linear in the size of all data. In [6], Goh introduced a DSSE scheme that also has linear search complexity. In 2010, van Liesdonk et al. ([20]) proposed a DSSE with a rather large storage and the updates leak information more than it is usually accaptable from a DSSE.

In 2012 Kamara et al. ([10]) introduced a sublinear search time DSSE. They achieve such an improvement at the expense of more leakage (their scheme in every update reveals hashes of the unique keywords contained in the updated document). A year later, Kamara and Papamanthou ([11]) fixed this problem by increasing the index size. This was the first parallel DSSE. In 2014, Cash et al. ([2]) proposed a very simple DSSE based on dictionaries that scales nicely. This scheme however, is optimal when the number of updates is small, i.e. it has great performance only as a semi-static SSE. The dynamic version of this scheme is not efficient mainly with respect to storage management. Our schemes are partially based on this proposal. Finally, very recently, Hahn and Kerschbaum ([9]) proposed a dynamic scheme that leaks only the access pattern and has asymptotically optimal search time.

However, none of the above schemes offers forward privacy. To the best of our knowledge, only two DSSE solutions possess this property. Chang and Mitzenmacher's ([3]) proposal achieves forward privacy, but at the cost of linear

Scheme	Search Time	Index Size	Update Cost	Forw.Priv.	In. Leak
[19]	$O(\min(\alpha + \log(N), \lvert I_\omega \rvert \log^3 N))$	$O(N)$	$O(\lvert W_f \rvert \log^2 N))$	Yes	N
[2]	$O(\lvert I_\omega \rvert + d_\omega)$	$O(N)^*$	$O(\lvert W_f \rvert + \lvert W \rvert \log \lvert F \rvert))$	No	N
'Ext' (this paper)	$O(\max(\log^2(\lvert W \rvert)/c, \lvert I_\omega \rvert))$	$O(N)$	$O(\max(\lvert W_f \rvert \log^2 \lvert W \rvert, \log^2 \lvert F \rvert))$	Yes	N, $\lvert W \rvert$, $\lvert F \rvert$
'Ext' (given $\lvert W \rvert$)	$O(\lvert I_\omega \rvert)$	$O(N)$	$O(\max(\lvert W_f \rvert, \log^2 \lvert F \rvert))$	Yes	N, $\lvert F \rvert$

Table 1: Comparison of our schemes with the DSSE from [19] and [2]. For the [2], the index size of the static SSE is used. The index size of the dynamic version depends on the sequence of search and update operations.

search complexity. The second scheme, was recently introduced by Stefanov, Papamanthou and Shi ([19]). It is an hierarchical construction, inspired by the ORAM algorithms, that offers forward privacy with minimum leakage.

2.2 Comparison with prior work

Next, we compare our schemes with two proposals, the DSSEs presented in [19] and [2]. The first one is the only efficient DSSE that offers forward privacy, while the second DSSE's design is similar to ours. In Table 1, there is a summary of the comparison. We use only the extended version as both our schemes have the same performance.

In [19], the worst-case search complexity is asymptotically $O(\min(\alpha + \log N, |I_\omega| \log^3 N)$, where $|I_\omega|$ is the number of documents containing the keyword ω we are searching for, α is the number of times this keyword was historically added to the collection and N is the total number of keyword/file identifier pairs. We see that the complexity depends on the stored pairs. As this number increases, the search complexity increases. Our scheme is independent from N and depends only on the number of the keywords $|W|$. Thus, for a given number of keywords, our scheme is linear on $|I_\omega|$, i.e. it is optimal. Also, in [19], the worst-case update complexity is $O(|W_f| \log^2 N)$, and the space of the data structure is $O(N)$. Again, our update complexity is independent from N, while our storage complexity is also optimal.

In [2], the SSE was mainly designed to serve as a static scheme. The authors describe how to modify the static SSE in order to support updates, but they use revocation lists. Thus, when several additions and deletions are performed the list increases linearly and the storage is not efficiently used. Also, the search overhead increases as the revocation list must be searched. However, the scheme with limited file updates scales nicely and supports very large databases. They use dictionaries and each label is computed in a way that facilitates parallelization. We follow the same approach. Also, their scheme leaks less information than ours, at the setup phase. More precisely, it leaks only N, while our (the extended version) scheme leaks also the number of keywords. Finally, they do not offer forward (and backward) privacy.

3. DEFINITIONS

3.1 Notation

First, we introduce some notation. The set of binary strings of length n is denoted as $\{0,1\}^n$, the cardinality of a set X is denoted as $|X|$. A function $G : \mathbb{N} \to \mathbb{R}$ is negligible in x if for every positive polynomial $p(\cdot)$ there exists a x_0 such that for all $x > x_0$, $g(x) < \frac{1}{p(\cdot)}$. We denote the output z of a (possibly probabilistic) algorithm A as $z \leftarrow A$. Sampling uniformly random from a set X is denoted as $x \leftarrow X$.

We use λ to denote the security parameter. Let F be the set of all files and W the set of all the keywords. Each file $f \in F$ has a unique file identifier $label_f \in \{0,1\}^\lambda$. Also, each keyword has a unique identifier, but to simplify the notation, we use the symbol ω_j to refer to both the keyword and its identifier. Each file f_i contains a set of keywords $W_{f_i} \subseteq W$ and each keyword $\omega_j \in W$ is contained in a subset of the files $F_{\omega_j} \subseteq F$. The set of all file identifiers that contain a keyword ω_j is denoted by I_{ω_j} and it is defined as $I_{\omega_j} = \{label_f : f \in F_{\omega_j}\}$. Let DB defined as $DB = (label_i, W_{f_i})^d{}_{i=1}$, i.e. it is a list of file identifier/keyword pairs $W_{f_i} \subset \{0,1\}^*$, and let N be the total number of file/keyword pairs.

3.2 DSSE and security definitions

Our definition is similar to previous works.

Definition 1. *(DSSE). A dynamic index-based SSE scheme is a tuple of three polynomial-time algorithms DSSE = (SetUp, Search, Update) such that:*

- $(K, \sigma) \leftarrow SetUp(1^\lambda, DB)$*: is a probabilistic algorithm that takes as input a security parameter λ and outputs the clients secret state K and, it initiates the DSSE data structure and returns its initial state σ.*
- $(I_\omega, \sigma') \leftarrow Search(K, \sigma, \omega)$*: is a deterministic algorithm that takes as input the secret state K, the state of the SSE structure σ and a keyword ω. It outputs a set of file identifiers I_w and the updated SSE structure state σ'.*
- $\sigma' \leftarrow Update(K, \sigma, label_f, op)$*: is a deterministic algorithm that takes as input the secret key state K, the DSSE structure state σ, a file identifier $lalel_f$ and the type of the operation (op = 'add' or 'delete'). It adds (deletes) the file in (from) the DSSE structure and outputs the updated DSSE structure state σ'.*

We use the standard simulation model to define the scheme's security, in line with all the schemes that followed Curtmola et al. ([5]) work. Our scheme is secure in the semi-honest model, where the server follows the protocol, but is curious. We allow the server to learn some information and we use leakage functions to define this knowledge. We define three leakage functions, namely $\mathcal{L}_{stp}$, $\mathcal{L}_{srch}$ and $\mathcal{L}_{upd}$, one for each of the three DSSE operations.

$Ideal_{\mathcal{F},\mathcal{S},\mathcal{Z}}$: Let $\mathcal{F}$ be the ideal functionality and $\mathcal{S}$ the simulator, i.e. the ideal world adversary. Initially, an environment $\mathcal{Z}$ sends the client a message "setup" and the client forwards this message to $\mathcal{F}$. The simulator $\mathcal{S}$ learns $\mathcal{L}_{stp}$ (the setup leakage function is defined in Section 4.1). In each time step, the environment $\mathcal{Z}$ selects either a search or an update operation. That is that, it either chooses a keyword ω and the client sends the search operation to the ideal functionality $\mathcal{F}$ or it chooses a file identifier $label_f$ and

an operation $op =$ 'add' (and the set of keywords as input) or $op =$ "$delete$" and the client sends the update operation to the ideal functionality $\mathcal{F}$. In the first case, the simulator is notified of $\mathcal{L}_{srch}$ (the search leakage function is defined in Section 4.1), while in the second case it learns $\mathcal{L}_{upd}$ (the update leakage function is defined in Section 4.1). Next, $\mathcal{S}$ sends $\mathcal{F}$ either abort or continue message and, as a result, $\mathcal{F}$ sends the client "protocol abort", "update success", or the identifiers of the matching documents for the search query. The environment $\mathcal{Z}$ observes these outputs. Finally, the environment $\mathcal{Z}$ outputs a bit b.

$Real_{\pi,\mathcal{A},\mathcal{Z}}$: Let π be the real world DSSE protocol and $\mathcal{A}$ the real world adversary, i.e. the server. Initially, an environment $\mathcal{Z}$ sends the client a message "setup" and the client executes the real world SetUp protocol with the server. In each time step, the environment $\mathcal{Z}$ selects either a search or an update operation. That is that, it either chooses a keyword ω and the client executes the real world Search protocol with the server or it chooses a file identifier $label_f$ and a keyword set I_ω (only for file addition), an operation $op =$ 'add' or $op =$ "$delete$" and the client executes the real world Update protocol with the server using the selected inputs. The environment $\mathcal{Z}$ observes the client's outputs, i.e. either "protocol abort", "update success" or the identifiers of the matching documents for a search query. Finally, the environment $\mathcal{Z}$ outputs a bit b.

DEFINITION 2. *The protocol π is $(\mathcal{L}_{stp}, \mathcal{L}_{srch}, \mathcal{L}_{upd})$-secure against adaptive dynamic chosen keyword attacks, if for all polynomial-time semi-honest real world adversaries A, there exists a probabilistic polynomial-time simulator $\mathcal{S}$, such that for all non-uniform polynomial time environment $\mathcal{Z}$*

$$|Pr[Real_{\pi,\mathcal{A},\mathcal{Z}}(\lambda) = 1] - |Pr[Ideal_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1]| \leq negl(\lambda).$$

4. OUR SIMPLE SCHEME

In this Section, we present the simplest of our schemes. In order to support queries, the inverted index for keyword must be stored, while addition/deletion of files requires the maintenance of the file index.

The scheme is based on the following observation: at any time t the set I_ω of the file identifiers related to a keyword ω can be divided into two subsets $I_\omega^{(0)}$ and $I_\omega^{(1)}$. The first subset $I_\omega^{(0)}$ contains the file identifiers that have been used in a previous search for ω, i.e. they have been leaked, and that they have not been deleted. Clearly, we do not have to protect the privacy of this set and, we only care to store the identifiers efficiently. Thus, we use a data structure S_0 to efficiently store and manage this set of identifiers, without any security requirements. The second subset $I_\omega^{(1)}$ contains the file identifiers that have not been used in a search reply, since their last addition into the DSSE. This second subset is stored in another data structure S_1 that protects the set's privacy. Note that this set includes also the file identifiers that were leaked, deleted and, then, added again. Both, S_0 and S_1 are used to store the inverted index for keyword. After each search operation the corresponding S_1 entries are moved to S_0. Finally, in order to support file update, we use a third data structure S_3 to store the file index. That is that, we store the sequence of keywords W_i per file $f_i \in F$.

For our simple DSSE scheme, we are using dictionaries and ORAMs. More precisely, for the three main data structures, i.e. S_0, S_1 and S_2, we are using dictionaries, while we employ two ORAM structures to store the client's state. The first ORAM, $ORAM_\omega$, is used to store keyword related information, and the second ORAM, $ORAM_f$, is used for the client's file state. This gives us forward privacy. We also use an $IND-CPA$ secure secret-key encryption scheme SKE, and a random oracle $H : \{0,1\}^\lambda \times \{0,1\}^* \leftarrow \{0,1\}^\lambda$, where λ is the security parameter. The secret-key encryption scheme is a tuple of three algorithms $SKE = (Gen, Enc, Dec)$ and it is used to encrypt the files F. The ciphertexts are outsourced and managed using the corresponding file identifier. For the rest of the paper, when we refer to label or data, we mean a bitstring.

The S_0 data structure supports the following three operations:

- $insert(l, d)$: inserts the data item d in the structure with label l. If d is the j-th item with label l, the value j is returned.
- $get(l)$: reads all data items from the structure with label l. Returns the set of the deleted items.
- $remove(l, j)$: removes the j-th entry item with the label l. Returns "OK" when data is deleted.

The S_1 data structure supports the following operations:

- $insert(l, d)$: stores the data item d with label l. Returns "OK" when data is stored.
- $removeAll(l)$: deletes the all data with label l. Returns all data with that label.

The S_2 data structure supports three operations:

- $insert(l, d)$: inserts the data item d in the structure with label l. Returns "OK" when data is stored.
- $removeAll(l)$: deletes the all data with label l. Returns all data with that label.
- $update(l, j, d)$: uses data d to update the j-th entry item with label l. Returns "OK" when data is updated.

Finally, it is possible to read and write data from and into the two ORAM structures using the secret keys K_W and K_F.

Each file has a unique file identifier $label_f$. This identifier changes each time the file is deleted from the DSSE and a new one is randomly selected when the file is added again. On the other hand, each keyword has a unique and invariant identifier ω and a temporary secret key K_ω. A new key value K_ω is selected each time we search for the keyword ω. All labels l_p used in S_1 are computed by applying the random oracle to the secret key K_ω and a counter.

For each keyword/file identifier pair $(\omega, lable_f)$ there are two dictionary entries, one in S_2 and one either in S_0 or in S_1. More precisely, if the pair belongs in $I_\omega^{(1)}$, then there is an entry in S_1 with label l_p

$$S_1(l_p) = (label_f | j)$$

and one entry in S_2

$$S_2(label_f) = (l_p | '1')$$

where $l_p = H(K_\omega | j)$ and $'1'$ indicates that the pair has not been read yet. Also, j is the value of the counter used to

compute l_p and at the same time indicates that it is the j-th entry in S_2 with label $label_f$.

On the other hand, if the pair belongs in $I_\omega^{(0)}$, then in S_0 there is an entry with label ω, such that

$$S_0(\omega) = (label_f|j).$$

i.e. the entry copied as it is from S_1 and, in S_2,

$$S_2(label_f) = (\omega|i|'0')$$

where $'0'$ indicates that the identifier/keyword pair has been read and it is stored in S_0 with label ω and it is the i-th item with this label.

For the rest of the paper, we use the following notation to refer to the different parts of an entry. Thus, each entry p in S_2 is of the form $p = (p.l|p.s)$ or $p = (p.l|p.j|p.s)$, i.e. $p.s$ equals the single status bit at the end of the entry, $p.l$ is the label at the beginning and $s.j$ the value j, when $p.s =' 0'$. Similarly, an entry p in S_1 is of the form $p = (p.l|p.j)$.

Finally, the $ORAM_\omega$ structure's data is a pair (K_ω, c) per keyword ω, where K_ω is the current secret key used to label the entries in S_1 related to ω and c indicates how many such entries have been added since the last search for ω. On the other hand, $ORAM_f$ entries store only the current identifier $label_f$ of a file f.

Secret keys.

Three λ-bit secret keys are stored at the client, namely K_F, K_W and K. The first two are the secret keys for the two ORAM structures, while the third is used for the file encryption.

The Simple DSSE operations.

The operations of the DSSE appear in Fig. 1, Fig. 2, Fig. 3 and Fig. 4. More precisely:

Setup operation. The client chooses three random λ-bit strings K_F, K_W and K, for the two ORAM structures, and for the files encryption (using the SKE $Gen(1^\lambda)$). The client initiates the two ORAM structures $ORAM_\omega$ and $ORAM_f$ of size $|W|$ and $|F|$ and fills in S_1 and S_2 with the inverted index for keyword and the file index respectively. Initially, the structure S_0 is empty.

Search operation. The search for a keyword ω returns the set of file identifiers I_ω. It is performed in two steps. The first step is straightforward and very efficient as it has to do with the retrieval of the file identifiers that have been used in a previous search. These identifiers are stored with the same label ω in S_0. This step is parallizable with high locality. The second step is related to the retrieval of the identifiers set stored in S_1. Using the secret key K_ω of the keyword, and the counter c the server computes all the labels that correspond to the keyword ω. This step is parallizable as well. The retrieved identifiers from S_1 are then stored in S_0 using the label ω and the corresponding entries in S_2 are updated. Finally, the keyword's secret key K_ω is updated with a new randomly selected laue and the counter is set to zero.

Add file operation. When a file $f_i \in F$ is added, the corresponding sequence of keywords W_i must be stored in S_2 using a completely new label $label_f$, i.e. we treat the file as a new one, even if it was deleted in the past from the DSSE. Also, the structure S_1 must be updated. For each keyword $\omega_j \in W_{f_i}$ a new label is computed by the client using the keyword's secret key K_{ω_j} and the keyword's counter. The secret key and the counter are retrieved obliviously from the ORAM $ORAM_W$. The counter is increased by one.

Delete file operation. When a file f_i is deleted, $label_f$ is retrieved from $ORAM_F$. We use the information stored at S_2 with the same label $label_f$ to find and delete all the entries in S_0 and S_1 that correspond to a keyword $\omega_j \in W_i$. Finally, all entries with label $label_f$ are deleted from S_2.

Algorithm $SetUp(DB)$

1. $K \xleftarrow{\$} SKE.Gen(1^\lambda)$; $K_F \xleftarrow{\$} \{0,1\}^\lambda$; $K_W \xleftarrow{\$} \{0,1\}^\lambda$
2. Initiate two ORAMs of size $|W|$ and $|F|$
3. For each $\omega \in W$:
   $K_\omega \xleftarrow{\$} \{0,1\}^\lambda$; $c = I_\omega$
   $ORAM_\omega.write(\omega, (K_\omega, c))$
4. For each $f \in F$:
   Encrypt f: $f' = SKE.Enc(K, f)$
   $label_f \xleftarrow{\$} \{0,1\}^\lambda$
   Add $(label_f, f')$ to list C
   $ORAM_f.write(f, label_f)$; i = 0;
   For each $\omega \in W_f$:
      $i++$
      $p = H(K_\omega|i)$
      $S_1.insert(p, (label_f|i))$
      $S_2.insert(label_f, (p|'1'))$
5. Output C, S_1, S_2

Figure 1: The Setup protocol of the simple DSSE scheme.

4.1 Leakage functions

The *SetUp* operation leaks the number of files $|F|$ and keywords $|W|$, the size of each file and the number of keywords per file $|W_i|$,i.e.

$$\mathcal{L}_{stp} := (|F|, |W|, |W_i|_{1\le i\le n}, len(f_i)_{1\le i\le n}).$$

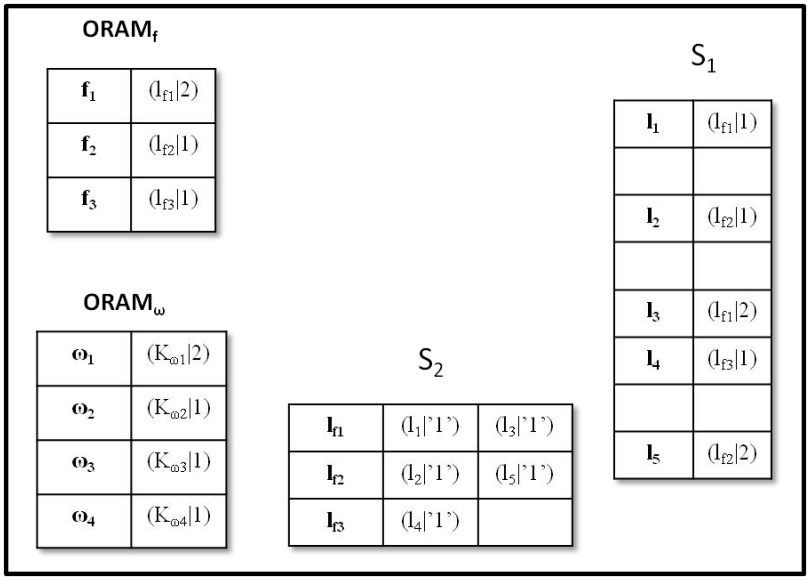
A search operation for a keyword ω leaks the set of file identifiers I_ω matching the keyword, that have been added or deleted in the past (i.e. the access pattern at time t $ACCP_t(\omega)$), and the time the same keyword was accessed in the past (i.e. the search pattern $SEAP_t(\omega)$). The search leakage is defined as $\mathcal{L}_{srch} := (\omega, I_\omega, ACCP_t(\omega), SEAP_t(\omega))$.

The Add operation leaks the file identifier $label_f$, the number of keywords $|W_i|$ and the file size $len(f)$. The Delete operation leaks only the file identifier $label_f$. Thus, in total we define

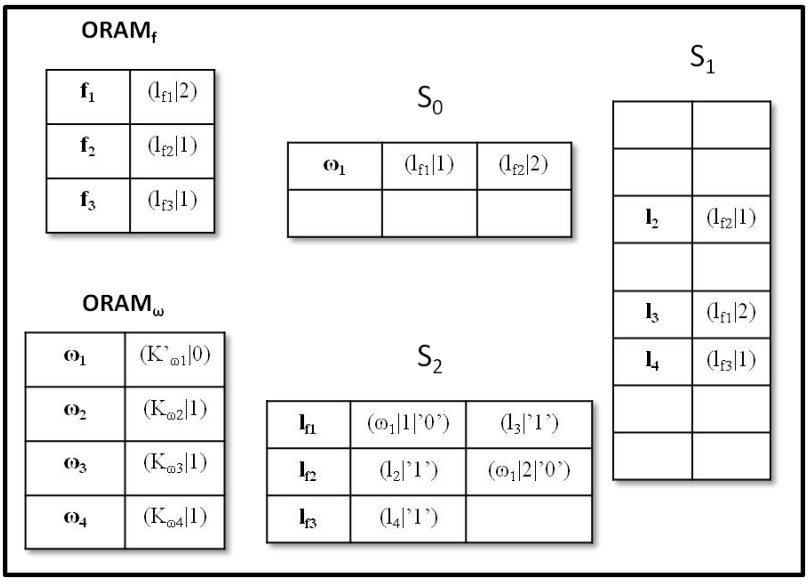
$$\mathcal{L}_{upd} := (op, label_f, len(f), |W_f|),$$

where op is the type of operation.

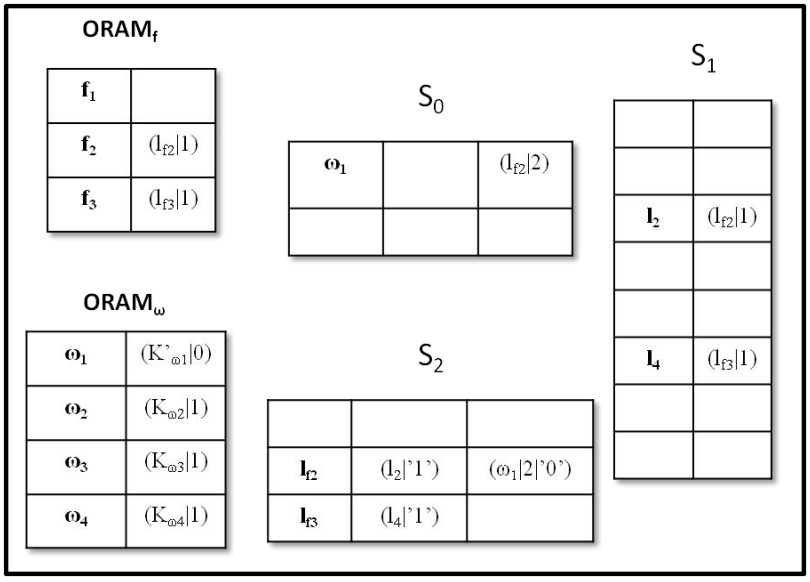
Note that, similarly to [19], our definition of leakage captures forward privacy. That means that, the leaked set I_ω contains only documents that were added in the past, but no future file additions. However, I_ω can contain deleted



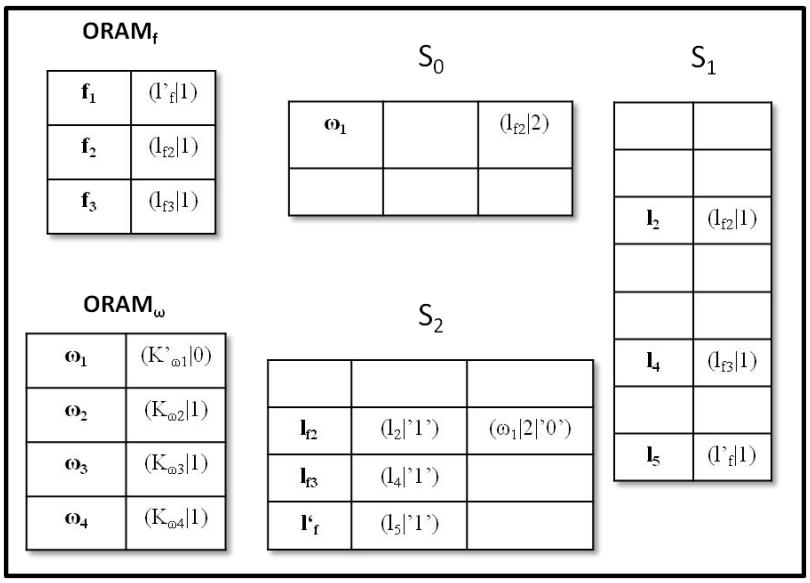
(a) Initial state of the simple DSEE.



(b) After Search for ω_1.



(c) After deleting f_1.



(d) After the addition of the modified f_1.

Figure 5: Toy example for the simple DSSE scheme.

Algorithm $Search(\omega)$
The client side

1. $(K_\omega, c) \leftarrow ORAM_\omega.read(\omega)$
2. $K'_\omega \xleftarrow{\$} \{0,1\}^\lambda$; $c' = 0$
3. $ORAM_\omega.write(\omega, (K'_\omega, c'))$
4. Output $\omega, (K_\omega, c)$

The server side

1. $I_\omega \leftarrow S0.get(\omega)$
2. For $i = 1 : c$ do
   $p \leftarrow H(K_\omega|i)$
   $block \leftarrow S_1.remove(p)$
   $I_\omega \leftarrow I_\omega \cup block.l$
   $j \leftarrow S0.insert(\omega, block)$
   $S_2.update(block.l, block.j, (\omega|j|'0'))$
3. Output I_ω

Figure 2: The Search protocol of the simple DSSE scheme.

files, i.e. our definition of leakage does not satisfy backward privacy.

Algorithm $Delete(f)$
The client side

1. $label_f \leftarrow ORAM_f.read(f)$
2. Output $label_f$

The server side

1. Delete the ciphertext with label $label_f$
2. $I_p \leftarrow S_2.removeAll(label)$
3. For $p \in I_p$ do
   If $p.s$==”l” then
   $S_1.remove(p.l)$
   else
   $S_0.remove(p.l, p.j)$
4. Output “OK”

Figure 3: The Delete protocol of the simple DSSE scheme.

4.2 A Toy Example

In this section, we present a toy example to explain the design philosophy of our scheme. Let $\{f_1, f_2, f_3\} \in F$ and that $\{\omega_1, \omega_2, \omega_3, \omega_4\} \in W$. The initial keyword sequences are: $W_{f_1} = \{\omega_1, \omega_2\}$, $W_{f_2} = \{\omega_1, \omega_3\}$, and $W_{f_3} = \{\omega_4\}$ (see Fig. 5).

Search ω_1**.** The client first queries $ORAM_\omega$ and retrieves obliviously the secret key K_{ω_1} and the counter value c, sends them to the server, and stores back into the $ORAM_\omega$ a new random key K'_{ω_1} and sets the counter value to zero. The server uses the key and the counter to compute two labels l_1 and l_5 and adds the stored labels to $I_{\omega_1} = \{l_{f_1}, l_{f_2}\}$. Since S_0 is initially empty, I_{ω_1} is ready. Then, it deletes the two

Algorithm $Add(f, W_f)$
The client side

1. Encrypt f: $f' = SKE.Enc(K, f)$
2. $label_f \xleftarrow{\$} \{0,1\}^\lambda$
3. For $\omega \in W_f$ do
   $(K_\omega, c) = ORAM_\omega.read(\omega)$
   $c++; ORAM_\omega.write(\omega, (K_\omega, c))$
   $p = H(K_\omega|c)$
   Add p to list L
4. $ORAM_f.write(f, label_f)$
5. Output L, $label_f$, f'

The server side

1. Store f' using label $label_f$
2. For $i = 1 : |L|$:
   $p = L[i]$
   $S_1.insert(p, (label_f|i))$
   $S_2.insert(label_f, (p|'1'))$

Figure 4: The Add protocol of the simple DSSE scheme.

entries from S_1, adds two new items with the same label ω_1 into S_0 and updates the corresponding items in S_2 with labels in I_{ω_1}.

Delete file f_1. The client first queries $ORAM_f$ and retrieves the files current label l_{f_1} and sends it to the server. The server removes all the items with label l_{f_1} from S_2. Following the first item $(\omega_1|1|'0')$, the server deletes from S_0 the first item with label ω_1 and following the second item $(l_3|'1')$, the server deletes from S_1 the item with label l_3.

Add file f_1. The client wants to store a modified version of the file f_1 and let now $W_{f_1} = \{\omega_1\}$. The client selects a new label l'_{f_1} for the file. Then, from $ORAM_\omega$ the client retrieves obliviously the secret key K'_{ω_1} and the counter value of ω_1. The counter is incremented by one and label $l'_5 = H(K'_\omega|1)$ is computed and the counter is stored back into $ORAM_\omega$ and the new label l'_{f_1} with the number of keywords is obliviously stored into $ORAM_f$. The client encrypts the file f_1 using the key K and sends to the server the cipher text f', the label l'_5 and the label l'_{f_1}. The server updates S_1 and S_2 accordingly.

5. OUR EXTENDED SCHEME

In this section, we present a more secure version of the DSSE scheme. More precisely, this extended version hides the number of keywords $|W_f|$ per file for the initial set of files, i.e. the SetUp leakage function is now

$$\mathcal{L}_{stp} := (|F|, |W|, N, len(f_i)_{1 \le i \le n}).$$

The *Search* and *UpDate* leakage functions are exactly the same as in the simple case.

The extended scheme treats the searched keywords and the newly added files the same way as in the simple version, i.e. the structures S_0, S_1 and S_2 and the ORAMs $ORAM_\omega$ and $ORAM_f$ are as before. However, two new structures are used to manage the files and the related keywords that were available at the setup phase. Namely, the structures $\hat{S}_1$ and $\hat{S}_2$. The first one, $\hat{S}_1$, is used to store the inverted index for keywords and the second one, $\hat{S}_2$ is used for the file index storage. Both of them are dictionaries with the entries encrypted and a different label per entry.

The $\hat{S}_1$ data structure supports the following operations:

- $insert(l, d)$: stores the data item d with label l. Returns "OK" when data is stored.
- $removeAll(l)$: deletes the all data with label l. Returns all data with that label.

The $\hat{S}_2$ data structure's operations are:

- $insert(l, d)$: inserts the data item d in the structure with label l. Returns "OK" when data is stored.
- $removeAll(l)$: deletes the all data with label l. Returns all data with that label.
- $update(l, j, d)$: updates the j-th entry item with label l. Returns "OK" when data is updated.

Secret keys.

Besides the three λ-bit secret keys K_F, K_W and K that are used in the simple scheme another λ-bit secret key $\hat{K}$ is needed. This key is used to compute two keys per keyword and two per file for the initial set of files. More precisely, for a keyword ω the keys $\hat{K}_\omega^{(1)} = H(\hat{K}|1|\omega)$ and $\hat{K}_\omega^{(2)} = H(\hat{K}|2|\omega)$ are defined and for each file f the keys $\hat{K}_f^{(1)} = H(\hat{K}|1|f)$ and $\hat{K}_f^{(2)} = H(\hat{K}|2|f)$.

Again, each file f has a random (that changes with each file update) identifier $label_f$. For each pair $(\omega, label_f)$ that appears in the initial set of files, two entries are stored, one into $\hat{S}_1$ and one into $\hat{S}_2$. More precisely, the labels $l_p = H(\hat{K}_\omega^{(1)}|c)$ and $l'_p = H(\hat{K}_f^{(1)}|c')$ are computed, where c, c' are counters on the number of pairs with the same keyword and the same file identifier $label_f$, respectively.

A data item is stored encrypted into $\hat{S}_1$ with a label l_p as,

$$\hat{S}_1(l_p) = ((label_f|l'_p) \oplus H(\hat{K}_\omega^{(2)}|r), r)$$

where r is a random string of length λ. Similarly, an entry of $\hat{S}_2$ is encrypted and stored with label l'_p as,

$$\hat{S}_2(l'_p) = ((l_p|'1') \oplus H(\hat{K}_f^{(2)}|r'), r')$$

where r' is a random string of length λ and $'1'$ indicates that the keyword/file identifier pair has not been read yet and it is stored in $\hat{S}_1$ with label l_p. However, when the keyword ω is searched, as in the simple scheme, the entry of $\hat{S}_2$ is modified and it is stored again unencrypted with label $label_f$ as,

$$\hat{S}_2(label_f) = (\omega|i|'0')$$

where $'0'$ indicates that the pair has been read and it is i-th item stored in S_0 with label ω.

The operations of the DSSE appear in Fig. 6, Fig. 7, and Fig. 8. More precisely:

SetUp operation. The client chooses four random λ-bit strings K_F, K_W, K, $\hat{K}$ for the two ORAM structures, the

files encryption (using the SKE $Gen(1^\lambda)$) and the management of $\hat{S}_1$ and $\hat{S}_2$. The client initiates the two ORAM structures $ORAM_\omega$ and $ORAM_f$ of size $|W|$ and $|F|$ and fills in $\hat{S}_1$ and $\hat{S}_2$ with the inverted index for keyword and the file index respectively. Initially, the structures S_0, S_1 and S_2 are empty.

Search operation. The search for keyword's ω file identifiers I_ω is performed in three steps. The first and second step are the same as in the simple scheme, i.e. the structures S_0 and S_1 are searched. In the third step, $K_\omega^{(1)}$ is used to compute the labels in $\hat{S}_1$ and $K_\omega^{(2)}$ to decrypt the corresponding entries. This third step is also parallizable. As always, S_0 is updated with the file identifiers from $\hat{S}_1$.

Add operation. Exactly the same as in the simple version of the DSSE. Only the structures S_1 and S_2 are used.

Delete operation. We distinguish two cases. If the file f was inserted during the setup phase and it was never deleted, then $\hat{S}_2$ is used to track the pairs stored in S_0 and $\hat{S}_1$. The client retrieves from $ORAM_f$ the file identifier $label_f$ and computes the two keys $\hat{K}_f^{(1)}$ and $\hat{K}_f^{(2)}$ and sends them to the server. The server computes the related labels used in $\hat{S}_2$ and decrypts the entries. Then, the entries are removed as well as the corresponding entries in S_0 and $\hat{S}_1$.

If the file was added in a later time, i.e. after the setup phase, the delete operation is the same as in the simple case using S_2 to retrieve the file index.

5.1 A Toy Example

In this section, we present a toy example to explain mainly the differences between the extended and the simple DSSE schemes. Let again $\{f_1, f_2, f_3\} \in F$ and $\{\omega_1, \omega_2, \omega_3, \omega_4\} \in W$. The initial keyword sequences are: $W_{f_1} = \{\omega_1, \omega_2\}$, $W_{f_2} = \{\omega_1, \omega_3\}$, and $W_{f_3} = \{\omega_4\}$. In Fig. 9 you can see the initial state of the DSSE. The structures S_0, S_1 and S_2 are empty and do not appear in the figure. In Fig. 10, we can see the state of the DSSE after searching for ω_1 and after adding f_4 with $W_{f_4} = \{\omega_1, \omega_2\}$. The addiction is the same as in the simple version of the DSSE. Regarding the search for ω_1, the client computes the keys $K_{\omega_1}^{(1)}$ and $K_{\omega_1}^{(2)}$ and sends them with ω to the server. The server computes the labels $l_1 = H(K_{\omega_1}^{(1)}|1)$ and $l_4 = H(K_{\omega_1}^{(1)}|2)$ (computes until an empty position is reached), xors the entries with $H(K_{\omega_1}^{(2)}|1)$ and $H(K_{\omega_1}^{(2)}|1)$ and retrieves the set $I_{\omega_1} = \{l_{f_1}, l_{f_2}\}$. Since S_0 is initially empty, I_{ω_1} is ready. Then, it deletes the two entries from $\hat{S}_1$, adds two new items with the same label ω_1 into S_0 and updates the corresponding items in $\hat{S}_2$.

6. COMPLEXITY ANALYSIS

In our scheme, we make a black-box use of the ORAM algorithm. Thus, we can use any of the proposed solutions. For the complexity evaluation, we consider two of the most asymptotically efficient ORAMs. The first one was introduced by Kushilevitz et al. ([12]) and has hierarchical structure. This scheme has communication overhead $O(\log^2 M/\log\log M)$ blocks and constant client storage, where M is the number of data blocks. Its storage requirements are $O(M)$. The second ORAM is the tree-based construction, known as the Path-ORAM scheme ([18]). This scheme has communication overhead $O(\log^2 M/\log\chi)$ blocks, where χ equals the block size divided by $\log M$ and small client storage $O(\log M)$, where M is the number of data blocks. Its storage requirements are again $O(M)$. Improved implementations of the Path ORAM algorithm were recently proposed (for instance [15], [22]).

Algorithm $SetUp(DB)$

1. $K \xleftarrow{\$} SKE.Gen(1^\lambda)$; $\hat{K} \xleftarrow{\$} \{0,1\}^\lambda$
2. $K_F \xleftarrow{\$} \{0,1\}^\lambda$; $K_W \xleftarrow{\$} \{0,1\}^\lambda$
3. Initiate two ORAMs of size $|W|$ and $|F|$
4. For each $\omega \in W$:
   $K_\omega \xleftarrow{\$} \{0,1\}^\lambda$; $c = I_\omega$
   $ORAM_\omega.write(\omega, (K_\omega, c))$
5. For each $f \in F$:
   Encrypt f: $f' = SKE.Enc(K, f)$
   $label_f \xleftarrow{\$} \{0,1\}^\lambda$
   Add $(label_f, f')$ to list C
   $ORAM_f.write(f, label_f)$
   $\hat{K}_f^{(1)} = H(\hat{K}|1|f)$;$\hat{K}_f^{(2)} = H(\hat{K}|2|f)$;
   $i = 0$
   For each $\omega \in W_f$:
      $i++$
      $\hat{K}_\omega^{(1)} = H(\hat{K}|1|\omega)$; $\hat{K}_\omega^{(2)} = H(\hat{K}|2|\omega)$
      $p_f = H(\hat{K}_f^{(1)}|i)$; $p_\omega = H(\hat{K}_\omega^{(1)}|i)$
      $r_1 \xleftarrow{\$} \{0,1\}^\lambda$;$r_2 \xleftarrow{\$} \{0,1\}^\lambda$
      $\hat{S}_1.insert(p_\omega, ((label_f|p_f) \oplus H(\hat{K}_\omega^{(2)}|r_1), r_1))$
      $\hat{S}_2.insert(p_f, ((p_\omega|'1') \oplus H(\hat{K}_f^{(2)}|r_2), r_2))$
6. Reshuffle C //the list is reshuffled to randomize the output
7. Output C, $\hat{S}_1$, $\hat{S}_2$, S_1, S_2

Figure 6: The SetUp protocol of the extended DSSE scheme

Search Complexity. Both the simple and the extended version require two ORAM operations and $O(|I_\omega|)$ communication cost for the sending the file identifiers. Thus, the search complexity is $O(\max(\log^2(|W|)/\gamma, |I_\omega|))$ where γ depends on the ORAM scheme that we use.

Add and delete file complexity. Adding a file requires one $ORAM_f$ write to store the new file label. Then, for each keyword $\omega_j \in W_{f_i}$ two $ORAM_\omega$ operations (read and write) are needed and one label per keyword is send. Thus, the file addition complexity is $O(|W_f|(\log^2(|W|)/\gamma + 1) + \log^2(|F|)/\gamma)$. On the other hand, a file deletion requires only one ORAM read. Thus, the pair deletion complexity is $O(\log^2(|F|)/\gamma)$.

Storage Complexity. The size of the two ORAMs are $O(|W|)$ and $O(|F|)$ respectively. The three (or five) dictionaries can have entries of fixed size and in total $O(N)$ are used. Thus, the storage required is $O(N)$, as N is bigger than $|W|$ and $|F|$.

Algorithm $Search(\omega)$
The client side

1. $(K_\omega, c) \leftarrow ORAM_\omega.read(\omega)$
2. $K'_\omega \xleftarrow{\$} \{0,1\}^\lambda$; $c' = 0$
3. $ORAM_\omega.write(\omega, (K'_\omega, c'))$
4. $\hat{K}^{(1)}_\omega = H(\hat{K}|1|\omega); \hat{K}^{(2)}_\omega = H(\hat{K}|2|\omega)$
5. Output $\omega, (\hat{K}^{(1)}_\omega, \hat{K}^{(2)}_\omega), (K_\omega, c)$

The server side

1. $I_\omega \leftarrow S0.get(\omega)$
2. If $I_\omega = \emptyset$, then
   For $c = 1$ until remove returns $\perp$
   $p_\omega \leftarrow H(K^{(1)}_\omega, i)$
   $block \leftarrow \hat{S}_1.remove(p_\omega)$
   $(data.l|data.l') \leftarrow (block.l|block.l') \oplus H(\hat{K}^{(2)}_\omega|block.r)$
   $I_\omega \leftarrow I_\omega \cup data.l$
   $j \leftarrow S0.insert(\omega, data)$
   $\hat{S}_2.update(data.l', (\omega|j|'0'))$
3. For $i = 1 : c$ do
   $p \leftarrow H(K_\omega|i)$
   $block \leftarrow S_1.remove(p)$
   $I_\omega \leftarrow I_\omega \cup block.l$
   $j \leftarrow S0.insert(\omega, block)$
   $S_2.update(block.l, block.j, (\omega|j|'0'))$
4. Output I_ω

Figure 7: The Search protocol of the extended DSSE scheme.

Parallelization and Locality. All the data structures can be accessed in parallel. Search can be performed with parallelism, as well as the add/delete file operations. Also, we have divided the stored data into two types: leaked and not leaked. The leaked data include the pairs that have been revealed after a search operation, as well as the connection between the label of a newly added file and the labels used to store related information. Thus, both S_0 and S_2 data structures can have high locality by storing data in contiguous area of memory positions, i.e. entries with the same label ω and $label_f$, respectively. Thus, we can replace dictionary reads with array reads.

7. SECURITY ANALYSIS

THEOREM 1. *If the used secret key encryption scheme is SKE is $IND-CPA$ secure, the ORAM algorithm is secure, and H is a random oracle, then our simple DSSE scheme is $(\mathcal{L}_{stp}, \mathcal{L}_{srch}, \mathcal{L}_{upd})$-secure against adaptive dynamic chosen keyword attacks.*

Algorithm $Delete(f)$
The client side

1. $label_f \leftarrow ORAM_f.read(f)$
2. $\hat{K}^{(1)}_f = H(\hat{K}|1|f); \hat{K}^{(2)}_f = H(\hat{K}|2|f);$
3. Output $(label_f, K^{(1)}_f, K^{(1)}_f)$

The server side

1. Delete the ciphertext with label $label_f$
2. $flag \leftarrow' new'$
3. $I_p \leftarrow S_2.removeAll(label_f)$
4. If $I_p == \emptyset$
   $flag \leftarrow' init'$
   For $i = 1 : c$ until remove returns $\perp$
   $p_f = H(K^{(1)}_f|i)$
   $block \leftarrow \hat{S}_1.remove(p_f)$
   $I_p \leftarrow I_p \cup block.l \oplus H(K^{(2)}_f|block.r)$
5. For $p \in I_p$ do
   If $p.s$=="0" then
   $S_0.remove(p.l, p.j)$
   else if $flag ==' new'$
   $S_1.remove(p.l)$
   else $\hat{S}_1.remove(p.l)$
6. Output "OK"

Figure 8: The Delete file protocol of the extended DSSE scheme

PROOF. (sketch) The polynomial time simulator goes as follows.

1. *Setting up the environment.* The simulator $\mathcal{S}$ creates an empty dictionary ρ to store the answers of the random oracle. Also, it creates a table T_W of $|W|$ entries filled with randomly chosen λ-bit strings. Finally, it creates an empty hash table T_f to store the current labels of the files, and two empty chained hash tables T_0 and T_2, i.e. tables of lists. In the simple DSSE scheme, it is easy to verify that the setup protocol can be composed of several application of the add file operation. The storage of the ciphertexts can be trivially simulated by encrypting all zero strings with the IND-CPA encryption scheme. It is left out of the proof (for instance see the proof in [9]).
2. *Simulating the Search protocol.* The value $k \leftarrow T_W[\omega]$ is retrieved and a new random is stored at $T_W[\omega]$. Also, the set I_w is leaked. Part of it is the list $I_0 \leftarrow T_0[\omega]$. The other part is retrieved from T_2. For each $l \in I_\omega \backslash I_0$, remove from the list $T_2[l]$ the item with ω, and add l in the list $T_0[\omega]$.
3. *Simulating the Add file protocol.* From the leakage function besides the type of the operation, the file

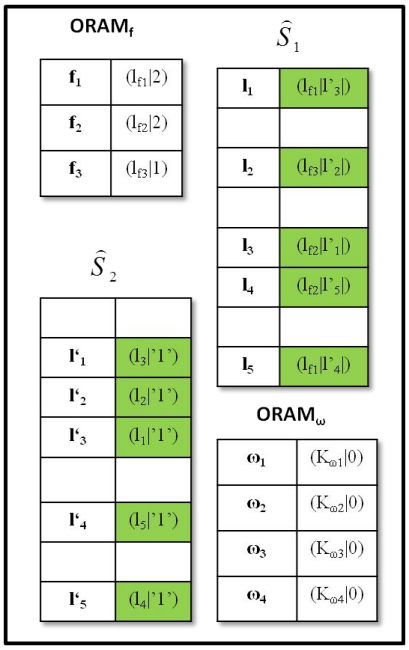


Figure 9: Toy example of the extended DSSE: initial state. The colored cells are encrypted.

f and the number of keywords $|W_f|$ are leaked. A label $label_f$ is chosen at random and it is stored at $T_f[f] \leftarrow label_f$. A list is created of $|W_f|$ items as follows. Among the $|W|$ keywords $|W_f|$ are chosen at random. Then, a counter is set $c \leftarrow 0$ and for each of the selected keywords the counter is increased by one and it is checked if $\rho[k][\omega][c]$ has a value, where k is the value at $T_W[\omega]$. If it has a value, then this value is added in the list with ω. Otherwise, a random label l is chosen, it is stored at $\rho[k][\omega][c]$ and the pair (l, ω) is added in the list. Finally, the list is stored at $T_2[lable_f]$.

4. *Simulating the Delete file protocol.* From the leakage function besides the type of the operation, the file f is leaked. The label $label_f$ is retrieved from $T_f[f]$. Then, all items in T_0 that contain $label_f$ are deleted. Also, the list $T_2[lable_f]$ is deleted.

□

THEOREM 2. *If the used secret key encryption scheme is SKE is $IND-CPA$ secure, the ORAM algorithm is secure, and H is a random oracle, then our extended DSSE scheme is $(\mathcal{L}_{stp}, \mathcal{L}_{srch}, \mathcal{L}_{upd})$-secure against adaptive dynamic chosen keyword attacks.*

PROOF. (sketch) For the simulation of S_0, S_1 and S_2 we work as above. The simulator has to simulate two more structures, i.e. $\hat{S}_1$ and $\hat{S}_2$. The two structures are similar to the ones used in [2]. Then, we can use an hybrid argument to show the output of the simulator is indistinguishable from the output of the DSSE. The proof is similar to the one used in [2]. □

The security of the scheme can be proved in the standard model with some extra communication overhead. We use

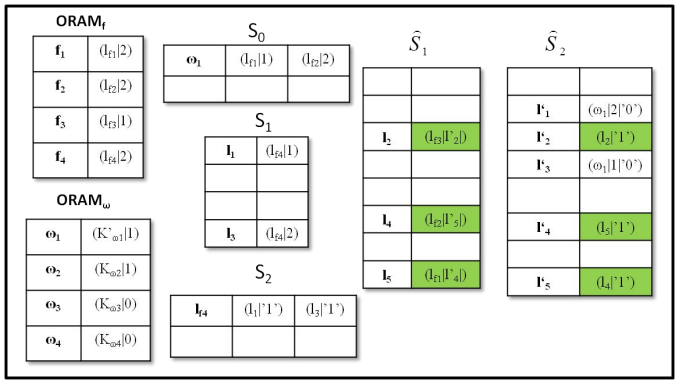


Figure 10: Toy example of the extended DSSE: final state. The colored cells are encrypted.

a pseudorandom function for the computation of the labels at S_1 and the client, instead of sending the current random key K of the keyword, he computes all the labels and then sends them to the server. The encryption scheme for the dictionary entries (for $\hat{S}_1$ and $\hat{S}_2$) is a one-time pad like, for instance the CTR mode with a random IV, and the client computes the encryption pads and send them to the server for the decryption.

8. DISCUSSION AND CONCLUSION

In this paper, we introduced two efficient DSSE schemes that achieve forward privacy with very small leakage. To the best of our knowledge only one scheme exists that offers the same level of security in terms of information leakage. Our schemes are ORAM based, and we demonstrated that with a limited use of an ORAM algorithm, we can achieve forward security and, at the same time, to minimize the overhead that this primitive introduces. In terms of search and update complexity, both our schemes depend only on the number of different keywords used and not on the total number of the keyword/file identifier pairs. That constitutes both our proposals optimal, when the number of keywords is fixed.

Our schemes can be easily further extended. One possible extension is to enrich the update operations. Add/delete operations for the keywords can be supported or even more fine-grained modifications at the level of a single pair. Also, since our schemes follow similar design philosophy with the SSE in [2], the implementation optimizations that are proposed in that paper can be applied. Finally, the data structures can be extended to perform complex Boolean queries on encrypted data via the OXT protocol ([1]).

9. REFERENCES

[1] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly scalable searchable symmetric encryption with support for Boolean queries. In CRYPTO, LNCS, vol. 8042, Springer, pages 353–373, 2013.

[2] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawcyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In the 21st Annual Network and Distributed System Security Symposium, NDSS, 2014.

[3] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In Applied Cryptography and Network Security (ACNS), vol.3531, LNCS, Springer, pages 442–455, 2005.

[4] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. Journal of the ACM, 45(6), 1998.

[5] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In Proceedings of the 13th ACM conference on Computer and communications security, CCS, pages 79–88, 2006.

[6] E. J. Goh. Secure indexes. IACR Cryptology ePrint Archive, 2003:216, 2013.

[7] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In STOC, pages 182–194, 1987.

[8] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In SODA, pages 157–167, 2012.

[9] F. Hahn, and F. Kerschbaum. Searchable Encryption with Secure and Efficient Updates. In the ACM conference on Computer and communications security, CCS, Scottsdale, Arizona, USA, 2014.

[10] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In the ACM Conference on Computer and Communications Security, pages 965–976, 2012.

[11] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In Financial Cryptography (FC), 2013.

[12] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In SODA, pages 143–156, 2012.

[13] R. Ostrovsky. Software Protection and Simulation On Oblivious RAMs. PhD thesis, Massachusetts Institute of Technology, 1992.

[14] B. Pinkas, and T. Reinman. Oblivious RAM Revisited. In CRYPTO, LNCS, Springer, vol. 6223, pages 502–519, 2010.

[15] L. Ren, C. Fletcher, A. Kwon, M. van Dijk, and S. Devadas. Unified Oblivious-RAM: Improving Recursive ORAM with Locality and Pseudorandomness. IACR Cryptology ePrint Archive, 2014:205, 2014.

[16] R. Sion and B. Carbunar. On the practicality of private information retrieval. In Proceedings of the Network and Distributed System Security Symposium, NDSS, 2007.

[17] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In IEEE Symposium on Security and Privacy, Washington, DC, USA, 2000.

[18] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In ACM Conference on Computer and Communications Security, CCS, pages 299–310, 2013.

[19] E. Stefanov, C. Papamanthou, and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In the Annual Network and Distributed System Security Symposium, NDSS 2014, California, USA, 2014.

[20] P. van Liesdonk, S. Sedghi, J. Doumen, P. H. Hartel, and W. Jonker. Computationally efficient searchable symmetric encryption. In Secure Data Management, pages 87–100, 2010.

[21] P. Williams, and R. Sion. Usable PIR. In NDSS 2008, San Diego, California, USA, 2008.

[22] X. Yu, L. Ren, C. Fletcher, A. Kwon, M. van Dijk, and S. Devadas. Enhancing Oblivious RAM Performance Using Dynamic Prefetching. IACR Cryptology ePrint Archive, 2014:234, 2014.