Symmetric Searchable Encryption with Sharing and Unsharing

Sarvar Patel[*] Giuseppe Persiano[†] Kevin Yeo[‡]

October 3, 2017

Abstract

We consider *Symmetric Searchable Encryption with Sharing and Unsharing* (SSEwSU), a notion that models Symmetric Searchable Encryption (SSE) in a multi-user setting in which documents can be dynamically shared and unshared among users. Previous works on SSE involving multiple users have assumed that all users have access to the same set of documents and/or their security models assume that all users in the system are trusted.

As in SSE, every construction of a SSEwSU will be a trade-off between efficiency and security, as measured by the amount of *leakage*. In multi-user settings, we must also consider cross-user leakage (*x-user leakage*) where a query performed by one user would leak information about the content of documents shared with a different user.

We start by presenting two strawman solutions that are at the opposite side of the efficiency-leakage bidimensional space: x-uz, that has zero x-user leakage but is very inefficient, and x-uL, that is very efficient but highly insecure with very large x-user leakage. We give a third construction, x-um, that is as efficient as x-uL and more efficient than x-uz. At the same time, x-um is considerably more secure than x-uL. Construction x-um is based on the concept of a *Re-writable Deterministic Hashing* (RDH), which can be thought of as a two-argument hash function with tokens that add re-writing capabilities. Sharing and unsharing in x-um is supported in constant (in the number of users, documents, and keywords) time. We give a concrete instantiation whose security is based on the Decisional Diffie-Hellman assumption. We provide a rigorous analysis of x-um and show a tight bound on the leakage in the presence of an active adversary that corrupts a subset of the users. We report on experimental work that show that x-um is very efficient and x-user leakage grows very slowly as queries are performed by the users.

Additionally, we present extensions of x-um. We modify x-um to support a finer grained access granularity, so a document can be shared to a user either only for reading (i.e., searching) or for writing (i.e., editing). We also extend x-um to the bilinear setting to further reduce leakage.

[*]Google, Inc., `sarvar@google.com`
[†]Google, Inc. and Università di Salerno, `giuper@gmail.com`
[‡]Google, Inc., `kwlyeo@google.com`

Contents

1 Introduction **3**

2 Symmetric Searchable Encryption with Sharing and Unsharing **5**

2.1 Security definition . . . 6

3 A construction based on DDH **7**

3.1 The leakage function $\mathcal{L}$ . . . 10

3.2 Simulator . . . 13

3.3 Security of un-sharing . . . 13

3.4 Search and Editing rights . . . 14

3.5 Further reducing the leakage . . . 15

4 Experiments **15**

4.1 Performance . . . 17

4.2 Leakage Growth . . . 17

A Re-writable Deterministic Hashing **19**

A.1 Regular instances . . . 20

A.2 A regular RDH with minimum leakage . . . 22

A.3 A construction based on regular RDH . . . 25

A.4 Simulator . . . 26

1 Introduction

Symmetric Searchable Encryption (SSE) [13] has been the object of intensive research and study in the last years. The original scenario consists of a Manager that has a corpus $\mathcal{D}$ of documents. Manager wishes to store $\mathcal{D}$ on Server in encrypted form. At later times, Manager would like to be able to provide Server with appropriate search tokens to select the ids (pointers) of the documents that contain given keywords. SSE is symmetric (or private-key) in the sense that only the Manager can encrypt the documents. The security consists of protecting the Manager (and his corpus $\mathcal{D}$) from a curious Server.

We introduce the concept of a *Symmetric Searchable Encryption with Sharing and Unsharing* (SSEwSU) where each document is *shared* by a subset of the users. Sharing is *dynamic* and a document can be subsequently *unshared* from a user. Such scenarios rise naturally in large organizations where different subsets of members of the organization collaborate on different documents at any give time. A document of $\mathcal{D}$ is described by a tuple $(d, \mathsf{Kw}(d), \mathtt{meta}_d)$ consisting of a document id d (or a pointer to the location of the document), a list $\mathsf{Kw}(d)$ of keywords taken from the universe $\mathcal{W}$ of keywords (e.g., the English dictionary), and some metadata $\mathtt{meta}_d$ (e.g., title, snippet, creation time). Each user $u \in \mathcal{U}$ can access a subset $\mathsf{Access}(u)$ of the corpus $\mathcal{D}$. The subset $\mathsf{Access}(u)$ of documents shared with user u varies over time as documents can be added/removed. A search for keyword w performed by u returns all pairs $(d, \mathtt{meta}_d)$ such that $w \in \mathsf{Kw}(d)$ and d is currently in $\mathsf{Access}(u)$. Typically, the user looks at the metadata of the documents returned by the search to determine which document to download. Manager is not involved in searches and all users u can perform queries on documents in $\mathsf{Access}(u)$ (and only on those) without further intervention of Manager. Moreover, sharing and unsharing of documents should be efficient and require only the Manager and the Server to be active.

Every SSE exposes some information, called the *leakage*, about the document corpus and queries. The design of SSE is a balancing act between efficiency and security (that is, reducing leakage). In a SSEwSU, because of its multi-user nature, it is possible that information leaked by a query of one user is carried over across different users. We call this *x-user leakage* and pay special attention to it when analyzing the leakage of our constructions. In this paper, we discuss three constructions and concentrate on the one which presents a good balance of efficiency and security.

A construction with no x-user leakage. In a first construction, that we call x-uz (*zero x-u*), sharing is implemented on top of any single-user SSE (supporting addition/deletion of documents) using an independent instance for each user u. When a document d is shared with user u, d is added to the instance of single-user SSE of u and it is removed when the document is unshared from u. As it can be easily seen, x-uz has no x-user leakage since each user is independent. This construction requires space proportional to $\sum_{u\in\mathcal{U}} \sum_{d\in\mathsf{Access}(u)} |\mathsf{Kw}(d)| = O(|\mathcal{U}| \cdot |\mathcal{D}| \cdot |\mathcal{W}|)$ which is very inefficient. The next constructions try to get an efficient solution with limited x-user leakage.

An efficient construction. Following a dual approach, in a second construction, that we call x-uL (*Large x-u*), we have an independent instance of single-user SSE for each document d. The secret key K_d of the instance for document d is given to all users u with access to d. To perform a search, user u sends Server a search token for each document $d \in \mathsf{Access}(u)$. The per-document partition is to ensure proper user access control. Indeed the Server stores, for each document d, the current list of users that have access to d and, upon receiving a search token for d, it checks if the user has access to d before actually performing the search. The amount of memory required by x-um is $O(|\mathcal{D}| \cdot |\mathcal{W}| + |\mathcal{D}| \cdot |\mathcal{U}|)$ space.

Construction x-uL suffers from extensive x-user leakage that accumulates as queries are being performed. For a simple example, suppose user u_1 performs a query for keyword w_1 by sending search tokens for each document in $\mathsf{Access}(u_1)$. The search tokens sent for a document d are generated by an algorithm using K_d. Any other user $u_2 \neq u_1$ with access to d, $d \in Access(u_2)$, would generate search tokens using the same algorithm and K_d. So, Server can use these search tokens for all users in $\mathsf{AccList}(d)$. So, Server infers information for every user $u_2 \neq u_1$ such that $d \in \mathsf{Access}(u_1) \cap \mathsf{Access}(u_2)$. In other words, each query extends the leakage to all the users, even to those users who never performed any query. If, subsequently, u_2 searches for w_2 then, for all of $\mathsf{Access}(u_1) \cap \mathsf{Access}(u_2)$, Server knows exactly those documents with w_1, w_2, both and neither.

Where does the problem come from? The x-user leakage of $\mathsf{x\text{-}uL}$ is due to two factors. First of all, the SSE is partitioned according to the document d so to allow the Server to enforce the access list. Secondly, query Q_1 and Q_2 of two different users, when restricted to the documents in the intersection, coincide. We see two contrasting needs fighting here: for the sake of efficiency, we record keyword occurrences in documents in a user-independent fashion but this forces the queries of different users to be the same. This has the effect that when one user queries, it does so on behalf of all the other users.

To go around this apparent stalemate, we introduce an intermediate level that helps remove the user-dependent part from a query to be matched with the user-independent encryption of the keyword-document pairs. The intermediate level will be implemented by means of tokens that depend on the user and the document. The absence of a token for a user and a document prevents a user for searching on the document thus effectively implementing access control. The introduction of the extra level requires space $O(|\mathcal{D}| \cdot |\mathcal{U}|)$, that is proportional to the size of the access lists, thus keeping overall efficiency.

An efficient construction with better leakage. The third construction, $\mathsf{x\text{-}um}$ (*minimal x-u*), is based on *Re-writable Deterministic Hashing* (RDH). RDH is an enhancement of a hash function with the possibility of constructing tokens that can maul hash values. Roughly speaking, we consider a two-argument hash function H. With slight abuse of terminology, we call *ciphertext* the value obtained by evaluating H on two arguments, which we refer to as *plaintexts*. For any two plaintexts A and B, it is possible to construct a *token* $\mathtt{tok}_{A\to B}$ that, when applied to a ciphertext of A, returns a new ciphertext in which A is replaced with B and the other plaintext stays unchanged.

Roughly, $\mathsf{x\text{-}um}$ can be described as follows. $\mathsf{Manager}$ computes $\mathsf{H}(w, d)$ for every $w \in \mathsf{Kw}(d)$ exactly once independently of the number of users that can access d. So, $\sum_d |\mathsf{Kw}(d)|$ ciphertexts are produced. $\mathsf{Manager}$ also produces *authorization* token $\mathtt{tok}_{u\to d}$ for every $d \in \mathsf{Access}(u)$, thus $\sum_u |\mathsf{Access}(u)|$ tokens are produced. All computed tokens and ciphertexts are given to Server for total space $O(|\mathcal{D}| \cdot |\mathcal{W}| + |\mathcal{D}| \cdot |\mathcal{U}|)$. A search is then performed in the following way: user u computes *query* ciphertext $\mathsf{H}(u, w)$ and a pointer to $\mathtt{tok}_{u\to d}$ and sends it to Server. Server applies the $\mathtt{tok}_{u\to d}$ to the query ciphertext. If u has access to d and d contains w, Server has $\mathtt{tok}_{u\to d}$ and the application to the query ciphertext $\mathsf{H}(u, w)$. produces $\mathsf{H}(w, d)$. If Server finds $\mathsf{H}(w, d)$ among the ciphertexts received from $\mathsf{Manager}$, then $w \in \mathsf{Kw}(d)$. The rewriting capability of the tokens makes it unnecessary to duplicate the pair (w, d) for each user that has access to document d (greatly improving on the efficiency of $\mathsf{x\text{-}uz}$). When a match is found as a result of a search of user u_1, nothing is leaked about user u_2 until u_2 queries for the *same* keyword. The actual construction involves the use of pseudorandom functions to prevent dictionary attacks (since H is deterministic and keywords and documents come from a potentially small universe). Encoding the access control into tokens dispenses the need to keep independent SSE for each user like $\mathsf{x\text{-}uL}$.

Our contribution. Our main contribution is a general construction of SSEwSU based on RDH. We present a notion of security for RDH and show that this notion is sufficient for an implementation of SSEwSU in which Server uses $\sum_{d\in\mathsf{Access}(u)} |\mathsf{Kw}(d)| + \sum_u |\mathsf{Access}(u)|$ space (thus matching the leaky construction $\mathsf{x\text{-}uL}$). Our construction supports sharing and unsharing of document d in constant time. Both operations can be carried out by the $\mathsf{Manager}$ or by any $u \in \mathsf{AccList}(d)$ by asking Server to update its internal data structure. No other user is affected. Searches can be carried out by each user u without requiring $\mathsf{Manager}$ and in time proportional to $|\mathsf{Access}(u)|$. We extend the scheme to allow two levels document access: *search* access (users are allowed to search for keywords) and *edit* access (users are allowed to add/remove keywords).

Our formal security setting follows the simulation-based abstraction of *real* vs. *simulated* games. We consider an active adversarial Server that corrupts a subset of users, $\mathcal{C}$, (thus gaining access to all the information of the users in $\mathcal{C}$, including the documents the users have access to) and show that its view can be simulated given a precisely defined leakage profile on the corpus $\mathcal{D}$ and search queries.

We propose a construction of RDH whose security is derived from the Decisional Diffie-Hellman (DDH) assumption. The instantiation of RDH under DDH is presented in the main body. We perform extensive experimentation that show our proposal is practical. With more experiments, we show that the x-user leakage for $\mathsf{x\text{-}um}$ is significantly slower compared to $\mathsf{x\text{-}uL}$ using real world data. Construction $\mathsf{x\text{-}um}$ hits the middle ground between the two extremes by providing the same efficiency as $\mathsf{x\text{-}uL}$ while reducing the leakage to an acceptable level. We also extend $\mathsf{x\text{-}um}$ to further reduce the leakage by going to a bilinear setting.

Related work. The notion of a *Symmetric Searchable Encryption* was introduced in [13] (in [1] for the asymmetric case) and is an active research area. The original scenario consists of two parties that, in our terminology, are Manager wishing to store an encrypted version of his data on Server and being able to delegate searches to Server while protecting the privacy of the data and queries. This basic setting was extended in [4], which considered the extension of multiple users authorized by Manager. The same setting is considered in several subsequent papers (for example, [2, 3, 5, 7]). We mention the work in [2] was the first to obtain sub-linear time for more general Boolean queries thus extending [4]. The recent work of [8] extends this by providing a sub-linear time construction for all Boolean formulae. In the works cited above, "multiple users" means users (other than the Manager) can perform searches using tokens provided by the Manager. However, all users have access to the same set of documents. We are interested in allowing different users access to different subsets of the documents. A public-key multi-user setting with restricted access control structures was considered in [12]. The concept of a Multi-Key Searchable Encryption [11] is closer to our setting. When used in the context of SSEwSU, Multi-Key Searchable Encryption would yield searches with performance similar to our construction (that is, a user performing a search transfers a token of size proportional to the number of document he has access to) but does not support efficient dynamic sharing and unsharing of documents. The concept of RDH was implicit in [2] and was used to reduce the cost of conjunctive keyword search.

The Database Management System (DBMS) of [9], even though much wider in scope, is also related to our work. In the presence of many users, the focus is on security from unauthorized users. In this respect, it differs from single-user SSE where users are trusted. The natural extension of [9] to multiple users, allowing each user access to the whole database, requires all users to share a secret key hidden from Server. If Server corrupts a single user, all security is lost. In our construction, corruption of a user leaks only documents accessible by the corrupted user. Earlier proposals [10] for DBMS were similar to single-user SSEs as the threat scenario focused on security against the curious Server.

Roadmap. Section 2 presents the concept of a SSEwSU and define its security notion. Section 3 describes an implementation of SSEwSU in cyclic groups for which DDH is conjectured to hold. We identify the leakage $\mathcal{L}$ that can be learned by Server when it corrupts a subset of users, $\mathcal{C}$, in Section 3.1. We postpone the proof of security to the appendix. Section 3.4 describes how to extend the basic construction to support finer grained access control where users can have only search rights to a document or editing rights. We present a modified construction of x-um using bilinear pairings in Section 3.5, which reduces x-user leakage. In the appendix, we consider RDH and show how to construct SSEwSU from an RDH. Specifically, the notion of RDH and its security notion is presented in Appendix A. An implementation of RDH is described and proved under the DDH assumption in Appendix A.2. Section A.3 presents a general construction of SSEwSU using RDH. The construction of x-um in Section 3 is obtained by plugging a special class of secure RDH. Section A.4 shows that no more information is leaked to an adversarial Server by giving a simulator, which, on input of $\mathcal{L}$, outputs a simulated transcript that is indistinguishable from the real view of Server.

2 Symmetric Searchable Encryption with Sharing and Unsharing

We introduce the concept of a *Symmetric Searchable Encryption with Sharing and Unsharing* (SSEwSU) and give an implementation based on the DDH assumption. A SSEwSU is a collection of six algorithms: EncryptDoc, Enroll, SearchQuery; SearchReply, AccessGranting, AccessRevoking for three types of players: one Manager, one Server, and several users. The algorithms interact in the following way.

1. Manager has corpus $\mathcal{D}$ consisting of triplets $(d, \mathsf{Kw}(d), \mathtt{meta}_d)$ of document id d, the set of keywords $\mathsf{Kw}(d)$ in document d, and document metadata $\mathtt{meta}_d$. Manager computes an encrypted version xSet of $\mathcal{D}$ and a master key K by running algorithm $\mathsf{EncryptDoc}(\mathcal{D})$. Manager sends xSet to Server and keeps K in private memory. Manager instructs Server to initialize uSet to be empty. Both xSet and uSet are kept private by Server.

2. Manager executes Enroll to add a new user u to the system. Keys $\mathcal{K}_u$ for u are returned by the algorithm. Manager stores the pair $(u, \mathcal{K}_u)$ in private memory and sends $\mathcal{K}_u$ to u. When a user is enrolled, they do

not have any access rights, that is $\mathsf{Access}(u) = \emptyset$.

3. To share document d with user u (that is, $\mathsf{Access}(u) := \mathsf{Access}(u) \cup \{d\}$), $\mathsf{Manager}$ executes $\mathsf{AccessGranting}$ on input the pair $(u, \mathcal{K}_u)$, document id d and master key K. $\mathsf{AccessGranting}$ outputs *authorization token* $U_{u,d}$ for u and d and keys $\mathcal{K}_d$ for document d. $U_{u,d}$ is given to Server for inclusion to uSet and $\mathcal{K}_d$ is given to user u.

   $\mathsf{AccessRevoking}$ is used in a similar way by $\mathsf{Manager}$ to revoke user u's access to document d. Instead, $U_{u,d}$ is given to Server to remove from uSet.

4. To search for all documents d in $\mathsf{Access}(u)$ that contain keyword w, user u executes $\mathsf{SearchQuery}$ on input w, $\mathcal{K}_u$ and $\{(d, \mathcal{K}_d\}_{d \in \mathsf{Access}(u)}$ to construct the *query* qSet that is passed onto Server.

5. On input qSet, Server runs $\mathsf{SearchReply}$ using xSet and uSet to compute Result, which is returned to u. Using the correct keys, u decrypts Result and obtains the ids and metadata of documents in $\mathsf{Access}(u)$ which contain w.

We denote the set of users with access to document d by $\mathsf{AccList}(d)$. So, $d \in \mathsf{Access}(u)$ if and only if $u \in \mathsf{AccList}(d)$. With slight abuse of notation, for a subset $\mathcal{C}$ of users, $\mathsf{Access}(\mathcal{C})$ is the union of $\mathsf{Access}(u)$ for $u \in \mathcal{C}$.

The definition is tailored for a static corpus of documents (no document is added and/or edited). This is reflected by the fact that $\mathsf{Manager}$ computes the encrypted version of the corpus by using $\mathsf{EncryptDoc}$ at the start. Section 3.4 discusses how to extend the construction to have users with editing rights that can add/remove keywords from document. Users can also create a new document, d, where they will, initially, be the only member of $\mathsf{AccList}(d)$.

2.1 Security definition

We give our security definition for SSEwSU by following the real vs. simulated game approach. In our trust model, $\mathsf{Manager}$ is honest since he owns the corpus $\mathcal{D}$ whose privacy we wish to protect. We assume Server is not malicious and computes query results as prescribed by $\mathsf{SearchReply}$, stores xSet as received from $\mathsf{Manager}$, and updates uSet instructed by $\mathsf{Manager}$. A malicious Servercan omit answers to queries or reply arbitrarily. Little can be done against DOS attacks from a malicious Server. Checking that Server replies correctly to queries is beyond the scope of our work. We assume that Server is curious with access to $\mathsf{xSet}, \mathsf{uSet}$ and all queries. In the multi-user setting, we have to consider that Server might corrupt a subset $\mathcal{C}$ of users gaining access to the keys of users in $\mathcal{C}$.

In the real game with a set of users $\mathcal{U}$, we consider the *server view*, $\mathsf{sView}^{\mathcal{U},\mathcal{C}}$. Here, Server corrupts the subset $\mathcal{C} \subseteq \mathcal{U}$ of users gaining access to the set of keys $\{K_u\}_{u \in \mathcal{C}}$ and $\{\mathcal{K}_d\}_{d \in \mathsf{Access}(\mathcal{C})}$. We assume no revocation (unsharing) is made as a curious Server may keep all authorization tokens $U_{u,d}$ provided. All queries are assumed to be performed after all sharing operations as a curious server can always postpone or duplicate the execution of a query. The view is relative to a snapshot of the system (that is xSet and uSet) resulting from a sequence of sharing operations by $\mathsf{Manager}$ whose cumulative effect is encoded in $\mathsf{Access}(u)$ for all $u \in \mathcal{U}$. We define an *instance* of SSEwSU, $\mathcal{I} = \{\mathcal{D}, \{d, \mathsf{Kw}(d), \mathtt{meta}_d\}_{d \in \mathcal{D}}, \{\mathsf{Access}(u)\}_{u \in \mathcal{U}}, \{(u_i, w_i)\}_{i \in [q]}\}$, consisting of

1. a set of documents $\{d, \mathsf{Kw}(d), \mathtt{meta}_d\}_{d \in \mathcal{D}}$;
2. collection $\{\mathsf{Access}(u)\}_{u \in \mathcal{U}}$ of subsets of document ids;
3. the set of search queries $Q_i = (u_i, w_i)$, for $i \in [q]$; i-th query is performed by user u_i for keyword w_i.

We define the view with respect to security parameter λ of Server when corrupting set $\mathcal{C}$ of users on instance $\mathcal{I}$ of SSEwSU, which is the output of experiment $\mathsf{sView}^{\mathcal{U},\mathcal{C}}(\lambda, \mathcal{I})$.

```
$\mathsf{sView}^{\mathcal{U},\mathcal{C}}(\lambda, \mathcal{I})$
 1. Set $(\mathsf{xSet}, \mathcal{K}) \leftarrow$
        $\mathsf{EncryptDoc}(1^\lambda, \mathcal{D}, \{d, \mathsf{Kw}(d), \mathtt{meta}_d\}_{d\in\mathcal{D}})$;
 2. Set $\{K_u\}_{u\in\mathcal{U}} \leftarrow \mathsf{Enroll}(1^\lambda, \mathcal{U})$;
 3. Set $(\mathsf{uSet}, \{\{\mathcal{K}_d\}_{d\in\mathsf{Access}(u)}\}_{u\in\mathcal{U}}) \leftarrow$
        $\mathsf{AccessGranting}(\{K_u\}_{u\in\mathcal{U}}, \{\mathsf{Access}(u)\}_{u\in\mathcal{U}}, \mathcal{K})$;
 4. For each $i \in [q]$
        $\mathsf{qSet}_i =$
          $\mathsf{SearchQuery}(w_i, K_{u_i}\{(d, K_d, K_d^{\mathrm{enc}})\}_{d\in\mathsf{Access}(u_i)})$;
        $\mathsf{Result}_i = \mathsf{SearchReply}(\mathsf{qSet}_i)$;
 3. Output $(K_u, \mathcal{K}_u)_{u\in\mathcal{C}}, \mathsf{xSet}, \mathsf{uSet}, (\mathsf{qSet}_i, \mathsf{Result}_i,)_{i\in[q]}$;
```

View of Server corrupting $\mathcal{C}$ for instance $\mathcal{I}$

We slightly abuse notation by passing a set of values as a parameter to an algorithm instead of a single value. By this, we mean that the algorithm is invoked on each value of the set received and that outputs are collected and returned as a set. For example, "$\mathsf{Enroll}(1^\lambda, \mathcal{U})$" denotes the sequential invocation of algorithm Enroll on input $(1^\lambda, u)$ for all $u \in \mathcal{U}$.

Definition 1. *We say that a* SSEwSU *is* secure with respect to leakage $\mathcal{L}$ *if there exists an efficient simulator* $\mathcal{S}$ *such that for every coalition* $\mathcal{C}$ *and every instance* $\mathcal{I}$

$$\{\mathsf{sView}^{\mathcal{U},\mathcal{C}}(\lambda, \mathcal{I})\} \approx_c \{\mathcal{S}(\lambda, \mathcal{L}(\mathcal{I}, \mathcal{C}))\}.$$

3 A construction based on DDH

In this section, we describe x-um, a construction of SSEwSU, based on the Decisional Diffie-Hellman Assumption. Our construction uses Server storage that is proportional to the number $\sum_u |\mathsf{Access}(u)|$ of pairs (u, d) with $d \in \mathsf{Access}(d)$ and the number $\sum_{d\in\mathcal{D}} |\mathsf{Kw}(d)|$ of pairs (d, w) with $w \in \mathsf{Kw}(d)$. Manager only needs space proportional to the number of enrolled users and each user needs space proportional to the number of documents they have access to. A query, produced by user u, has size proportional to $|\mathsf{Access}(u)|$. Finally, we mention that access granting (sharing) and revocation (unsharing) take constant time and only require Server and Manager to collaborate. Additionally, a user with access can provide access to another user while interacting with Server without Manager.

The security proof is postponed to the appendix and obtained as part of a more general framework. We show that RDH (see Appendix A) can be used to construct SSEwSU. A construction of RDH under the DDH Assumption is presented. The construction of x-um is a special case of the general construction and its security follows from the general construction.

Decisional Diffie-Hellman Assumption. A *group generator* $\mathcal{GG}$ is an efficient randomized algorithm that on input 1^λ outputs the description of a cyclic group $\mathcal{G}$ of prime order p for some $|p| = \Theta(\lambda)$ along with a generator g for $\mathcal{G}$. We say that the *Decisional Diffie-Hellman* assumption holds for group generator $\mathcal{GG}$ if distributions D_λ^0 and D_λ^1 are computational indistinguishable, where

$$D_\lambda^\xi = \left\{(g, \mathcal{G}) \leftarrow \mathcal{GG}(1^\lambda); x, y, r \leftarrow \mathbb{Z}_{|\mathcal{G}|} : (g^x, g^y, g^{x\cdot y+\xi\cdot r})\right\}.$$

In addition to assuming the DDH Assumption, our construction uses the following three cryptographic primitives.

1. a pseudorandom family of functions $\{\mathsf{G}(K, \cdot)\}_{K\in\{0,1\}^\star}$ such that, for each K of length λ, function $\mathsf{G}(K, \cdot)$ takes λ-bit long inputs and returns strings of length λ;

2. a pseudorandom family of functions $\{\mathsf{F}(K, \cdot)\}_{K\in\{0,1\}^\star}$ such that, for each K of length λ, function $\mathsf{F}(K, \cdot)$ takes λ-bit long inputs and returns elements of $\mathbb{Z}_p$, for p prime of length $\Theta(\lambda)$;

3. a key-oblivious CPA secure private-key encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$.

All of our cryptographic primitives are known to exist under the DDH Assumption.
An informal description. We start by describing a simple version that does not offer adequate security. Assume that all document ids, d, user ids, u, and keywords, w, are mapped to elements of a group. The occurrence of $w \in \mathsf{Kw}(d)$ is encoded by Manager by computing the x-pair, consisting of the product $w \cdot d$ and of an encryption of $\mathtt{meta}_d$. All x-pairs are given to Server. The fact that $u \in \mathsf{AccList}(d)$ is encoded by computing *authorization token* $d \cdot u^{-1}$ and giving it to Server. The set of all x-pairs and authorization tokens produced by Manager are called the xSet and uSet respectively. To search for a keyword w in $\mathsf{Access}(u)$, user u produces *query* $u \cdot w$. Server multiplies the query by the corresponding authorization token. If the result appears as a first component of an x-pair, the second component is returned to the user to decrypt. Correctness is obvious but very weak security is offered. Suppose that two users u_1 and u_2 query for the same keyword w thus producing $\mathtt{qct}_1 = u_1 \cdot w$ and $\mathtt{qct}_2 = u_2 \cdot w$. Then the ratio $\mathtt{qct}_1/\mathtt{qct}_2$ can be used to turn an authorization token for u_1 to access document d into an authorization token for user u_2 for the same document. Indeed $d/u_1 \cdot \mathtt{qct}_1/\mathtt{qct}_2 = d/u_2$, so the server can extend queries of u_2 to $\mathsf{Access}(u_1) \cup \mathsf{Access}(u_2)$.

So, we move to a group where DDH is conjectured to hold. Consider x-pairs consisting of an x-*ciphertext* computed as $g^{w \cdot d}$ along with a y-ciphertext that is an encryption of $\mathtt{meta}_d$. Authorization tokens are computed in the same way as before. A *query* is computed as $g^{u \cdot w}$ and ids to authorization tokens. In performing the search, Server uses the authorization tokens as an exponent for the query ciphertext (that is, $g^{u \cdot w}$ is raised to the power d/u). The value obtained is looked up as the first component of an x-ciphertext. If found, the associated y-ciphertext is returned. Using the exponentiation one-way function in a group in which DDH is conjectured to hold (and thus partial information is hidden) may not suffice as the set of documents and keywords might be small enough for Server to conduct a dictionary attack. We shall use pseudorandom functions computed by using document and user keys. Keys are distributed to whether a document is shared with a user. The main technical difficulty is to prove that DDH and pseudorandomness are sufficient to limit the leakage obtained by Server that has corrupted a subset of users. This means Server has gained access to the xSet and uSet for documents shared to at least one corrupted user. Server also knows the query patterns of all users.

We are now ready to formally describe x-um. In our construction, algorithms AccessGranting and AccessRevoking use AuthComputing. For user u, on input of user keys $K_u, \widetilde{K}_u$, document id d, and master keys K_1, K_2, K_3, returns u's authorization token $U_{u,d}$ to access document d, identifier $\mathtt{uid}_{u,d}$ and the set of keys $\mathcal{K}_d$ for document d. Algorithm AccessGranting is executed by Manager to grant user u access to document d. It consists in running AuthComputing to obtain $\mathcal{K}_d$, that is sent to user u, and $\mathtt{uid}_{u,d}, U_{u,d}$ that are sent to Server for insertion of $U_{u,d}$ at $\mathsf{uSet}[\mathtt{uid}_{u,d}]$. Algorithm AccessRevoking runs AuthComputing and sends $\mathtt{uid}_{u,d}$ to Server for deletion of $\mathsf{uSet}[\mathtt{uid}_{u,d}]$. Once $U_{u,d}$ has been removed from uSet, user u can still produce a query ciphertext $\mathtt{qct}_d$ for document d in the context of searching for keyword w but Server will not contribute y-ciphertext to Result even if $w \in \mathsf{Kw}(d)$.

```
$\mathsf{EncryptDoc}(1^\lambda, \mathcal{D})$
Executed by Manager to encrypt the corpus $\mathcal{D}$
  1. randomly select $(g, \mathcal{G}) \leftarrow \mathcal{GG}(1^\lambda)$ and
       initialize $\mathsf{xSet} = \emptyset$;
  2. randomly select three master keys
       $K_1, K_2, K_3 \leftarrow \{0,1\}^\lambda$;
  3. for every document $d$ with metadata $\mathtt{meta}_d$
       set $K_d = \mathsf{F}(K_1, d)$, $\widetilde{K}_d = \mathsf{F}(K_2, d), K_d^{\mathrm{enc}} = \mathsf{G}(K_3, d)$;
       for every keyword $w \in \mathsf{Kw}(d)$:
         set x-ciphertext $X_{w,d} = g^{\mathsf{F}(\widetilde{K}_d, d) \cdot \mathsf{F}(K_d, w))}$;
         set y-ciphertext $Y_{w,d} = \mathsf{Enc}(K_d^{\mathrm{enc}}, \mathtt{meta}_d)$;
  4. all pairs $(X_{w,d}, Y_{w,d})$ are added in random order to the array $\mathsf{xSet}$;
  5. return $(\mathsf{xSet}, K_1, K_2, K_3)$;
```

```
$\mathsf{Enroll}(1^\lambda, u)$
Executed by Manager to enroll user $u$
  1. randomly select user key $K_u, \widetilde{K}_u \leftarrow \{0,1\}^\lambda$;
  2. return $K_u, \widetilde{K}_u$;
```

```
$\mathsf{SearchQuery}(w, (u, K_u, \widetilde{K}_u), \{(d, K_d, K_d^{\mathrm{enc}})\}_{d \in \mathsf{Access}(u)})$
Executed by user $u$ to search for keyword $w$
  1. for each $(d, K_d, K_d^{\mathrm{enc}})$
       $(\mathtt{uid}_{u,d}, \mathtt{qct}_d) = (\mathsf{F}(\widetilde{K}_u, d), g^{\mathsf{F}(K_d, w) \cdot \mathsf{F}(K_u, d)})$;
  2. all query ciphertexts $(\mathtt{uid}_{u,d}, \mathtt{qct}_d)$ are added in random order to the array $\mathsf{qSet}$;
  3. return $\mathsf{qSet}$;
```

```
$\mathsf{SearchReply}(\mathsf{qSet})$
Server replying to $u$'s search query consisting of $s$ query ciphertexts
  1. set $\mathsf{Result} = \emptyset$;
  2. for each query ciphertext $(\mathtt{uid}_{u,d}, \mathtt{qct}_d) \in \mathsf{qSet}$
       set $\mathtt{ct} = \mathtt{qct}_d^{\mathsf{uSet}[\mathtt{uid}_{u,d}]}$;
       if $\mathsf{xSet}$ contains pair $(\mathtt{ct}, Y) \in \mathsf{xSet}$ then
           add $Y$ to $\mathsf{Result}$;
  3. return $\mathsf{Result}$;
```

```
$\mathsf{AuthComputing}((u, K_u, \widetilde{K}_u), d, (K_1, K_2, K_3))$
Executed by Manager to share document $d$ with user $u$
  1. compute $K_d = \mathsf{F}(K_1, d)$, $\widetilde{K}_d = \mathsf{F}(K_2, d)$, and $K_d^{\text{enc}} = \mathsf{F}(K_3, d)$;
  2. set authorization token $U_{u,d} = \mathsf{F}(\widetilde{K}_d, d)/\mathsf{F}(K_u, d)$;
  3. set authorization token id $\mathtt{uid}_{u,d} = \mathsf{F}(\widetilde{K}_u, d)$;
  4. set $\mathcal{K}_d = (d, K_d, K_d^{\text{enc}})$;
  5. return $(\mathtt{uid}_{u,d}, U_{u,d}, \mathcal{K}_d)$;
```

3.1 The leakage function $\mathcal{L}$

In this section, we formally identify the leakage $\mathcal{L}(\mathcal{I}, \mathcal{C})$ that Server obtains about the instance $\mathcal{I}$ from the view $\mathsf{sView}^{\mathcal{U},\mathcal{C}}(\lambda, \mathcal{I})$ when corrupting users in $\mathcal{C}$. In the security proof, we will show that nothing more than $\mathcal{L}$ is leaked by our construction by giving a simulator that, on input $\mathcal{L}$, simulates the entire view. Construction x-um and its proof are a special case of the *Regular* RDH based construction from Appendix A.3. For the sake of exposition, we outline the proof tailored for the construction based on DDH here.

Passive adversary. As a warm-up, we informally describe the leakage obtained by Server, starting from the case of passive Server (that is, $\mathcal{C} = \emptyset$ and no user is corrupted). We will then present a formal definition of leakage for the general case. By looking ahead, the leakage for $\mathcal{C} = \emptyset$ corresponds to items 0 and 7 in the formal definition leakage.

If $\mathcal{C} = \emptyset$, the Server observes xSet, uSet, the query ciphertexts and their interaction with xSet and uSet, including whether each query ciphertext is successful. The size $n := |\mathsf{xSet}|$ leaks the number of pairs (d, w) such that $w \in \mathsf{Kw}(d)$ and the size $m := |\mathsf{uSet}|$ leaks the number of pairs (u, d) such that $d \in \mathsf{Access}(u)$. Note, the xSet by itself does not leak any information about the number of keywords in a document or the number of documents containing a certain keyword (we will see, under the DDH, it is indistinguishable from a set of random group elements). The length of each query ciphertext leaks the number of documents the querier has access to. Note that leakage of (an upper bound on) the size of data is unavoidable.

The interaction of the query ciphertexts with xSet and uSet also leak some information. We set q to be the number of queries, and denote $l := \sum_{i\in[q]} n_{q_i}$ where $l_{\mathsf{q}_i} := |\mathsf{qSet}_i|$. A query ciphertext is uniquely identified by the triple (u, w, d) of the user u, the searched keyword w, and the document d for which the query ciphertext is searching.

Roughly speaking, we show that in x-um, Server only learns whether two query ciphertexts share two of three components. We assume that no user searches for the same keyword twice and so no two query ciphertexts share all three components. We remind the reader that in x-uL, Server would learn whether two queries are relative to the same document and this allowed the propagation of x-user leakage. In contrast, two query ciphertexts of two different users would only leak if they were for the same document and the same keyword in x-um. In other words, the only way to have x-user leakage is two users with at least a common document must perform a query for the same keyword.

A useful way to visualize the growth of x-user leakage is a graph G in which the users are vertices and a query of a user u_1 leaks about the documents of user u_2 if and only if u_1 and u_2 are in the same connected component. The larger the connected components in the graph, the more x-user leakage each query entails. For both constructions, the graph starts with no edges and edges are added as queries are performed. In x-uL, for every query of user u_1 for keyword w, an edge is added to all vertices of users u_2 that have at least one document in common with u_1, independently of w. In x-um, an edge is added to all vertices of users u_2 that have at least document in common with u_1 *and* have performed a query for keyword w. Thus, x-user leakage accumulates for every query in x-uL whereas in x-um x-user leakage grows slower and only accumulates across users for queries for repeated keywords.

Let us now explain where the leakage comes from. Consider two query ciphertexts, $(\mathtt{uid}_1, \mathtt{qct}_1)$ and

$(\mathtt{uid}_2, \mathtt{qct}_2)$, identified by (u_1, w_1, d_1) and (u_2, w_2, d_2), respectively. Start by observing that if $u_1 = u_2 = u$ and $w_1 = w_2 = w$, then $(\mathtt{uid}_1, \mathtt{qct}_1)$ and $(\mathtt{uid}_2, \mathtt{qct}_2)$ are part of the same query qSet issued by user u for keyword w. Thus, they can be easily identified as such by the Server. Next, consider the case in which $(\mathtt{uid}_1, \mathtt{qct}_1)$ and $(\mathtt{uid}_2, \mathtt{qct}_2)$ are queries from the same user and relative to the same document. That is, $u_1 = u_2 = u$ and $d_1 = d_2 = d$ but $w_1 \neq w_2$. This can be easily identified by the Server since $\mathtt{uid}_1 = \mathtt{uid}_2$. Note the leakage described so far is relative to queries from the same user. Suppose now that $(\mathtt{uid}_1, \mathtt{qct}_1)$ and $(\mathtt{uid}_2, \mathtt{qct}_2)$ are for the same document and the same keyword; that is, $w_1 = w_2 = w$ and $d_1 = d_2 = d$ but $u_1 \neq u_2$. In this case, when $\mathtt{qct}_1$ and $\mathtt{qct}_2$ are coupled with $U_{u_1,d}$ and $U_{u_2,d}$, respectively, they produce the same value (that belongs to xSet if and only if $w \in \mathsf{Kw}(d)$).

By summarizing, the leakage provides three different equivalence relations, denoted $\approx_d, \approx_w, \approx_u$, over the set $[l]$ of the query ciphertexts defined as follows. Denote by (u_i, w_i, d_i) the components of the generic i-th query:

1. $i \approx_d j$ iff $u_i = u_j$ and $w_i = w_j$; that is the i-th and the j-th query ciphertext only differ with respect to the document; we have q equivalence classes corresponding to the q queries performed by the users;

2. $i \approx_w j$ iff $u_i = u_j = u$ and $d_i = d_j = d$; that is the i-th and the j-th query ciphertext only differ with respect to the keyword. We denote by r the number of the associated equivalence classes $D_1, \ldots, D_r$. Equivalence class D_i can be seen of consisting of pairs of the index of a query ciphertext and the index of an x-ciphertext.

3. $i \approx_u j$ iff $w_i = w_j$ and $d_i = d_j$; that is the i-th and the j-th query ciphertext only differ with respect to the user; we denote by t the number of the associated equivalence classes $E_1, \ldots, E_t$. Equivalence class E_i can be seen of consisting of pairs of the index of a query ciphertext and a token.

Note that the equivalence classes of $\approx_w$ can be deduced from those of $\approx_u$ but we keep the two notions for clarity.

Active adversary. Consider the case where the adversarial Server corrupts a subset $\mathcal{C} \neq \emptyset$ of users. All information about documents shared to users in $\mathcal{C}$ are leaked to Server. For documents instead that are not accessible by users in $\mathcal{C}$, we fall back to the case of no corruption and the leakage is the same as described above.

In determining the leakage of our construction, we make the natural assumption that a user u knows all the keywords appearing in all documents $d \in \mathsf{Access}(u)$. This is justified by the fact that keywords are taken from a potentially small space and that u could search for all possible keywords in the document d (or u could just download d). If u is corrupted by Server, then we observe that Server is able to identify the entry of the xSet relative to (w, d) and the entry of uSet relative to (u, d) (this can be done by constructing an appropriate query ciphertext using the keys in u's possession). From these two entries, and by using the keys $K_u, \widetilde{K}_u, K_d$ and $\widetilde{K}_d$ in u's possession, $\mathsf{F}(\widetilde{K}_d, d)$, $\mathsf{F}(K_d, w), \mathsf{F}(\widetilde{K}_u, d)$ and $\mathsf{F}(K_u, d)$ can be easily derived. Moreover, we assume that the set $\mathsf{AccList}(d)$ of users with which d is shared is available to u. In this case, we make the assumption that for all $v \in \mathsf{AccList}(d)$, Server can identify the entry of uSet corresponding to $U_{v,d}$ from which the two pseudo-random values contributing to token $U_{v,d}$ can be derived. In general, we make the *conservative assumption* that knowledge of $\mathsf{F}(k, x)$ (or of any expression involving $\mathsf{F}(k, x)$) and k allows the adversarial Server to learn x by means of a dictionary attack; indeed in our construction the argument x of a PRF is either a keyword or a document id. In both cases, they come from a small space where dictionary attacks are feasible. We stress that these assumptions are not used in our construction (for example, honest parties are never required to perform exhaustive evaluations) but they make the adversary stronger thus yielding a stronger security guarantee. If this assumption is unsupported in a specific scenario, our security guarantees still hold and stronger guarantees can be obtained for the same scenario.

We remind the reader that the view of Server when corrupting users in $\mathcal{C}$ for instance $\mathcal{I}$, includes $(K_u, \widetilde{K}_u, D_u)_{u \in \mathcal{C}}$, where D_u is $\{d, K_d, K_d^{\mathsf{Enc}}\}_{d \in \mathsf{Access}(u)}$. Additional the view contains $\mathsf{xSet}, \mathsf{uSet}$ and the set $(\mathsf{qSet}_i, \mathsf{Result}_i)_{i \in [q]}$ of query ciphertexts and the results for each query. Without loss of generality, we assume that no two queries are identical (that is, from the same user and for the same keyword).

First, Server learns from the view n, the number of x-ciphertexts (and y-ciphertexts), m, the number of tokens, q, the number of queries, and l_{q_i}, the number of query ciphertexts for each query $i \in [q]$, and if each query is successful or not.

0) n, m, q and $n_{\mathsf{q}_i} = |\mathsf{qSet}_i|$ for $i \in [q]$.

In addition, we make the natural assumption that Server learns the following information regarding documents and queries for each user $u \in \mathcal{C}$.

1. the set $\mathsf{Access}(u)$ of documents that have been shared with $u \in \mathcal{C}$;
2. the set $\mathsf{Kw}(d)$ of keywords and the metadata $\mathtt{meta}_d$, for each $d \in \mathsf{Access}(\mathcal{C})$;
3. the set $\mathsf{AccList}(d)$ of users, for each $d \in \mathsf{Access}(\mathcal{C})$;
4. (u_i, w_i) for all $i \in [q]$ such that $u_i \in \mathcal{C}$;

Therefore, Server obtains keywords, metadata, and set of users that have access, for all documents that can be accessed by at least one Server corrupted user $u \in \mathcal{C}$. Moreover, Server also knows all queries issued by the corrupted users.

Consider x-ciphertext $X_{w,d} = g^{\mathsf{F}(\tilde{K}_d, d) \cdot \mathsf{F}(K_d, w)}$. If $d \in \mathsf{Access}(\mathcal{C})$, then w and d are available to Server by Points 1 and 2 above. Therefore, Server knows exactly all the entries of the xSet corresponding to documents in $\mathsf{Access}(\mathcal{C})$ and nothing more. This implies that if no query is performed, no information is leaked about documents not available to the members of $\mathcal{C}$.

More leakage is derived from the queries. Let us consider a generic query ciphertext,

$$\mathtt{uid}_{u_i,d} = \mathsf{F}(\widetilde{K}_{u_i}, d), \mathtt{qct}_d = g^{\mathsf{F}(K_d, w_i) \cdot \mathsf{F}(K_{u_i}, d)}$$

for document d produced as part of the i-th query qSet_i issued by user u_i for keyword w_i. If $u_i \in \mathcal{C}$, then K_d, K_{u_i} and $\widetilde{K}_{u_i}$ are available to Server and thus (u_i, w_i, d) is leaked. If $u_i \notin \mathcal{C}$ and $d \in \mathsf{Access}(\mathcal{C})$ then K_d is available (whence, by our conservative assumption, w_i is available too) but K_{u_i} and $\widetilde{K}_{u_i}$ are not available. In this case, d and w_i are leaked. We further observe that query ciphertexts from the same user $u_i \notin \mathcal{C}$ and document $d \in \mathsf{Access}(\mathcal{C})$ are easily clustered together since they all share exponent, $\mathsf{F}(K_{u_i}, d)$, and $\mathtt{uid}_{u_i,d}$, $\mathsf{F}(\widetilde{K}_{u_i}, d)$. We define $\hat{u}_i$ to be the smallest index $j \leq i$ such that $u_j = u_i$ and $d_j = d$. We say that if $u_i \notin \mathcal{C}$ and $d \in \mathsf{Access}(\mathcal{C})$, then $(\hat{u}_i, w_i, d)$ is leaked.

Suppose $u_i \notin \mathcal{C}$, $d \notin \mathsf{Access}(\mathcal{C})$ but $\mathsf{Access}(u_i) \cap \mathsf{Access}(\mathcal{C}) \neq \emptyset$; that is u_i shares document $d' \neq d$ with $\mathcal{C}$. Then $\mathtt{qct}_{d'}$ leaks w_i (and d' as discussed in the previous point) and this leakage is extended to all the query ciphertexts from the same query. We say $(\hat{u}_i, w_i, \perp)$ is leaked. Notice that identity of $d \notin \mathsf{Access}(\mathcal{C})$ is not leaked.

Finally, let us consider $u_i \notin \mathcal{C}$ and $\mathsf{Access}(u_i) \cap \mathsf{Access}(\mathcal{C}) = \emptyset$, in which case we say that $\mathtt{qct}_d$ is a *closed* query ciphertext. This is the case described for the passive case (as in the case all query ciphertexts are closed) and Server can cluster together the closed query ciphertexts that are for the same keyword *and* document and those that are for the same user *and* document. We can thus summarize leakage derived from query ciphertexts as follows.

5. for every query ciphertext $\mathtt{qct}_d$ of i-th query qSet_i for keyword w_i performed by user u_i;
   (a) if $u_i \in \mathcal{C}$, then (u_i, w_i, d) is leaked; the query is called an *open* query;
   (b) if $u_i \notin \mathcal{C}$ and $d \in \mathsf{Access}(\mathcal{C})$, then $(\hat{u}_i, w_i, d)$ is leaked;
   (c) if $u_i \notin \mathcal{C}$, $d \notin \mathsf{Access}(\mathcal{C})$ and $\mathsf{Access}(u_i) \cap \mathsf{Access}(\mathcal{C}) \neq \emptyset$, then $(\hat{u}_i, w_i, \perp)$ is leaked; the query is called an *half-open* query;
6. Equivalence classes $D_1, \ldots, D_r$ over the set of pairs of closed query ciphertexts and ciphertexts.
7. Equivalence classes $E_1, \ldots, E_t$ over the set of pairs of closed query ciphertexts and tokens.

In what follows we will denote by $\mathcal{L}(\mathcal{I}, \mathcal{C})$ the leakage described in Point 0-7 above. In the proof (see Appendix A) we will show that our construction does not leak any information about an instance $\mathcal{I}$ other than $\mathcal{L}(\mathcal{I}, \mathcal{C})$ where Server corrupts users in $\mathcal{C}$. We do so by describing a simulator $\mathcal{S}$ for SSEwSU that takes as input a coalition $\mathcal{C}$ of users along with $\mathcal{L}(\mathcal{I}, \mathcal{C})$ and returns a view that is indistinguishable from the real view of Server.

3.2 Simulator

In this section, we give an overview of the simulator that we use to prove security against passive Server; that is, the case $\mathcal{C} = \emptyset$. Specifically, the output of simulator Sim on in input the leakage $\mathcal{L}(\mathcal{I}, \emptyset)$ of instance $\mathcal{I}$ is indistinguishable from $\mathsf{View}^{\mathcal{U},\emptyset}(\lambda, \mathcal{I})$. The full proof of security with non-empty $\mathcal{C}$ is found in Appendix A.3.

We proceed through a series of hybrid experiments, with the first experiment being the real experiment, GameR, that produces the view and the last being an experiment that can be efficiently performed by using the leakage of the instance.

In the first game, Game_0, we replace the pseudorandom functions F and G with completely random functions R_1 and R_2. Authorization token ids $\mathtt{uid}_{u,d}$ are computed using $\mathsf{F}(\widetilde{K}_u, d)$ exactly once. Therefore, all $\mathtt{uid}_{u,d}$ are chosen randomly. By the security of the pseudorandom function, the output of Game_0 is indistinguishable from GameR.

In the second game, Game_1, the query ciphertexts are computed in the following way. First of all, $X_{w,d}$ is computed for all pairs (w, d) but only those for which $w \in \mathsf{Kw}(d)$ are added to xSet. Similarly, $U_{u,d}$ is computed for all pairs (u, d) but only those for which $d \in \mathsf{Access}(u)$ are added to the uSet. Finally, the query ciphertext for (u, w, d) is computed as $X_{w,d}^{1/U_{u,d}}$. It is easy to see that the output of Game_0 and Game_1 are identical.

Now observe that in the output of Game_1, the value $R_1(K_u, d)$ appears only in the computation of $U_{u,d}$ and thus, choosing each $U_{u,d}$ independently at random will not modify the the output distribution. We call this game Game_2.

Next, in Game_3, we select all $X_{w,d}$'s independently at random. It is not difficult to see that, under the DDH assumption, the output of Game_3 is indistinguishable from the output of Game_2. Finally, observe that to execute Game_3 we only need to know the size of the xSet and uSet (so that the right number of random and independent elements can be chosen), the number of queries and their sizes, and the relation between query ciphertexts, elements of uSet and elements of xSet which is encoded in the equivalence classes $E_1, \ldots, E_t$. In other words, the information needed to efficiently execute Game_3 constitutes exactly the leakage. We thus take Sim to be the efficient algorithm that performs Game_3 on input of the leakage.

3.3 Security of un-sharing

In this section, we show that once document d has been unshared with user u (that is, token $U_{u,d}$ has been removed from $\mathsf{uSet}[\mathtt{uid}_{u,d}]$) u cannot find out from Server if d contains a certain keyword w. This holds in spite of the fact that a malicious u might still have K_d. Clearly, nothing can prevent a malicious Server from not removing the entry from the uSet, thus effectively not unsharing d. Therefore, we offer security guarantees for unsharing when Server is honest, but still curious. Of course, $\mathsf{Manager}$ is assumed to be honest. We allow u to corrupt a subset $\mathcal{C}$ of the users. We assume that no member $v \in \mathcal{C}$ has access to d. Otherwise u may use v's keys to construct queries for d as user v.

The crux of the argument is if, $\widetilde{K}_d$ were chosen at random, then it would be random even given the view of any $\mathcal{C}$ for which $d \notin \mathsf{Access}(\mathcal{C})$. This implies that $\mathsf{F}(\widetilde{K}_d, d)$ is unpredictable and any coalition that manages to produce a query ciphertext $\mathtt{qct}_d$ can be used to break the pseudorandomness of F. $\widetilde{K}_d$ is not random though as it is computed as $\mathsf{F}(K_3, d)$. However, since K_3 is not part of the view of the coalition, by applying a similar reasoning, we obtain that the ability of a coalition to produce a query ciphertext $\mathtt{qct}_d$ contradicts the pseudorandomness of F.

3.4 Search and Editing rights

In $\mathsf{x\text{-}um}$ described above, access to a document is binary: either a user has access to a document or it does not. In some settings, one would like to be able to discriminate between users that have *search* access to the document (so that the document can be returned as a result to a search query for a keyword) and *edit* access (the user can add/remove keywords from the document). Edit capabilities can be easily added to the construction in the previous section by stipulating that for user u to add keyword w to a document d shared with him, it is sufficient to produce the query ciphertext authorization token id that when combined with the token $\mathsf{uSet}[\mathtt{uid}_{u,d}]$ produced the x-ciphertext to be added to the xSet. User with access to the document can remove keywords. Note that both adding and removing a keyword can be performed without the $\mathsf{Manager}$.

In some settings, though, one would like a finer access control to documents and have the ability to give users either *search* rights or *search and edit* rights. It does not make much sense to have edit rights but not search rights. From now on, we will simply talk of edit rights. Let us denote by $\mathsf{Search}(u)$ and $\mathsf{Edit}(u)$ the sets of documents to which user u has search rights and edit rights, respectively. We next describe a modification that allows $\mathsf{Manager}$ to grant and revoke search and edit rights to users in an efficient way.
The data structure. Each document d is associated with four keys $\widetilde{K}_d, K_d^{\mathsf{s}}, K_d^{\mathsf{e}}$, and K_d^{enc} and the Server maintains three sets: the xSet which has an entry for each pair (w, d) such that $w \in \mathsf{Kw}(d)$, the sSet which has an entry for each pair (d, u) such that $d \in \mathsf{Search}(u)$, and the eSet which has an entry for each pair (d, u) such that $d \in \mathsf{Edit}(u)$.

Similarly to the previous scheme, the xSet entry relative to (w, d) consists of an x-ciphertext $g^{\mathsf{F}(K_d^{\mathsf{s}}, w)\cdot\mathsf{F}(\widetilde{K}_d, d)}$ and a y-ciphertext $\mathsf{Enc}(K_d^{\mathsf{enc}}, \mathtt{meta}_d)$. The (d, u) authorization in the $\mathsf{sSet}[\mathsf{F}(\widetilde{K}_u, d)]$ is computed as $\mathsf{F}(\widetilde{K}_d, d)/\mathsf{F}(K_u, d)$. In a similar manner, the (d, u) authorization in the $\mathsf{eSet}[\mathsf{F}(\widetilde{K}_u, K_d^{\mathsf{e}})]$ is computed as $\mathsf{F}(\widetilde{K}_d, d)/\mathsf{F}(K_d^{\mathsf{e}}, K_u)$. Files are shared (and unshared) for searching and editing by the $\mathsf{Manager}$ by constructing the appropriate entries for sSet (search) and for sSet and eSet (edit) and by asking Server to add/remove them to/from the current data structure. Moreover, users are provided with K_d^{enc} and K_d^{s} for documents d shared only for searching and with $K_d^{\mathsf{enc}}, K_d^{\mathsf{s}}$ and K_d^{d} for documents d shared for editing. It is also possible for $u \in \mathsf{AccList}(d)$ to grant access to user v in a similar manner.
Searching for a keyword. The query ciphertext $\mathtt{qct}_d$ for searching for keyword w is computed as ciphertext $g^{\mathsf{F}(K_d^{\mathsf{s}}, w)\cdot\mathsf{F}(K_u, d)}$. Note, that a user needs only K_d^{s}, K_u and $\widetilde{K}_u$ to compute the query ciphertext. Query ciphertext are composed with tokens from the sSet and the resulting ciphertext is searched for in the xSet. If a matching x-ciphertext is found, the corresponding y-ciphertext is returned. Proof of security can be easily derived from the proof of security of the previous construction.
Editing a document. Let us now explain how user u adds a keyword w to a document d that has been shared for editing with u (removing is achieved similarly). Keep in mind that the occurrence of w in d is encoded by having x-ciphertext $g^{\mathsf{F}(K_d^{\mathsf{s}}, w)\cdot\mathsf{F}(\widetilde{K}_d, d)}$ in the xSet. We require user u to produce an *editing* ciphertext that combined with the token in the eSet for u and d gives the desired x-ciphertext to be added to xSet. The editing ciphertext needed to add w to document d is computed as $g^{\mathsf{F}(K_d^{\mathsf{s}}, w)\cdot\mathsf{F}(K_d^{\mathsf{e}}, K_u)}$. It is easy to see that when authorization $\mathsf{F}(\widetilde{K}_d, d)/\mathsf{F}(K_d^{\mathsf{e}}, K_u)$ is applied to the editing ciphertext one obtains x-ciphertext $g^{\mathsf{F}(K_d^{\mathsf{s}}, w)\cdot\mathsf{F}(\widetilde{K}_d, d)}$. The y-ciphertext can be provided instead directly by u.

3.5 Further reducing the leakage

We present $\mathsf{x\text{-}ump}$ (for *pairing x-um*) that reduces the leakage at the expense of an increased computational load for Server. In $\mathsf{x\text{-}ump}$, the ciphertext queries are randomized. This has the consequence that Server only learns whether two ciphertext queries from different users are for the same keyword w and the same document d when $w \in \mathsf{Kw}(d)$ (that is, successful queries). We only provide a high-level construction and proof since $\mathsf{x\text{-}um}$ and $\mathsf{x\text{-}ump}$ are quite similar. First, we define the bilinear setting of the construction and the computational assumption for security.

An *asymmetric bilinear setting* is defined by an efficient algorithm $\mathcal{GG}$ that takes as input the security parameter λ and outputs the description of three multiplicative cyclic groups $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_T$ of prime order p with $|p| = \Theta(\lambda)$ and the description of an efficiently computable non-trivial bilinear mapping $\mathsf{e} : \mathcal{G}_1 \times \mathcal{G}_2 \rightarrow \mathcal{G}_T$. That is, for $g_1 \in \mathcal{G}_1$ with $g_1 \neq 1_{\mathcal{G}_1}$, and $g_2 \in \mathcal{G}_2$ with $g_2 \neq 1_{\mathcal{G}_2}$, $\mathsf{e}(g, g) \neq 1_{\mathcal{G}_T}$ and for all $a, b \in \mathbb{Z}_p$, we have $\mathsf{e}(g_1^a, g_2^b) = \mathsf{e}(g_1, g_2)^{ab}$.

We describe how the element of the xSet and of the uSet are constructed and how users perform queries. Let g_1 and g_2 be two generators of $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively, and set $g_T = \mathsf{e}(g_1, g_2)$. The x-ciphertext $X_{w,d}$ is computed as $X_{w,d} = g_2^{\mathsf{F}(\widetilde{K}_d, d) \cdot \mathsf{F}(K_d, w))}$ and authorization token $\mathsf{uSet}[\mathtt{uid}_{u,d}]$ is computed just as in $\mathsf{x\text{-}um}$. That is, $\mathsf{uSet}[\mathsf{F}(\widetilde{K}_u, d)] = \mathsf{F}(\widetilde{K}_d, d) / \mathsf{F}(K_u, d)$. Just as in $\mathsf{x\text{-}um}$, xSet includes all $X_{w,d}$ with $w \in \mathsf{Kw}(d)$ and uSet includes all $U_{u,d}$ with $d \in \mathsf{Access}(u)$. User u computes the query ciphertext for keyword w and document d, by randomly picking $R \in \mathbb{Z}_p$ and returning $\left(\mathsf{F}(\widetilde{F}_u, d), g_1^R, g_T^{R \cdot \mathsf{F}(K_d, w) \cdot \mathsf{F}(K_u, d)}\right)$. Query ciphertext $(\mathtt{uid}_{u,d}, \mathtt{qct}_1, \mathtt{qct}_2)$ is successful if there exist x-ciphertext $X \in \mathsf{xSet}$ such that $\mathsf{e}(\mathtt{qct}_1, X) = \mathtt{qct}_2^{\mathsf{uSet}[\mathtt{uid}_{u,d}]}$. Similar modifications from Section 3.4 may be applied for search and edit access separation.

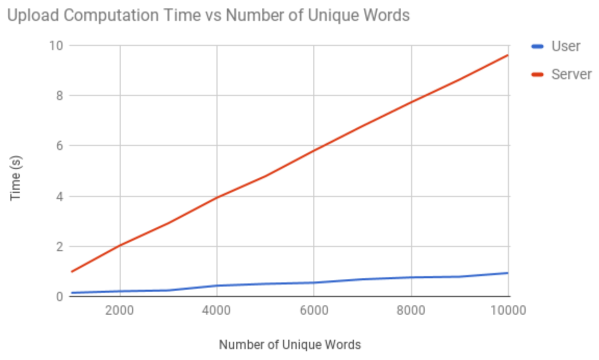
Let us discuss the impact of randomized query ciphertexts. In $\mathsf{x\text{-}um}$, the server checks if the value obtained by raising the query ciphertext to $\mathsf{uSet}[\mathtt{uid}_{u,d}]$ is found in the xSet, requiring one exponentiation. In contrast, in $\mathsf{x\text{-}ump}$, the verification process must be repeated for every x-ciphertext. We stress that this extra computation is performed by Server and the time needed by a user to compute a query ciphertext remains constant and independent from the sizes of xSet and uSet. On the other hand, x-user leakage is greatly reduced. Specifically, if two successful queries are for the same keyword w and document d, they will be both hit x-ciphertext $X_{w,d}$. However, two unsuccessful queries for the same (w, d) will not be recognized as such by the Server thanks to the random factor R. The proof of security is very similar to the one of $\mathsf{x\text{-}um}$ and it only requires DDH to be hard in group $\mathcal{G}_2$.

4 Experiments

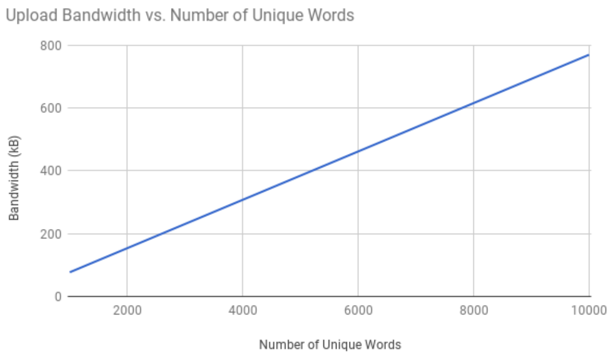
In this section, we investigate the costs of $\mathsf{x\text{-}um}$ and experimentally evaluate the growth of leakage as queries are being performed. All experiments are conducted on two identical machines, one for the Server and one for the user. The machines used are Ubuntu PC with Intel Xeon CPU (12 cores, 3.50 GHz). Each machine has 32 GB RAM with 1 TB hard disk.

Our experiments will only measure costs associated with $\mathsf{x\text{-}um}$. In practice, $\mathsf{x\text{-}um}$ is accompanied by some storage system that allows retrieval of encrypted data. We ignore costs that would be incurred by such a storage system.

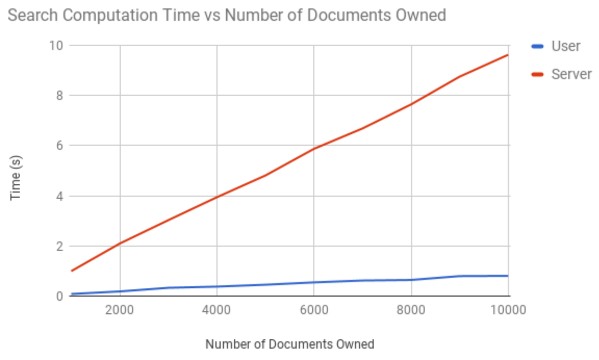
All associated programs are implemented using C++ and do not take advantage of the multiple cores available. We use SHA-256-based G and F and AES under Galois Counter Mode for $(\mathsf{Enc}, \mathsf{Dec})$. These cryptographic functions implementations are from the BoringSSL library (a fork of OpenSSL 1.0.2). The length of the keys used are 128 bits. All identifiers (document and user) are also 128 bits. We use the NIST recommended Curve P-224 (which has the identifier NID_secp224r1 in OpenSSL) as $\mathbb{G}$. All group exponents are serialized in big-endian form. Elliptic curve points are serialized to octet strings in compressed form using the methods defined by ANSI X9.62 ECDSA.



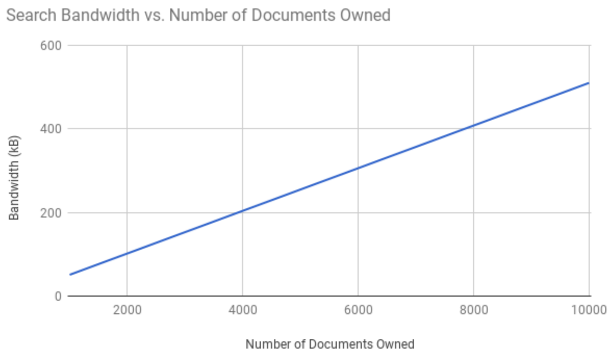
(a) Upload Computation Time.



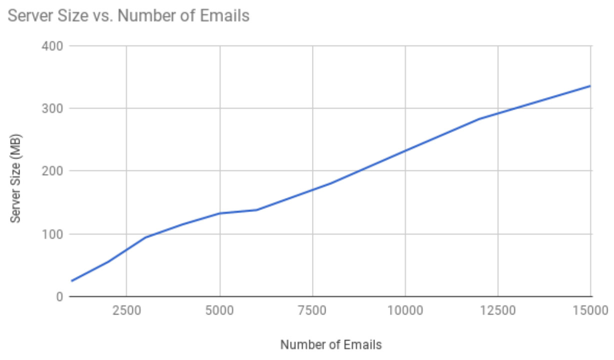
(b) Upload Bandwidth.



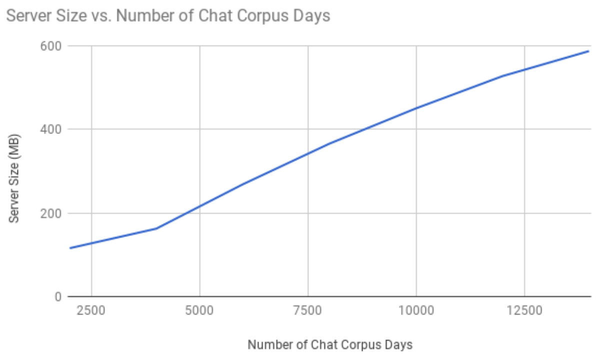
(c) Search Computation Time.



(d) Search Bandwidth.



(e) Enron Email Server Storage.



(f) Ubuntu Chat Server Storage.

Figure 1: Performance Evaluation.

4.1 Performance

We measure the computation time and bandwidth of uploading and searching documents of x-um. As expected, the upload and search metrics grow linearly in the number of unique terms and number of owned documents respectively. Furthermore, we note that the amount a user's computational time is much smaller than the server. This is very important as single machine users are more limited in computational power compared to large cloud service providers. The results can be seen in Figure 1.

Enron Email Dataset. We consider storing the Enron email dataset [6] generated by 150 employees using x-um. Any user that is the sender, recipient, cc'd or bcc'd of an email will be given search access to that email. The sender will be granted edit access. Every recipient of the email will be given edit access with $\frac{1}{2}$ probability. The server storage required is 5-6 times the size of the emails being uploaded.

We remark that SSE might be insecure for emails. For example, the methods described in [15] show that injections attacks leaks the contents of all emails stored. We use the dataset as a means to test practicality.

Ubuntu Chat Corpus. In a separate experiment, we store the Ubuntu Chat Corpus (UCC) [14] with over 700000 users using our scheme. Like emails, the chat logs provide an excellent framework for multi-user searchable schemes. We split the chat corpus into days. That is, each day of history becomes a single file. All users who appear in the chat log for a day will have read rights. Each of the appearing users will also receive write rights with probability $\frac{1}{2}$. For this dataset, we also stem the input for each language. Stemming removes common words as well as providing pseudonyms.

4.2 Leakage Growth

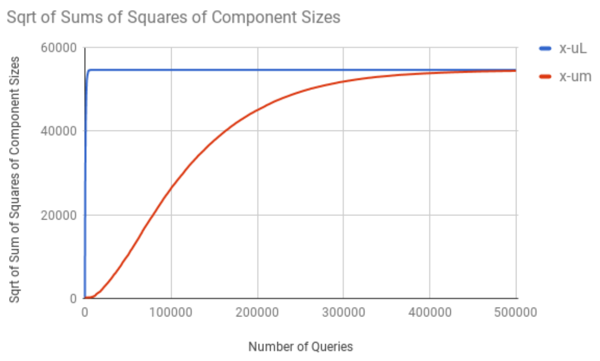
Through experiments, we attempt to compare the rate of leakage growth of x-uL and x-um as queries are performed. Recall the graph G of users, described in Section 3.1. An edge between users u_1 and u_2 means that queries by u_1 or u_2 leak information about documents in $\mathsf{Access}(u_1) \cap \mathsf{Access}(u_2)$. For x-um, an edge (u_1, u_2) exists iff both users queried for the same keyword w and share at least one document in common. On the other hand, an edge exists in x-uL if both users share at least one document in common and either user ever queried. Furthermore, x-user leakage is transitive. If two users are in the same component, their queries can leak information about their intersection.

As G becomes more connected, more x-user leakage exists. If G has no edges (and consists of $|\mathcal{U}|$ connected components, one for each user), no x-user leakage exists. Conversely, the complete graph has x-user leakage for every pair of users. One can consider the vector of connected component sizes of G and how it varies as queries are performed. The initial vector consists of $|\mathcal{U}|$ 1's (each vertex is in a connected component by itself). We measure leakage by the length of the vector of connected component sizes of the current graph G. We pad with 0's to keep the vector of dimension $|\mathcal{U}|$. We measure length by the L_2 norm (the square root of the sum of the squares of component sizes) and L_∞ norm (the largest component size). We also plot the total number of components.

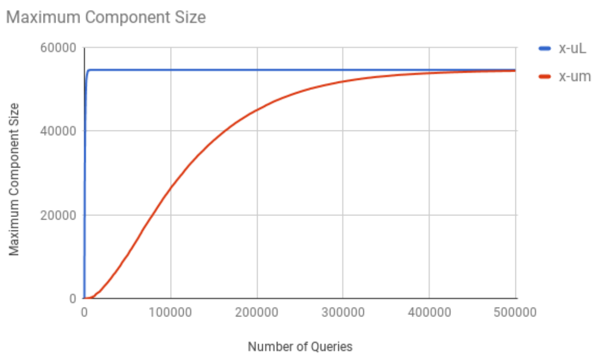
Using 2500 days of UCC data with approximately 55000 users, we compute these metrics for x-uL and x-um. Keywords are drawn from the global distribution of terms in UCC after stemming. The querying user is drawn uniformly at random from all users. We see that x-um leakage grows significantly slower than x-uL in all three metrics. In particular, for all three metrics, x-uL approaches a single connected component with 100 queries. For x-um, it is possible to perform hundreds of thousands queries before this threshold is reached. In fact, it takes at least 80000 queries to reach 1/3 of the metrics of a single component. The results may be seen in Figure 2.

References

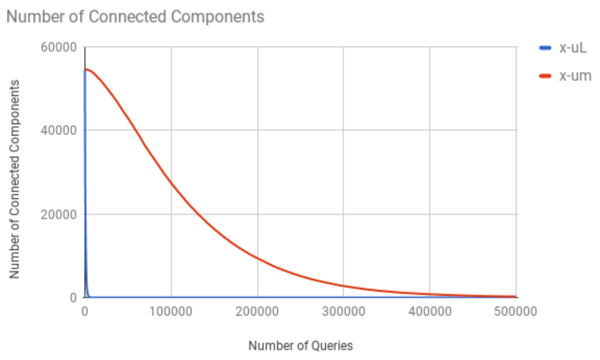
[1] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In C. Cachin and J. L. Camenisch, editors, *EUROCRYPT 2004*, pages 506–522. Springer, 2004.



(a) Sqrt of Sum of Squares of Component Sizes.



(b) Maximum Component Size.



(c) Number of Connected Components.

Figure 2: Leakage Evaluation.

[2] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013*, pages 353–373, 2013. Also Cryptology ePrint Archive, Report 2013/169.

[3] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In M. Abe, editor, *ASIACRYPT 2010*, pages 577–594, 2010. Also Cryptology ePrint Archive, Report 2011/010.

[4] 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*, pages 79–88, 2006. Also Cryptology ePrint Archive, Report 2006/210.

[5] C. Dong, G. Russello, and N. Dulay. *Shared and Searchable Encrypted Data for Untrusted Servers*, pages 127–143. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[6] EDRM (edrm.net). Enron data set. `http://www.edrm.net/resources/data-sets/edrm-enron-email-data-set`.

[7] S. Kamara and K. Lauter. Cryptographic cloud storage. In *Financial Cryptography and Data Security: FC 2010 Workshops, RLCPS, WECSR, and WLC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers*. Springer Berlin Heidelberg, 2010.

[8] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In J. Coron and J. B. Nielsen, editors, *EUROCRYPT 2017*, volume 10212, pages 94–124, 2017. Also Cryptology ePrint Archive, Report 2017/126.

[9] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 359–374. IEEE Computer Society, 2014.

[10] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, 2011.

[11] R. A. Popa and N. Zeldovich. Multi-key searchable encryption. Cryptology ePrint Archive, Report 2013/508, 2013.

[12] J. Shi, J. Lai, Y. Li, R. H. Deng, and J. Weng. *Authorized Keyword Search on Encrypted Data*, pages 419–435. Springer International Publishing, Cham, 2014.

[13] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[14] D. Uthus. Ubuntu chat corpus. `http://daviduthus.org/UCC/`.

[15] Y. Zhang, J. Katz, and C. Papamanthou. 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`.

A Re-writable Deterministic Hashing

In this section we introduce the concept of a Re-writable Deterministic Hashing scheme that can be seen as an enhanced one-way function with re-writing capabilities. Roughly speaking, a *Re-writable Deterministic Hashing* (RDH, in short) scheme operates on three types of objects, *plaintexts*, *ciphertexts* and *tokens*, and consists of three deterministic algorithms:

1. H produces a ciphertext from a pair of plaintexts.

2. GenToken produces a token from an ordered pair of plaintexts.

3. Apply applies a token to a ciphertext to get a ciphertext.

We assume that H is symmetric with respect to the plaintexts; that is, $\mathsf{H}(A, B) = \mathsf{H}(B, A)$ for all plaintexts A and B. The three algorithms are linked by the re-writing property.
Re-writing. For all plaintexts A, B, C we have that

$$\mathsf{Apply}(\mathsf{H}(A, B), \mathsf{GenToken}(B, C)) = \mathsf{H}(A, C).$$

A token tok can be *applied* to ciphertext ct if the token's first plaintext is one of the two plaintexts of the ciphertext. In this case, we say that ct and tok are compatible.
Security notion for a RDH. Security for a RDH is based on the real game, GameR, that receives as input the security parameter, λ, and an instance $\mathcal{I} = (N, (a_i, b_i)_{i\in[n]},$
$(c_j, d_j)_{j\in[m]}, \mathsf{Open})$ consisting of the number N of plaintexts, the indices of the two plaintexts in each of the n ciphertexts, each of the m tokens and the subset of indices $\mathsf{Open} \subseteq [N]$ of revealed plaintext (which we refer to as *open* ciphertexts). GameR samples and outputs a *transcript* $\mathcal{T}$. In GameR, each plaintext is randomly selected and the selected plaintexts are used to compute the specified ciphertexts and tokens. The open plaintexts and all ciphertexts and tokens are given in output, each with the indication of which open plaintext, if any, is involved in each of them.

```
GameR($\lambda, \mathcal{I}$)
 0. For $i = 1, \ldots, N$, randomly select $\mathtt{M}_i \leftarrow \{0,1\}^\lambda$;
 1. For $i = 1, \ldots, n$, output $\mathtt{ct}_i = \mathsf{H}(\mathtt{M}_{a_i}, \mathtt{M}_{b_i})$
        and $\mathtt{M}_x$ for $x \in \{a_i, b_i\} \cap \mathsf{Open}$;
 2. For $j = 1, \ldots, m$, output $\mathtt{tok}_j = \mathsf{GenToken}(\mathtt{M}_{c_j}, \mathtt{M}_{d_j})$
        and $\mathtt{M}_x$ for $x \in \{c_j, d_j\} \cap \mathsf{Open}$;
 3. Output $(i, \mathtt{M}_i)$ for $i \in \mathsf{Open}$;
```

GameR

Due to its deterministic nature, we consider security for RDH when plaintexts have large enough entropy. GameR, described above, corresponds to the scenario where certain plaintexts are chosen at random from the binary strings of a given length (the length λ of the plaintexts is the security parameter of experiment). We make the pessimistic assumption that if a ciphertext or a token contains an open plaintext then this information is leaked to an observer. This pessimistic assumption provides stronger security notions. We are now ready to present our security definition for RDH.

Definition 2. RDH *scheme is secure for class* $\mathbb{C}$ *of instances with respect to leakage function* Leak *if there exists an efficient simulator* S *such that, for every* $\mathcal{I} \in \mathbb{C}$

$$\{\mathsf{GameR}(\lambda, \mathcal{I})\}_\lambda \approx_c \{S(1^\lambda, \mathsf{Leak}(\mathcal{I}))\}_\lambda.$$

A.1 Regular instances

We introduce the notion of a regular instance of a RDH and then we identify the leakage $\mathsf{mL}(\mathcal{I})$ that is necessarily leaked by a transcript $\mathcal{T}$ of a regular instance $\mathcal{I}$ about the instance itself. We will then show, in Appendix A.2, that there exists a construction of RDH that is secure for regular instances with respect to leakage function mL, under the Decisional Diffie Hellman assumption. Thus, the construction can be considered optimal with respect to leakage. Finally, in Appendix A.3, we show that it is sufficient to consider regular instances of RDH for our application to SSEwSU with efficient sharing and un-sharing.

We start by introduction the following terminology. We classify the ciphertexts and the tokens of an instance $\mathcal{I} = (N, (a_i, b_i)_{i\in[n]}, (c_j, d_j)_{j\in[m]}, \mathsf{Open})$ as *open, half-open, closed* depending on whether both, only one or neither of the plaintexts belong to Open.

Definition 3. *An instance $\mathcal{I} = (N, (a_i, b_i)_{i\in[n]}, (c_j, d_j)_{j\in[m]}, \mathsf{Open})$ is* regular *if the following conditions hold:*

1. *no two tokens have the same first plaintext; that is, $i \neq j$ implies $c_i \neq c_j$;*
2. *if ciphertext with plaintexts (a, b) is compatible with a token through plaintext b then plaintext a does not appear in any token; that is, for all $i \in [n]$, if $b_i = c_j$ for some $j \in [m]$, then $a_i \neq c_k$ and $a_i \neq d_k$ for all $k \in [m]$;*
3. *all tokens are either open or closed; that is, for all $j \in [m]$, we have that $|\{c_j, d_j\} \cap \mathsf{Open}| \neq 1$.*

Leakage from closed ciphertexts. By definition, open tokens are not compatible with closed ciphertexts. Therefore, the only leakage that can derive from closed ciphertexts are from their interaction with closed tokens.

Definition 4. *We say that closed ciphertexts s and d of an instance $\mathcal{I}$ constitute* match $\mathtt{M} = (s, t, d)$ *with token t if*

$$\mathsf{Apply}(\mathtt{ct}_s, \mathtt{tok}_t) = \mathtt{ct}_d.$$

We call s the *source* ciphertext and d the *destination* ciphertext of match (s, t, d). In a match, by applying the token to the source ciphertext, we get to the destination. In a *closed collision*, we have a similar situation except the destination ciphertext is not part of the transcript.

Definition 5. *We say that closed ciphertexts s_1 and s_2 of an instance $\mathcal{I}$ constitute a* closed collision $\mathtt{C} = (s_1, s_2, t_1, t_2)$ *with tokens t_1 and t_2 of the same instance if for all transcripts of $\mathcal{I}$*

$$\mathsf{Apply}(\mathtt{ct}_{s_1}, \mathtt{tok}_{t_1}) = \mathsf{Apply}(\mathtt{ct}_{s_2}, \mathtt{tok}_{t_2})$$

and $\mathsf{Apply}(\mathtt{ct}_{s_2}, \mathtt{tok}_{t_2}) \neq \mathtt{ct}_d$*, for all $1 \leq d \leq n$.*

Similarly to matches, we call $\mathtt{ct}_{s_1}$ and $\mathtt{ct}_{s_2}$ the *source* ciphertexts of $\mathtt{C}$ and $\mathsf{H}(\mathtt{M}_{a_{s_1}}, \mathtt{M}_{d_{t_1}}) = \mathsf{H}(\mathtt{M}_{a_{s_2}}, \mathtt{M}_{d_{t_2}})$ its *destination* ciphertext. In the above definitions of matches and closed collisions, the equalities are intended as equalities between the formal random variables of the ciphertexts and tokens. The real experiment instantiates the random variables in sampling a transcript.

The graph $G_{\mathcal{I}}$ of an instance. Matches and closed collisions of an instance are revealed by every transcript of the instance and they give information about the instance itself. We encode the information in the *graph $G_{\mathcal{I}}$ of instance $\mathcal{I}$*. $G_{\mathcal{I}}$ contains one vertex for each of the n closed ciphertexts of the instance and for each ciphertext which is the destination ciphertext of a collision. Graph $G_{\mathcal{I}}$ contains directed edge from $\mathtt{ct}$ to $\mathtt{ct}'$ labeled with token $\mathtt{tok}$ (necessarily a closed token) if and only if $\mathsf{Apply}(\mathtt{ct}, \mathtt{tok}) = \mathtt{ct}'$. We denote the edge by writing $\mathtt{ct} \overset{\mathtt{tok}}{\to} \mathtt{ct}'$.

The following properties of the graph $G_{\mathcal{I}}$ of a regular instance will be useful in the proof of security. Essentially they show that, even though $G_{\mathcal{I}}$ has cycles, ciphertexts and tokens appear in cycles in a very well-behaved way. Specifically, ciphertexts appear in at most one cycle and tokens (in the form of edge labels) can appear in more than one cycle but, essentially, all cycles are the same cycle. Also, two edges of $G_{\mathcal{I}}$ are *adjacent* if they share one vertex.

Lemma 1. *Every vertex of $G_{\mathcal{I}}$ belongs to at most one cycle.*

Proof. Direct consequence of the fact that each vertex has outgoing degree at most one. □

Lemma 2. *Consider two pairs of adjacent edges e_0 and e_1, and e_0' and e_1'. If e_0 and e_0' are labeled with the same token $\mathtt{tok}_0$ then e_1 and e_1' are also labeled by the same token $\mathtt{tok}_1 \neq \mathtt{tok}_0$.*

Proof. Suppose $\mathtt{tok}_0$ has plaintexts (b_0, b_1). Thus there exists a and a' such that e_0 and e_0' are the edges

$$(a, b_0) \overset{\mathtt{tok}_0}{\to} (a, b_1) \text{ and } (a', b_0) \overset{\mathtt{tok}_0}{\to} (a', b_1).$$

Let $\mathtt{tok}_1$ be the token labeling the edge e_1 and $\mathtt{tok}_1'$ be the token labeling edge e_1'. Then both tokens must have the same first plaintext b_1 and thus, by regularity, it must be that $\mathtt{tok}_1 = \mathtt{tok}_1'$ and, obviously, $\mathtt{tok}_1, \mathtt{tok}_1' \neq \mathtt{tok}$. □

Corollary 1. *Let C_1 and C_2 be two cycles of $G_{\mathcal{I}}$ that contain an edge labeled by token* $\texttt{tok}$*. Then C_1 and C_2 contain the same number of edges and these edges are labeled by same set of tokens appearing in the same order.*

Corollary 2. *No path of $G_{\mathcal{I}}$ has two edges labeled by the same token.*

Leakage from half-open ciphertexts. We start with the following definition.

Definition 6. *We say that ciphertexts i_1 and i_2 of an instance $\mathcal{I}$ constitute a* half-open collision $\texttt{C} = (i_1, i_2)$ *for the same instance if*

$$a_{i_1} = a_{i_2} \quad \textit{and} \quad b_{i_1}, b_{i_2} \in R.$$

We observe that half-open collisions are revealed by a transcript. Indeed, if (i_1, i_2) is a half-open collision then plaintexts $\texttt{M}_{b_{i_1}}$ and $\texttt{M}_{b_{i_2}}$ are available and thus it is possible to construct $\texttt{tok} = \mathsf{GenToken}(\texttt{M}_{b_{i_1}}, B)$ and $\texttt{tok}' = \mathsf{GenToken}(B_{b_{i_2}}, B)$, for any arbitrary plaintext B. Now, the application of $\texttt{tok}$ to $\texttt{ct}_{i_1}$ gives $\mathsf{H}(\texttt{M}_{a_{i_1}}, B)$ which is equal to $\mathsf{H}(\texttt{M}_{a_{i_2}}, B)$ that can be obtained by applying $\texttt{tok}'$ to $\texttt{ct}_{i_2}$.

The half-open collisions allow to cluster the half-open ciphertexts according to the first ciphertext. Specifically, it is possible to organize the information derived from half-open collisions in a $n \times N$ matrix L whose entries are either $\perp$ or the index of an half-open ciphertext and such that

- ciphertexts in the same column have the same first plaintext (which is not in Open);
- all the ciphertexts in row r have $\texttt{M}_r$ as second plaintext (and $r \in \mathsf{Open}$);

The necessary leakage function. By summing up the discussion above, the leakage $\mathsf{mL}(\mathcal{I})$ associated with instance $\mathcal{I} = (N, (a_i, b_i)_{i\in[n]}, (c_j, d_j)_{j\in[m]}, \mathsf{Open})$ is defined as

1. the number n of ciphertexts and the number m of tokens;
2. $\mathsf{Open} \cap \{a_i, b_i\}$, for $i = 1, \ldots, n$;
3. $\mathsf{Open} \cap \{c_j, d_j\}$, for $j = 1, \ldots, m$;
4. matrix L encoding the half-open collisions;
5. the instance graph $G_{\mathcal{I}}$.

A.2 A regular RDH with minimum leakage

In our construction the plaintexts and tokens are from $\mathbb{Z}_p$, for p prime of length $\Theta(\lambda)$, and ciphertexts are element of a cyclic group $\mathbb{G}$ of order p in which the Decisional Diffie-Hellman problem is hard. Our construction is inspired by a similar construction used in [2].

We will use the following equivalent assumption in which $\mathbf{x} \leftarrow \mathbb{Z}_p^{l_0}$, $\mathbf{y} \leftarrow \mathbb{Z}_p^{l_1}$, $\mathbf{r} \leftarrow \mathbb{Z}_p^{l_0 \cdot l_1}$, and $\mathbf{x} \times \mathbf{y}$ is an $l_0 \times l_1$ matrix whose (i,j)-entry is $x_i \cdot y_j$. Moreover, for a matrix $A = (a_{i,j})$ we set $g^A = (g^{a_{ij}})$.

Lemma 3. *If DDH holds for $\mathcal{GG}$ then for any l_0, l_1 that are bounded by a polynomial in λ we have that distributions $D^0_{l_0,l_1,\lambda}$ and $D^1_{l_0,l_1,\lambda}$ are computational indistinguishable, where*

$$D^{\xi}_{l_0,l_1,\lambda} = \Big\{(g, \mathcal{G}) \leftarrow \mathcal{GG}(1^\lambda); \mathbf{x}, \leftarrow \mathbb{Z}^{l_0}_{|\mathcal{G}|}, \mathbf{y}, \leftarrow \mathbb{Z}^{l_1}_{|\mathcal{G}|}, \mathbf{r} \leftarrow \mathbb{Z}^{l_0 \cdot l_1}_{|\mathcal{G}|} : (g^{\mathbf{x}}, g^{\mathbf{x}}, g^{\mathbf{x}\times\mathbf{y}+\xi\cdot\mathbf{r}})\Big\}.$$

For every group generator $\mathcal{GG}$, we consider $\mathsf{RDH}_{\mathcal{GG}} = (\mathsf{H}, \mathsf{GenToken}, \mathsf{Apply})$ defined as follows:

- $\mathsf{H}(A, B) = g^{A \cdot B}$;
- $\mathsf{GenToken}(C, D) = D/C$;

- $\mathsf{Apply}(\mathtt{ct}, \mathtt{tok}) = \mathtt{ct}^{\mathtt{tok}}$;

We note that all three algorithms share as an implicit input a randomly selected output $(g, \mathcal{G})$ of the group generator $\mathcal{G}G$. We prove that if the DDH assumption holds for $\mathcal{G}G$, then the construction is secure with respect to regular instances by showing a simulator that outputs transcripts with same distribution as GameR on input $\mathsf{mL}(\mathcal{I})$.

```
Game_0(λ, I)
 0. Sampling the plaintexts
      for $i \in [N]$, randomly select $\mathtt{M}_i \leftarrow \mathbb{Z}_p$;
 1. Computing the open ciphertexts
      for each $i \in \mathtt{oC}$, set $\mathtt{ct}_i = g^{\mathtt{M}_{a_i} \cdot \mathtt{M}_{b_i}}$;
 2. Computing the open tokens
      for each $j \in \mathtt{oT}$, set $\mathtt{tok}_j = \mathtt{M}_{d_j} \cdot \mathtt{M}_{c_j}^{-1}$;
 3. Computing the half-closed ciphertexts
      for each column $c$ of matrix $L$:
        set $A_c = g^{\mathtt{M}_c}$;
        for each $r$ such that $L[r, c] = i \neq \perp$;
          set $\mathtt{ct}_i = A_c^{\mathtt{M}_{b_r}}$;
 4. Computing the closed tokens
      for each $j \in \mathtt{cT}$, set $\mathtt{tok}_j = \mathtt{M}_{d_j} \cdot \mathtt{M}_{c_j}^{-1}$;
 5. Computing the closed ciphertexts
      Computing sink ciphertexts
        for each sink ciphertext $i$, set $\mathtt{ct}_i = g^{\mathtt{M}_{a_i} \cdot \mathtt{M}_{b_i}}$;
      Breaking the cycles in $G_{\mathcal{I}}$
        for each cycle $i_0, \ldots, i_{\ell-1}$
          set $\mathtt{ct}_{i_0} = g^{\mathtt{M}_{a_{i_0}} \cdot \mathtt{M}_{b_{i_0}}}$;
      Completing $G_{\mathcal{I}}$
        repeat until all ciphertexts have been computed
          if $\mathtt{ct}_{k_1} \overset{\mathtt{tok}_j}{\to} \mathtt{ct}_{k_2}$ and $\mathtt{ct}_{k_2}$ has been computed
            set $\mathtt{ct}_{k_1} = \mathtt{ct}_{k_2}^{1/\mathtt{tok}_j}$;
```

To see that Game_0 above is equivalent to GameR, we make the following observations. Clearly, open ciphertexts and all tokens are computed in the same way in the two games. If $\mathtt{ct}_i$ is half-open and $L[r, c] = i$ then by definition of the matrix L we have that M_r and M_c are the two plaintexts of $\mathtt{ct}_i$. Let us now focus on the closed ciphertexts. By Lemma 1, every vertex of the instance graph $G_{\mathcal{I}}$ belongs to at most one cycle and thus the procedure sets the value of the ciphertexts consistently. Now we prove that the output of Game_1 above has the same distribution as the output of Game_0. Observe that the tokens that label the edges of a cycle of $\mathbb{G}_{\mathcal{I}}$ have the cumulative effect of re-writing a ciphertext into itself. This implies that the product of the l tokens that appear in a cycle of $\mathbb{G}_{\mathcal{I}}$ is equal to 1. Since by Corollary 2 no two edges of a cycle are labeled by the same token, we have that any set of $l-1$ tokens appearing over the edges of a cycle of length l depend on $l-1$ plaintexts (their first plaintexts) that only appear as plaintext in these tokens. We can thus conclude that any $l-1$ tokens appearing in a cycle of length l are randomly and independently distributed over $\mathbb{Z}_p$. The remaining token $\mathtt{tok}$ is the reciprocal of the product of the other $l-1$. Moreover, by Corollary 1, the value of the remaining token $\mathtt{tok}$ does not depend on the cycle chosen since all cycles contain the same tokens. A similar and simpler observation holds for paths. Therefore, we can randomly select all tokens independently at random from $\mathbb{Z}_p$ under the only constraint that the product of tokens appearing in the same cycle is 1.

In Game_2, we select all sink ciphertexts and all ciphertexts used to break cycles (the *cycle-breakers*) at random in $\mathbb{G}$. We next show that, under the DDH assumption, Game_1 and Game_2 are indistinguishable and we do so by means of a series of intermediate games in each one of which one more ciphertext is made random. More precisely, for any subset S of sink and cycle-breakers, we define Game_S to be Game_1 in which ciphertexts in S are chosen uniformly at random from $\mathbb{G}$. Clearly $\mathsf{Game}_\emptyset = \mathsf{Game}_1$ and, if $S^\star$ is the set of all the sinks and cycle-breakers, we $\mathsf{Game}_{S^\star} = \mathsf{Game}_2$. We next prove that, for any $\mathtt{ct} = g^{\mathtt{M}_a \cdot \mathtt{M}_b}$, games Game_S and $\mathsf{Game}_{S\cup\{\mathtt{ct}\}}$ are indistinguishable under the DDH assumption. Consider the algorithm that, on input a DH challenge $(X = g^x, Y = g^y, Z = g^{x\cdot y+\xi\cdot r})$ with $x, y, r \leftarrow \mathbb{Z}_p$ and $\xi \in \{0, 1\}$, executes Game_S with the following exceptions:

1. $\mathtt{ct}$ is computed by setting it equal to Z;
2. for column $c = a$, A_c is set equal to X in Step 3;
3. for column $c = b$, A_c is set equal to Y in Step 3;
4. if any of $\mathtt{M}_a$ or $\mathtt{M}_b$ (but not both) is used to compute a sink or a cycle-breaker ciphertext not in the set S, then X or Y, respectively, is used;

The above modifications have the effect of setting $\mathtt{M}_a = x$ and $\mathtt{M}_b = y$. Since the sink and cycle-breakers are closed ciphertexts, neither of $\mathtt{M}_a$ and $\mathtt{M}_b$ are needed in Step 3 as an exponent of A_c. It is easy to verify, if $\xi = 0$ then we obtain Game_S and if $\xi = 1$, then we obtain $\mathsf{Game}_{S\cup\{\mathtt{ct}\}}$. This gives indistinguishability of the two games.

Observe that in Game_2, A_c is used only in Step 3 and thus we obtain the same distribution by choosing it at random, which will be our final game, Game_3, which is shown below with all changes being bolded.

```
Game_3(λ, I)
 0. Sampling the plaintexts
      for $i \in [N]$, randomly select $\mathtt{M}_i \leftarrow \mathbb{Z}_p$;
 1. Computing the open ciphertexts
      for each $i \in \mathtt{oC}$, set $\mathtt{ct}_i = g^{\mathtt{M}_{a_i}\cdot\mathtt{M}_{b_i}}$;
 2. Computing the open tokens
      for each $j \in \mathtt{oT}$, set $\mathtt{tok}_j = \mathtt{M}_{d_j} \cdot \mathtt{M}_{c_j}^{-1}$;
 3. Computing the half-closed ciphertexts
      for each column $c$ of matrix $L$:
        **randomly select $A_c \leftarrow \mathbb{G}$;**
        for each $r$ such that $L[r, c] = i \neq \perp$;
          set $\mathtt{ct}_i = A_c^{\mathtt{M}_{b_r}}$;
 4. Computing the closed tokens
      **randomly select all closed tokens from $\mathbb{Z}_p$ under
        the constraint that the product of a cycle is 1;**
 5. Computing the closed ciphertexts
      Computing sink ciphertexts
        for each sink ciphertext $i$, **randomly select** $\mathtt{ct}_i \leftarrow \mathbb{G}$;
      Breaking the cycles in $G_\mathcal{I}$
        for each cycle $i_0, \ldots, i_{\ell-1}$
          **randomly select** $\mathtt{ct}_{i_0} \leftarrow \mathbb{G}$;
      Completing $G_\mathcal{I}$
        repeat until all ciphertexts have been computed
          if $\mathtt{ct}_{k_1} \overset{\mathtt{tok}_j}{\rightarrow} \mathtt{ct}_{k_2}$ and $\mathtt{ct}_{k_2}$ has been computed then
            set $\mathtt{ct}_{k_1} = \mathtt{ct}_{k_2}^{1/\mathtt{tok}_j}$;
```

Finally observe that Game_3 can be executed with the following inputs:

1. list of open plaintexts;
2. list of open ciphertexts (each with the indices of the corresponding open plaintexts);

3. list of open tokens (each with the indices of the corresponding open plaintexts);
4. the matrix L;
5. the instance graph;

which is exactly the leakage information obtained by the simulator.
The construction is regular. The construction above uses an RDH scheme. In this section, we show that all instances of RDH from the above construction are regular, except with negligible probability.

We start by observing that all tokens that are generated by the system are stored in the uSet and are generated when a user u is granted access to a document. If two tokens have the same first plaintext, it must be the case that there exist $(u_1, d_1) \neq (u_2, d_2)$ such that $\mathsf{F}(K_{u_1}, d_1) = \mathsf{F}(K_{u_2}, d_2)$. By the pseudo-randomness of the function F this happens with probability negligible in λ.

For the second condition, we observe that the only ciphertexts that are compatible with the tokens are the ciphertexts generated by the user during the search procedure. These ciphertexts have plaintexts $\mathsf{F}(K_d, w)$ and $\mathsf{F}(K_u, d)$. Thus, for the second condition to be violated, there must be d for which $\mathsf{F}(K_d, w)$ appears as a plaintext in a token. So, there exists (u, d') such that either $\mathsf{F}(K_u, d') = \mathsf{F}(K_d, w)$ or $\mathsf{F}(K_1, d') = \mathsf{F}(K_d, w)$. By the pseudo-randomness of F both events have negligible probability.

For the third condition, observe that if one of the two plaintexts of a token is known, then the other can be computed. A token is either closed or open and the instance contains no half-open token.

A.3 A construction based on regular RDH

We describe a construction of SSEwSU based on a regular RDH. When instantiated with the RDH of Appendix A.2, gives the SSEwSU presented in Section 3.

```
EncryptDoc($1^\lambda, \mathcal{D}$)
Executed by Manager to encrypt corpus $\mathcal{D}$
  1. initialize xSet $= \emptyset$;
  2. randomly select four master keys
       $K_1, K_2, K_3 \leftarrow \{0,1\}^\lambda$;
  3. for every document $d$ with metadata $\mathtt{meta}_d$
       set $K_d = \mathsf{F}(K_1, d)$, $\widetilde{K}_d = \mathsf{F}(K_2, d)$;
       set $K_d^{\mathrm{enc}} = \mathsf{G}(K_3, d)$;
       for every keyword $w \in \mathsf{Kw}(d)$:
         set $X_{w,d} = \mathsf{H}(\mathsf{F}(\widetilde{K}_d, d), \mathsf{F}(K_d, w))$;
         set $Y_{w,d} = \mathsf{Enc}(K_d^{\mathrm{enc}}, \mathtt{meta}_d)$;
  4. all the pairs $(X_{w,d}, Y_{w,d})$ are added in random order to the array xSet;
  5. return (xSet, $K_1, K_2, K_3$);
```

```
Enroll($1^\lambda, u$)
Executed by Manager to enroll user $u$
  1. randomly select user key $K_u, \widetilde{K}_u \leftarrow \{0,1\}^\lambda$;
  2. return $K_u, \widetilde{K}_u$;
```

```
SearchQuery$(w, (u, K_u, \widetilde{K}_u), \{(d, K_d, K_d^{\mathrm{enc}})\}_{d \in \mathsf{Access}(u)})$
Executed by user $u$ to search for $w$
  1. for each $(d, K_d, K_d^{\mathrm{enc}})$
       set $\mathtt{uid}_{u,d} = \mathsf{F}(\widetilde{K}_u, d)$;
       set $\mathtt{qct}_d = \mathsf{H}(\mathsf{F}(K_d, w), \mathsf{F}(K_u, d))$;
  2. all the query ciphertexts $(\mathtt{uid}_{u,d}, \mathtt{qct}_d)$ are added in random order to the array $\mathsf{qSet}$;
  3. return $\mathsf{qSet}$;
```

```
SearchReply($\mathsf{qSet}$)
Server replying to $u$'s search with $\mathsf{qSet}$
  1. set $\mathsf{Result} = \emptyset$;
  2. for each query ciphertext $\mathtt{qct} \in \mathsf{qSet}$ and for each token $\mathtt{tok} \in \mathsf{uSet}$
       set $\mathtt{ct} = \mathsf{Apply}(\mathtt{qct}, \mathtt{tok})$;
       if $(\mathtt{ct}, Y) \in \mathsf{xSet}$ then
           add $Y$ to $\mathsf{Result}$;
  3. return $\mathsf{Result}$;
```

```
AccessGranting$((u, K_u, \widetilde{K}_u), d, (K_1, K_2, K_3))$
Executed by Manager to share $d$ with $u$
  1. compute $K_d = \mathsf{F}(K_1, d)$, $\widetilde{K}_d = \mathsf{F}(K_2, d)$, and $K_d^{\mathrm{enc}} = \mathsf{F}(K_3, d)$;
  2. set $\mathtt{uid}_{u,d} = \mathsf{F}(\widetilde{K}_u, d)$;
  3. set $U_{u,d} = \mathsf{GenToken}(\mathsf{F}(K_u, d), \mathsf{F}(\widetilde{K}_d, d))$;
  4. set $\mathcal{K}_d = (d, K_d, K_d^{\mathrm{enc}})$;
  5. return $(\mathtt{uid}_{u,d}, U_{u,d}, \mathcal{K}_d)$;
```

A.4 Simulator

In this section, we show that our construction does not leak any information other than $\mathcal{L}(\mathcal{I}, \mathcal{C})$ to a coalition $\mathcal{C}$ of users about an instance $\mathcal{I}$. We describe a simulator $\mathcal{S}$ for SSEwSU on input of a coalition $\mathcal{C}$ of users along with $\mathcal{L}(\mathcal{I}, \mathcal{C})$, returns a view indistinguishable from a real view of $\mathcal{C}$. We refer the reader to the discussion in Section A.1 for a definition and explanation of the leakage function $\mathcal{L}$.

Let $\mathcal{I}_{\mathsf{RDH}}$ be the instance of RDH associated with an instance $\mathcal{I}$ of SSEwSU. We start by showing how $\mathsf{mL}(\mathcal{I}_{\mathsf{RDH}})$ can be constructed from $\mathcal{L}(\mathcal{I}, \mathcal{C})$. This implies that $\mathcal{S}$ has what it needs to run simulator Sim for $\mathcal{I}_{\mathsf{RDH}}$ and thus obtain simulated $\mathsf{xSet}, \mathsf{uSet}$ and query ciphertexts.

The list of ciphertexts in $\mathcal{I}_{\mathsf{RDH}}$ is easily constructed by observing that $\mathcal{I}_{\mathsf{RDH}}$ include N_{x} x-ciphertexts and $N_Q = \sum_{i \in [q]} |\mathsf{qSet}_i|$ query ciphertexts for a total of $n = N_Q$. To compute the open plaintexts, $\mathcal{S}$ randomly selects K_d, for $d \in \mathsf{Access}(\mathcal{C})$, and computes $\mathsf{F}(K_d, d)$ and $\mathsf{F}(K_d, w)$, for all $w \in \mathsf{Kw}(d)$. Then, it randomly selecting K_u for $u \in \mathcal{C}$ and computes $\mathsf{F}(K_u, d)$ for $d \in \mathsf{Access}(u)$. The matrix L is easily computed by using $(\hat{u}_i, w_i)$ for all half-open query ciphertexts (we remind the reader that these are the only half-open ciphertexts in $\mathcal{I}_{\mathsf{RDH}}$). The matches for $\mathcal{I}_{\mathsf{RDH}}$ can be obtained from the $(\hat{x}_i, \hat{t}_i)$ associated with the successful query ciphertexts (see Point 5.d in the definition of $\mathcal{L}(\mathcal{I}, \mathcal{C})$) and the collisions are constructed from the classes $U_1, \ldots, U_l$ (see Point 6 in the definition of $\mathcal{L}(\mathcal{I}, \mathcal{C})$).

By running Sim with the leakage constructed for instance $\mathcal{I}_{\mathsf{RDH}}$ of RDH, $\mathcal{S}$ obtains all simulated x-ciphertexts and query ciphertexts along with the tokens. To complete the simulation, $\mathcal{S}$ prepares the y-ciphertexts in the following way: for each pair (d, w) such that $d \in \mathsf{Access}(\mathcal{C})$ and $w \in \mathsf{Kw}(d)$, $\mathcal{S}$ uses K_d^{enc} to encrypt $\mathtt{meta}_d$ (see Point 2 in the definition of $\mathcal{L}(\mathcal{I}, \mathcal{C})$); for all remaining pairs (d, w) $\mathcal{S}$ picks a random key $K_{d,w}^{enc}$ and computes an encryption of 0^ℓ (we remind the reader that for all documents d, $|\mathtt{meta}_d| = \ell$).

Finally, let us argue that the output of the simulator is indeed indistinguishable from the view of the coalition. This is using the following intermediate games.

1. Game_0: we modify the real game by replacing each $\mathtt{uid}_{u,d}$ with a randomly chosen element of the group. Since $\mathtt{uid}_{u,d}$ is evaluated exactly once as $\mathsf{F}(\widetilde{K}_u, d)$, Game_0 is indistinguishable from the real game.

2. Game_1: we modify Game_0 by computing each y-ciphertext relative to a document $d \notin \mathsf{Access}(\mathcal{C})$ by encrypting $\mathtt{meta}_d$ with a randomly chosen key instead of using the same key K_d^{enc} for the y-ciphertext relative to the same document.

   Since K_d^{enc} is not revealed for $d \notin \mathsf{Access}(\mathcal{C})$ in the real view then we can use the key obliviousness property of Enc to prove indistinguishability of Game_1 and the view of the coalition.

3. Game_2: we modify Game_1 by computing each y-ciphertext relative to a document $d \notin \mathsf{Access}(\mathcal{C})$ by encrypting 0^ℓ with a randomly chosen key instead of encrypting the real $\mathtt{meta}_d$.

   Since K_d^{enc} is never revealed for $d \notin \mathsf{Access}(\mathcal{C})$ then we can use the CPA security of Enc to prove indistinguishability of Game_2 and Game_1.

4. Game_3: we modify Game_2 by picking the closed plaintext at random (instead of computing them by using F).

   Since that the seed of F used to compute the closed plaintexts does not appear in the real view we can thus use the pseudo-randomness of F to prove indistinguishability of Game_3 and Game_2.

5. Game_4: we modify Game_3 by computing the $\mathsf{xSet}, \mathsf{uSet}$ and the qSet_i using the simulator Sim for RDH instead.

   Indistinguishability of Game_4 and Game_3 follows directly from the property of Sim.

We conclude by observing that the output of Game_4 is exactly the output of $\mathcal{S}$.