Symmetric Searchable Encryption with Sharing and Unsharing

Sarvar Patel[1](✉), Giuseppe Persiano[1,2](✉), and Kevin Yeo[1](✉)

[1] Google LLC, Mountain View, USA
{sarvar,kwlyeo}@google.com
[2] Università di Salerno, Fisciano, Italy
giuper@gmail.com

Abstract. In this paper, we study Symmetric Searchable Encryption (SSE) in a multi-user setting in which each user dynamically shares its documents with selected other users, allowing sharees also to perform searches. We introduce the concept of a *Symmetric Searchable Encryption with Sharing and Unsharing*, an extension of Multi-Key Searchable Encryption (NSDI '14), that supports dynamic sharing and unsharing of documents amongst users. We also strengthen the security notion by considering a simulation-based notion that does not restrict sharing between honest and compromised users.

We present the notion of *cross-user leakage*, the information leaked about a user's documents and/or queries from the queries of other users, and introduce a novel technique to quantify cross-user leakage. Specifically, we model cross-user leakage by using a graph where nodes correspond to users and the presence of edges between two nodes indicates the existence of cross-user leakage between the two adjacent users. The statistics on the connected components of the cross-user leakage graph provide a quantifiable way to compare the leakage of multi-user schemes which has eluded previous works.

Our main technical contribution is mx-u, an efficient scheme with small cross-user leakage, whose security is based on the decisional Diffie-Hellman assumption. We prove a tight bound on the leakage of mx-u in the presence of an honest-but-curious adversary that colludes with a non-adaptively chosen subset of users. We report on experiments showing that mx-u is efficient and that cross-user leakage grows slowly as queries are performed.

Keywords: Cryptography · Cloud storage · Searchable encryption

1 Introduction

Symmetric Searchable Encryption (SSE), introduced by Song *et al.* [29], has been the object of intensive research in the last several years. The original scenario consists of a certain number of CorpusOwners, each with a distinct *corpus*

The full version of this paper can be found at [26].

$\mathcal{D}$ of documents. Each CorpusOwner wishes to store his own $\mathcal{D}$ on the Server in encrypted form and be able to subsequently provide the Server with appropriate search tokens to select the ids of (or pointers to) the documents that contain given keywords. In this context, the honest-but-curious Server tries to learn information about the corpus and queries.

Document sharing rises naturally in large organizations where different subsets of members of the organization collaborate on different documents at any give time and, in real-world scenarios, the ability to revoke access is crucial. For example, in organizations with many employees, it is impossible to assume that employees do not shift to different roles, teams and/or projects. As a member changes their responsibilities, organizations would like to revoke that member's access to documents which are no longer relevant as a way to reduce insider risk. Therefore, efficient and secure revocation is an integral functionality to many multiple user settings for searchable encryption.

In this paper, we introduce *Symmetric Searchable Encryption with Sharing and Unsharing* (SSEwSU), an extension of multi-key searchable encryption introduced by Popa and Zeldovich [27], where access is *dynamic* in the sense that a document *shared* with a user can be, subsequently, *unshared* from the same user. For our security notion, we adapt the simulator paradigm used by most previous works. In particular, we prove that a scheme leaks at most some leakage function, $\mathcal{L}$, by showing the existence of a probabilistically polynomial time (PPT) simulator that can compute a view for the adversary that is indistinguishable from the real view of the adversary. In the case of the single-key/single-user schemes, the adversary is the Server. To adapt to our multi-key/multi-user setting, the adversary is assumed to be the Server colluding with a subset of compromised users.

As we have noted above, the simulation-based security proof indicates that the adversary may learn at most $\mathcal{L}$ information about the documents stored in the scheme as well as the queries performed by the users. However, $\mathcal{L}$ is defined as a description of information instead of a quantifiable value. As a result, it is very difficult to compare the leakage profiles of different schemes. Furthermore, the damage that could be inflicted by an adversary that learns such a leakage profile of the documents and queries is not clear.

In this paper, we quantify the phenomenon, that we call *cross-user leakage*, consisting in information about one user's documents and/or queries being learned by the adversary as the result of the actions by another user. Intuitively, a searchable encryption scheme with good security should guarantee that the actions of a single user do not leak too much information about too many other users. We introduce the concept of a *cross-user leakage graph* to have a quantitative measure of the cross-user leakage. Specifically, each node of the cross-user leakage graph is associated with a user and an edge between two nodes indicates the presence of cross-user leakage between the two adjacent users. Immediately after a system has been initialized and no user has performed any action, the cross-user leakage graph has no edges. As actions are performed, edges are added as cross-user leakage appears. Therefore, the cross-user leakage graph describes the growth of leakage as more actions are performed. A connected component is a maximal subgraph such that all pairs of nodes are connected, and we use statistics on the connected components of the cross-user leakage, such as the number of connected components and means and variances on the sizes of all the connected components, as approximations of the total cross-user leakage learned by the adversary. Our description will be pessimistic and assume that cross-user leakage is transitive. That is, if two users belong to the same connected component, then we will assume that cross-user leakage has occurred between the two users even if there is not an edge between the two users.

Our main contribution is mx-u, a searchable encryption scheme with sharing and unsharing. We formally identify the leakage of our construction and show that is an upper bound on the actual leakage by constructing a simulator that simulates the view of the adversarial coalition on input the leakage. Furthermore, we show experimentally by means of the cross-user leakage graph that the growth of cross-user leakage in mx-u is significantly slower than other known schemes with similar efficiency.

Related Work. The notion of *Symmetric Searchable Encryption* was introduced by Song *et al.* [29] and continues to be an active research area. Boneh *et al.* [3] were the first to consider the asymmetric case. The original scenario consists of the CorpusOwner and the Server. In our terminology, the CorpusOwner outsources the storage of an encrypted version of his data on the Server while being able to delegate searches to the Server while protecting the privacy of the data and queries. This basic setting was extended by Curtmola *et al.* [9], which considered the extension of multiple users authorized by the CorpusOwner. The same setting is considered by several subsequent papers such as [8,12,16,19,24,30]. The work by Cash *et al.* [6] was the first to obtain sub-linear search times for conjunctive keyword searches. The recent work of Kamara and Moataz [17] present a scheme that may handle keyword searches for all Boolean formulae in sub-linear time. The work of Cash *et al.* [5] show that a bottleneck in searchable encryption schemes is the lack of locality of server accesses that incur a large number of cache misses and, subsequently, retrievals from secondary storage. Subsequent works [1,2,7,10,11] address the tradeoffs of locality and the time required for searching. Kamara *et al.* [18] consider the tradeoffs of using Oblivious RAM to further suppress the leakage profiles and the increased efficiency costs. In another line of work, several papers have investigated the amount of information that can be extracted from various leakage profiles [4,14,20,22,28,32,33].

In the work described in the previous paragraph, "multiple users" means users (other than the CorpusOwner) can perform searches using tokens provided by the CorpusOwner. However, all users have access to the same set of documents and all users are, typically, considered to be honest. We are interested in allowing different users access to different and dynamically changing subsets of documents as well as protecting against insider threats (adversarial users colluding with the Server). The concept of Multi-key Searchable Encryption, introduced by Popa and Zeldovich [27], is very close in spirit to our work. However, Grubbs *et al.* [14] point out that the security notion considered by Popa and Zeldovich [27] is insufficient in real settings. In addition, the construction proposed in [27] did not support unsharing documents (that is, revocation of sharing) as opposed to ours that allows for efficient sharing and unsharing. A more efficient scheme was presented by Kiayias *et al.* [21] but it suffers from the same security problems of the construction of [27].

In a parallel and independent work to ours, Hamlin *et al.* [15] present two multi-key searchable encryption schemes that consider a stronger security notion similar to ours. Their first construction is based on the existence of PRFs and uses constant-size query tokens. We note that our construction, $\mathsf{mx\text{-}u}$, requires query tokens that are linear in the number of the documents accessible to the querying user. On the other hand, the first construction of [15], essentially, duplicates a document for each user granted access while $\mathsf{mx\text{-}u}$ maintains a single copy of each document for all users and is, thus, more storage efficient. The second construction by Hamlin *et al.* [15] uses software obfuscation to reduce server storage and cannot be considered practical.

2 Key Ideas

We introduce some basic notation that will be used throughout the paper. A document is a tuple $(d, \mathsf{Kw}(d), \mathtt{meta}_d)$ consisting of an id d, a list $\mathsf{Kw}(d)$ of keywords taken from the universe $\mathcal{W}$ of keywords (e.g., English dictionary) and some metadata $\mathtt{meta}_d$ (e.g., title, snippet, creation time). We denote the subset of documents that are accessible to user $u \in \mathcal{U}$ by $\mathsf{Access}(u)$. Similarly, we denote the subset of users with access to document d by $\mathsf{AccList}(d)$. The set $\mathsf{Access}(u)$ 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 in $\mathsf{Access}(u)$ at the time the search is performed. Typically, the user looks at the metadata of the documents returned by the search to determine which document to fully download. A user u should be able to perform queries only on documents in $\mathsf{Access}(u)$ without further intervention of the $\mathsf{CorpusOwner}$. Moreover, sharing and unsharing of documents should be efficient and require only the $\mathsf{CorpusOwner}$ and the Server to be active.

We start by describing two constructions, $\mathsf{zx\text{-}u}$ and $\mathsf{lx\text{-}u}$, that are direct extensions of single-user SSE schemes into SSEwSU schemes. $\mathsf{zx\text{-}u}$ has no cross-user leakage but is extremely inefficient. In contrast, $\mathsf{lx\text{-}u}$ has very large cross-user leakage but is very efficient.

An Inefficient Construction with no Cross-User Leakage. The first construction we consider, $\mathsf{zx\text{-}u}$ (for *zero cross-user leakage*), consists in having an independent instance of a single-user SSE supporting addition/deletion of documents for each user. When a document d is shared with user u, d is added to u's instance of the single-user SSE. Similarly, when d is unshared with user u, d is removed from u's instance. $\mathsf{zx\text{-}u}$ has no cross-user leakage since each user's queries are performed on their independent single-user SSE instance. 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.

An Efficient Construction with Large Cross-User Leakage. Following a dual approach, $\mathsf{lx\text{-}u}$ (for *large cross-user leakage*) consists of an independent SSE instance for each document d. We refer to K_d as the private key associated to the SSE instance for document d. K_d is given to all users u with access to d. To search, user u sends the Server a search token for each document $d \in \mathsf{Access}(u)$. The per document partition is to ensure proper user access control. The Server stores a list of users with access to each document d in $\mathsf{AccList}(d)$. Therefore, $\mathsf{mx\text{-}u}$ only requires $O(|\mathcal{D}| \cdot |\mathcal{W}| + |\mathcal{D}| \cdot |\mathcal{U}|)$ space.

We show $\mathsf{lx\text{-}u}$ has large cross-user leakage that increases as queries are being performed. 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 in an identical manner using the same algorithm and K_d. So, the Server can use these search tokens for all users in $\mathsf{AccList}(d)$. The 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 users including users that never performed any query. Subsequently, suppose user u_2 searches for w_2. For all documents accessible by both u_1 and u_2 ($\mathsf{Access}(u_1) \cap \mathsf{Access}(u_2)$), the Server knows exactly whether each of the documents contain w_1, w_2, both and neither.

Where Does the Problem Come From? The cross-user leakage of $\mathsf{lx\text{-}u}$ is caused by two factors. First, $\mathsf{lx\text{-}u}$ is partitioned by documents to allow the Server to enforce access control for each document. Secondly, queries by two different users coincide over documents accessible by both users. We see two contrasting needs fighting here: storing keyword occurrences in documents in a user-independent fashion for efficiency forces the queries of different users over the same document to be identical. In other words, when one user queries, it does so on behalf of all the other users.

To overcome this apparent stalemate, we introduce an intermediate level in which the user-dependent queries are translated to user-independent queries that can be matched with encryptions of keyword-document pairs to perform a search. The intermediate level will be implemented by means of tokens that depend on the user and the document and a token will only be provided by $\mathsf{CorpusOwner}$ if the user has access to the document. The introduction of the extra level requires space $O(|\mathcal{D}| \cdot |\mathcal{U}|)$, which is proportional to the size of the original access lists. These ideas lead us to our main contributions.

Our Efficient Construction with Minimal Cross-User Leakage. Our main contribution, $\mathsf{mx\text{-}u}$, is a construction that has minimal cross-user leakage and can be intuitively described in terms of an abstract primitive that we call *Rewritable Deterministic Hashing* (RDH). RDH is an enhancement of a two-argument hash function H. With slight abuse of terminology, we will denote a *ciphertext* as the value obtained by evaluating H on two arguments, which we refer to as the *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. Using this abstraction, mx-u can be roughly described as follows. A more formal and precise description that follows this informal blueprint based on RDH is presented in Sect. 4.

During the initialization phase, the CorpusOwner computes $\mathsf{H}(w, d)$ for every (w, d) such that $w \in \mathsf{Kw}(d)$. We stress that this is done exactly once, independently of the number of users that can access d. In addition, the CorpusOwner produces *authorization* token $\mathtt{tok}_{u \to d}$ for every $d \in \mathsf{Access}(u)$. All the ciperthexts and the tokens are given to the Server. A user u that wishes to perform a query for keyword w computes the *query* ciphertext $\mathsf{H}(u, w)$ and sends it to the server. If u has access to d, the Server has received $\mathtt{tok}_{u \to d}$ from the CorpusOwner and the application of $\mathtt{tok}_{u \to d}$ to the query ciphertext $\mathsf{H}(u, w)$ produces $\mathsf{H}(w, d)$ that, if document d contains keyword w, has been given to the Server by the CorpusOwner during the initialization. A more precise description of mx-u can be found in Sect. 4.

The CorpusOwner produces $\sum_d |\mathsf{Kw}(d)|$ ciphertexts and $\sum_u |\mathsf{Access}(u)|$ tokens. All computed tokens and ciphertexts are stored on the Server for a total space of $O(|\mathcal{D}| \cdot |\mathcal{W}| + |\mathcal{D}| \cdot |\mathcal{U}|)$. This matches the efficiency of lx-u. The rewriting capability of RDH makes it unnecessary to duplicate the pair (w, d) for each user that has access to document d like zx-u. So, mx-u has greatly improved efficiency compared to zx-u. Unlike lx-u, mx-u achieves efficiency with only minimal cross-user leakage. We show that mx-u has cross-user leakage only when two distinct users $u_1 \neq u_2$ who share at least one common document $(\mathsf{Access}(u_1) \cap \mathsf{Access}(u_2) \neq \emptyset)$ query for the *same* keyword. A formal analysis of mx-u's leakage can be found in Sect. 4.1. In addition to searching, mx-u supports sharing and unsharing of documents very efficiently. Sharing (unsharing) a document d with user u simply adds (removes) the authorization token $\mathtt{tok}_{u \to d}$ from storage on the Server. So, sharing and unsharing can be performed in constant time.

We perform extensive experimentation on mx-u and shows that it is practical. We also experimentally show that the cross-user leakage for mx-u accumulates significantly slower compared to lx-u using real world data. Our experiments in Sect. 5 show that mx-u hits the middle ground between the two extremes by providing the same efficiency as lx-u with minimal cross-user leakage like zx-u.

3 Symmetric Searchable Encryption with Sharing and Unsharing

We formally define the concept of a *Symmetric Searchable Encryption with Sharing and Unsharing* (SSEwSU). We provide definitions and algorithms for the case of one CorpusOwner. The extensions to a more complex setting in which several CorpusOwners share documents to a set of users is obtained by considering an independent system for each CorpusOwner. A SSEwSU is a collection of six algorithms: EncryptDoc, Enroll, SearchQuery, SearchReply, AccessGranting, AccessRevoking for three types of players: one CorpusOwner, one Server and several users. The algorithms interact in the following way.

1. $\mathsf{CorpusOwner}$ 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 $\mathtt{meta}_d$. $\mathsf{CorpusOwner}$ computes an encrypted version xSet of $\mathcal{D}$ and a master key K by running algorithm $\mathsf{EncryptDoc}(\mathcal{D})$. $\mathsf{CorpusOwner}$ sends xSet to Server and keeps K in private memory. $\mathsf{CorpusOwner}$ instructs Server to initialize uSet to be empty. Both xSet and uSet are kept private by Server.
2. $\mathsf{CorpusOwner}$ executes Enroll to add a new user u to the system. Keys $\mathcal{K}_u$ for u are returned by the algorithm. $\mathsf{CorpusOwner}$ 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, $\mathsf{Access}(u) := \mathsf{Access}(u) \cup \{d\}$), $\mathsf{CorpusOwner}$ 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 $\mathsf{CorpusOwner}$ 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 $u \in \mathcal{C}$.

Our definition is tailored for a static corpus of documents (no document is added and/or edited). This is reflected by the fact that $\mathsf{CorpusOwner}$ computes the encrypted version of the corpus by using $\mathsf{EncryptDoc}$ at the start. Note that, even though the corpus of documents is static, each document can be dynamically shared and unshared.

Security Notion for SSEwSU. We give our security definition for SSEwSU by following the real vs. simulated game approach. In our trust model, we assume the $\mathsf{CorpusOwner}$ to be honest. If this is not the case, since $\mathsf{CorpusOwner}$ has access to its whole corpus $\mathcal{D}$, then there is nothing to be protected from compromised users. We assume the Server is honest-but-curious and computes query results as prescribed by $\mathsf{SearchReply}$, stores xSet as received from the $\mathsf{CorpusOwner}$, and updates uSet as instructed by the $\mathsf{CorpusOwner}$. In traditional SSE schemes, the Server is assumed to be curious with access to all observed ciphertexts (in this case, the xSet, the uSet and all queries). In the multi-user setting of SSEwSU, we have to consider that the Server might be colluding with a set of active, compromised users, $\mathcal{C}$. The Server gains access to all private keys of every compromised user in $\mathcal{C}$.

In the real game with a set of users $\mathcal{U}$, we consider the *server view*, $\mathsf{sView}^{\mathcal{U},\mathcal{C}}$ where the Server colludes with the non-adaptively chosen subset $\mathcal{C} \subseteq \mathcal{U}$ of users gaining access to all user keys of compromised users, $\{K_u\}_{u\in\mathcal{C}}$, and all document keys where the document is accessible by any compromised user, $\{\mathcal{K}_d\}_{d\in\mathsf{Access}(\mathcal{C})}$.

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 CorpusOwner 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: a set of documents $\{d, \mathsf{Kw}(d), \mathtt{meta}_d\}_{d\in\mathcal{D}}$, a collection $\{\mathsf{Access}(u)\}_{u\in\mathcal{U}}$ of subsets of document ids, the set of search queries $Q_i = (u_i, w_i)$, for $i \in [q]$, and the i-th query is performed by user u_i for keyword w_i. We define the view with respect to security parameter λ of the Server colluding with the set, $\mathcal{C}$, of compromised users on instance $\mathcal{I}$ of SSEwSU, as the output of the Real 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 $i \in [q]$
       $\mathsf{qSet}_i \leftarrow \mathsf{SearchQuery}(w_i, K_{u_i}\{(d, K_d, K_d^{\mathrm{enc}})\}_{d\in\mathsf{Access}(u_i)})$;
       $\mathsf{Result}_i \leftarrow \mathsf{SearchReply}(\mathsf{qSet}_i)$;
  5. Output $(K_u, \mathcal{K}_u)_{u\in\mathcal{C}}, \mathsf{xSet}, \mathsf{uSet}, (\mathsf{qSet}_i, \mathsf{Result}_i, )_{i\in[q]}$;
```

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 $u \in \mathcal{U}$.

We now present a formal definition of our security notion.

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 $\mathcal{I}$*

$$\{\mathsf{sView}^{\mathcal{U},\mathcal{C}}(\lambda, \mathcal{I})\} \approx_c \{\mathcal{S}(1^\lambda, \mathcal{L}(\mathcal{I}, \mathcal{C}))\}.$$

4 SSEwSU Based on Decisional Diffie-Hellman

In this section, we describe mx-u, a concrete construction of SSEwSU based on decisional Diffie-Hellman (DDH) that follows the blueprint based on the concept of a *Rewritable Deterministic Hashing* (RDH) (see Sect. 2 for an informal description). We perform an experimental evaluation of mx-u to evaluate both the leakage and performance in Sect. 5.

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 group elements. The occurrence of $w \in \mathsf{Kw}(d)$ is encoded by CorpusOwner 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 the Server. The fact that $u \in \mathsf{AccList}(d)$ is encoded by computing *authorization token* d/u and giving it to the Server. The set of all x-pairs and authorization tokens produced by CorpusOwner are called the xSet and uSet respectively. To search for a keyword w in $\mathsf{Access}(u)$, user u produces the *query* consisting of $u \cdot w$. The 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 u_2's ability to perform queries to all documents in $\mathsf{Access}(u_1) \cup \mathsf{Access}(u_2)$.

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, that is, d/u. A *query* is computed as $g^{u \cdot w}$ as well as pointers to relevant authorization tokens. In performing the search, the 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 does not suffice as the set of documents and keywords might be small enough for the Server to conduct dictionary attacks. Instead, we replace document ids, user ids and keywords with their evaluations of pseudorandom functions under the appropriate document and user keys. Document keys are distributed depending on whether a document is shared with a user while each user only has their own user key. 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, uSet and all keys from compromised users. Server also knows the patterns of accessing the xSet and uSet when users are performing search queries.

We now formally present the algorithms of mx-u. Both AccessGranting and AccessRevoking will be implemented using a single algorithm 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 authorization token $U_{u,d}$ allowing u to access document d, pointer (identifier) to the authorization token $\mathtt{uid}_{u,d}$ and the set of keys $\mathcal{K}_d$ for document d. Algorithm AccessGranting is executed by CorpusOwner 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 the pair $(\mathtt{uid}_{u,d}, U_{u,d})$ that are sent to the 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 the 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 the Server will not contribute the y-ciphertext to Result even if $w \in \mathsf{Kw}(d)$.

```
$\mathsf{EncryptDoc}(1^\lambda, \mathcal{D})$
Executed by CorpusOwner to encrypt the corpus $\mathcal{D}$.
  1. Randomly select $(g, \mathcal{G}) \leftarrow \mathcal{GG}(1^\lambda)$ and initialize $\mathsf{xSet} \leftarrow \emptyset$;
  2. Randomly select three master keys $K_1, K_2, K_3 \leftarrow \{0,1\}^\lambda$;
  3. For every document $d$:
       – Set $K_d \leftarrow \mathsf{F}(K_1, \mathrm{d})$;
       – Set $\widetilde{K}_d \leftarrow \mathsf{F}(K_2, \mathrm{d})$;
       – Set $K_d^{\mathrm{enc}} \leftarrow \mathsf{G}(K_3, \mathrm{d})$;
  4. For every document $d$ with metadata $\mathtt{meta}_d$ and keyword $w \in \mathsf{Kw}(d)$:
       – Set $X_{w,d} \leftarrow g^{\mathsf{F}(\widetilde{K}_d, d) \cdot \mathsf{F}(K_d, w))}$;
       – Set $Y_{w,d} \leftarrow \mathsf{Enc}(K_d^{\mathrm{enc}}, \mathtt{meta}_d)$;
  5. All pairs $(X_{w,d}, Y_{w,d})$ are added in random order to the array $\mathsf{xSet}$;
  6. Return $(\mathsf{xSet}, K_1, K_2, K_3)$;
```

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

```
$\mathsf{AuthComputing}((u, K_u, \widetilde{K}_u), d, (K_1, K_2, K_3))$
Executed by CorpusOwner to either share or unshare document $d$ with user $u$.
  1. Compute keys $K_d \leftarrow \mathsf{F}(K_1, d)$, $\widetilde{K}_d \leftarrow \mathsf{F}(K_2, d)$, and $K_d^{\mathrm{enc}} \leftarrow \mathsf{G}(K_3, d)$;
  2. Set $U_{u,d} \leftarrow \mathsf{F}(\widetilde{K}_d, d)/\mathsf{F}(K_u, d)$;
  3. Set $\mathtt{uid}_{u,d} \leftarrow \mathsf{F}(\widetilde{K}_u, d)$;
  4. Set $\mathcal{K}_d \leftarrow (d, K_d, K_d^{\mathrm{enc}})$;
  5. Return $(\mathtt{uid}_{u,d}, U_{u,d}, \mathcal{K}_d)$;
```

```
$\mathsf{SearchQuery}(w, (u, K_u, \widetilde{K}_u), \{(d, K_d, K_d^{\mathrm{enc}})\}_{d \in \mathsf{Access}(u)})$
Executed by user $u$ to search for documents with keyword $w$.
  1. For each $(d, K_d, K_d^{\mathrm{enc}})$,
       – Set $\mathtt{uid}_{u,d} \leftarrow \mathsf{F}(\widetilde{K}_u, d)$;
       – Set $\mathtt{qct}_d \leftarrow g^{\mathsf{F}(K_d, w) \cdot \mathsf{F}(K_u, d)}$;
  2. All pairs $(\mathtt{uid}_{u,d}, \mathtt{qct}_d)$ 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 $\mathsf{Result} \leftarrow \emptyset$;
  2. For each $(\mathtt{uid}_{u,d}, \mathtt{qct}_d) \in \mathsf{qSet}$:
       – Set $\mathtt{ct} \leftarrow \mathtt{qct}_d^{\mathsf{uSet}[\mathtt{uid}_{u,d}]}$;
       – If $(\mathtt{ct}, Y) \in \mathsf{xSet}$, then $\mathsf{Result} \leftarrow \mathsf{Result} \cup \{Y\}$;
  3. Return $\mathsf{Result}$;
```

4.1 The Leakage Function $\mathcal{L}$

In this section, we formally define the leakage $\mathcal{L}(\mathcal{I}, \mathcal{C})$ that Server obtains about 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 (see the full version [26]), 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.

A Warm-up Case. We start by informally describing the leakage obtained by Server when no user is compromised ($\mathcal{C} = \emptyset$). Looking ahead, the leakage for $\mathcal{C} = \emptyset$ corresponds to items 0 and 7 in the general case when $\mathcal{C} \neq \emptyset$.

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 $\mathsf{mx\text{-}u}$, the 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 $\mathsf{lx\text{-}u}$, the Server would learn whether two queries are relative to the same document and this allowed the propagation of cross-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 $\mathsf{mx\text{-}u}$. In other words, the only way to have cross-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 cross-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 cross-user leakage each query entails. For both constructions, the graph starts with no edges and edges are added as queries are performed. In $\mathsf{lx\text{-}u}$, 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 $\mathsf{mx\text{-}u}$, 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, cross-user leakage accumulates for every query in $\mathsf{lx\text{-}u}$ whereas in $\mathsf{mx\text{-}u}$ cross-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 test value for the xSet (that belongs to the 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 $D_1, \dots, D_r$. 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 $E_1, \dots, E_t$. 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 distinct for clarity.

The General Case. We now consider the case where the adversarial Server corrupts a subset $\mathcal{C} \neq \emptyset$ of users. As we shall see, in this case, all information about documents shared to users in $\mathcal{C}$ are leaked to the 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 the Server, then we observe that the 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. 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. $\mathsf{Access}(u)$ of documents that have been shared with $u \in \mathcal{C}$;
2. $\mathsf{Kw}(d)$ of keywords and the metadata $\mathtt{meta}_d$, for each $d \in \mathsf{Access}(\mathcal{C})$;
3. $\mathsf{AccList}(d)$ 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 $X_{w,d} = g^{\mathsf{F}(\widetilde{K}_d,d)\cdot\mathsf{F}(K_d,w)}$. 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 $j \le i$ 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 $\mathtt{qct}_d \in \mathsf{qSet}_i$ (the i-th query for keyword w_i 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 next theorem (see the full version [26] for the proof), we 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 showing that there exists 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.

We start by reviewing the DDH Assumption and then state our main result. 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 $\mathcal{G}$.

Definition 2. *The Decisional Diffie-Hellman (DDH) assumption holds for group generator $\mathcal{GG}$ if distributions D^0_λ and D^1_λ are computational indistinguishable, where* $D^\xi_\lambda = \left\{(g, \mathcal{G}) \leftarrow \mathcal{GG}(1^\lambda); x, y, r \leftarrow \mathbb{Z}_{|\mathcal{G}|} : (g^x, g^y, g^{x \cdot y + \xi \cdot r})\right\}$.

Theorem 1. *Under the DDH assumption,* mx-u *is secure with respect to leakage $\mathcal{L}$ as defined in Sect. 4.1.*

For the proof of the theorem above, we will use the following assumption that is equivalent to the DDH Assumption. Let $\mathbf{x} \in \mathbb{Z}_p^{l_0}$, $\mathbf{y} \in \mathbb{Z}_p^{l_1}$. Then, $\mathbf{x} \times \mathbf{y}$ is the $l_0 \times l_1$ matrix whose (i, j)-entry is $x_i \cdot y_j$. For a matrix $A = (a_{i,j})$, we set $g^A = (g^{a_{ij}})$.

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

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

5 Experiments

In this section, we investigate the costs of mx-u 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 mx-u. In practice, mx-u 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 (Enc, 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.

5.1 Performance

We measure the computation time and bandwidth of uploading and searching documents of mx-u (see Fig. 1(a)–(d)). 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.

Enron Email Dataset. We consider using mx-u to store the Enron email dataset [13]. We have 150 users and 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 access. Every recipient of the email will be given access with $\frac{1}{2}$ probability. The server storage required is 5–6 times the size of the emails being uploaded (see Fig. 1(e)). We remark that SSE might be insecure for emails (e.g., see injection attacks of [33]) and we use the dataset as a means to test practicality.

Ubuntu Chat Corpus. In a separate experiment, we store the Ubuntu Chat Corpus (UCC) [31] 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 and the server storage growth with the number of days considered is found in Fig. 1(f). Stemming removes common words as well as providing pseudonyms.

5.2 Leakage Growth

Figures 1(g)–(i) report the results of experiments in which we compare the rate of leakage growth of lx-u and mx-u as queries are performed[1]. The cross-user leakage graph G for mx-u has a node for each vertex and we have 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 mx-u, 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 lx-u if both users share at least one document in common and either user ever queried. Furthermore, cross-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, cross-user leakage grows. If G has no edges (and consists of $|\mathcal{U}|$ connected components, one for each user), there is no cross-user leakage, and this is the status of the system before queries have been performed. Conversely, the complete graph has cross-user leakage for every pair of users. The vector of the sizes of the connected components of G at any given point in time describes the current cross-user leakage. The initial vector consists of $|\mathcal{U}|$ 1's (each vertex is in a connected component by itself). We pad with 0's to keep the vector of dimension $|\mathcal{U}|$. We measure how the length of this vector grows as queries are performed by looking at the L_2 norm (the square root of the sum of the squares of component sizes) and the L_∞ norm (the size of the largest component). We also plot the total number of components.

We compute these metrics for both lx-u and mx-u, using 2500 days of UCC data with approximately 55000 users. Keywords are drawn from the global distribution of terms in UCC after stemming. The user that performs the query is drawn uniformly at random from all users. We see that mx-u leakage grows significantly slower than lx-u in all three metrics. In particular, for all three metrics, lx-u approaches a single connected component after about 100 queries. For mx-u, 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.

6 Extensions

In the full version of this paper, we discuss several extensions to our results.

[1] Note that all considerations about leakage growth apply to mx-u independently of the underlying RDH.

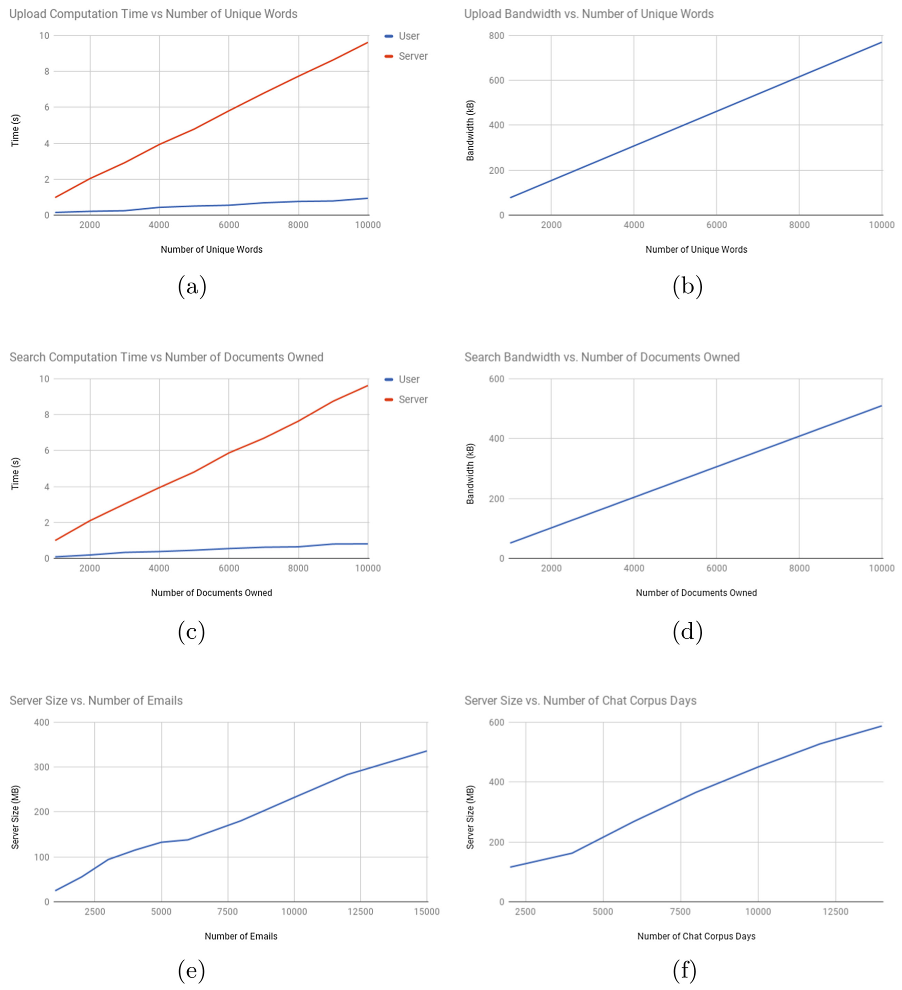
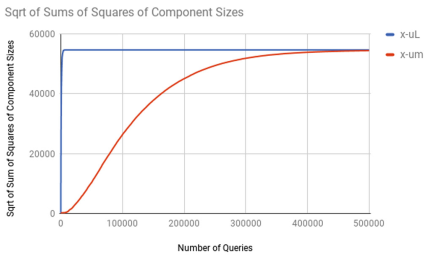
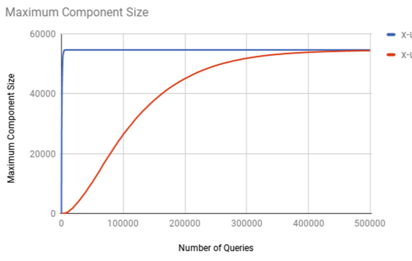


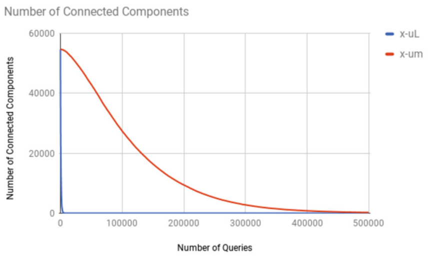
Fig. 1. (a) Upload computation time, (b) Upload bandwidth, (c) Search computation time, (d) Search bandwidth, (e) Enron Email server storage, (f) Ubuntu Chat server storage, (g) Sqrt sum of squares of component sizes, (h) Maximum component size, (i) Number of components.



(g)



(h)



(i)

Fig. 1. (*continued*)

Different Access Types. In the above constructions, we simply designate access as *search access* or the ability to search for keywords in documents. We show that the techniques in mx-u can be extended to handle multiple access types (such as ownership or editing). The main idea is to duplicate the uSet for each type of access type that we wish to enforce.

Reducing Leakage using Bilinear Pairings. We present a scheme that uses bilinear pairings which reduces the cross-user leakage compared mx-u in exchange for larger server computation. In particular, we show that cross-user leakage for two different users querying for the same keyword is present if and only if the keyword exists in a document shared by both users. However, the server must now perform computation for all elements in the xSet.

*Sharing by Non-*CorpusOwner *Users.* In mx-u, we only present techniques that CorpusOwner may construct authorization tokens to enable other users to search over documents in CorpusOwner's corpus. We present an algorithm that allows a non-CorpusOwner user that already has access to a document to share with another user without requiring the interaction of CorpusOwner.

Key Rotation by CorpusOwner*.* We present an efficient key rotation algorithm that can be performed by CorpusOwner for mx-u while simultaneously trying to prevent cross-user leakage accumulation. The goal is to rotate all the keys of ciphertexts in both the uSet and xSet while ensuring that the Server cannot determine whether an old and a new ciphertext contain the same plaintexts under different keys. We will use an oblivious shuffling algorithm [23] to achieve our goal by randomly permuting the elements of the xSet in a manner oblivious to Server. During the shuffle, CorpusOwner will select new keys and replace old RDH evaluations with new RDH evaluations under the new keys. As our shuffling algorithm, we use the K-oblivious shuffling variant of the CacheShuffle [25] which can leverage the fact that some N - K items of the xSet are never involved in searches to improve the efficiency of the shuffling algorithm.

7 Conclusions

In this work, we introduce the concept of *Symmetric Searchable Encryption with Sharing and Unsharing*, which extends previous multi-user definitions by requiring dynamic access allowing both sharing and unsharing. We present the *cross-user leakage graph*, a novel method to bound and quantify the leakage of searchable encryption schemes in multiple user settings. As a result, we are able to quantitatively compare different schemes, which is an important need that previous techniques had yet to achieve. As our main technical contribution, we present mx-u. We directly compare mx-u with previous schemes and show that other schemes are either inefficient or have greater cross-user leakage than mx-u.

References

1. Asharov, G., Naor, M., Segev, G., Shahaf, I.: Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In: Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing, pp. 1101–1114. ACM (2016)
2. Asharov, G., Segev, G., Shahaf, I.: Tight tradeoffs in searchable symmetric encryption. Cryptology ePrint Archive, Report 2018/507 (2018). https://eprint.iacr.org/2018/507
3. Boneh, D., Di Crescenzo, G., Ostrovsky, R., Persiano, G.: Public key encryption with keyword search. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 506–522. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_30
4. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, pp. 668–679 (2015)
5. Cash, D., et al.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: NDSS, vol. 14, pp. 23–26. Citeseer (2014)
6. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_20
7. Cash, D., Tessaro, S.: The locality of searchable symmetric encryption. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 351–368. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_20

8. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_33. Also Cryptology ePrint Archive, Report 2006/210
9. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, pp. 79–88 (2006). Also Cryptology ePrint Archive, Report 2006/210
10. Demertzis, I., Papadopoulos, D., Papamanthou, C.: Searchable encryption with optimal locality: achieving sublogarithmic read efficiency. Cryptology ePrint Archive, Report 2017/749 (2017). https://eprint.iacr.org/2017/749
11. Demertzis, I., Papamanthou, C.: Fast searchable encryption with tunable locality. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1053–1067. ACM (2017)
12. Dong, C., Russello, G., Dulay, N.: Shared and searchable encrypted data for untrusted servers. In: Atluri, V. (ed.) DBSec 2008. LNCS, vol. 5094, pp. 127–143. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70567-3_10
13. EDRM (EDRM.net): Enron data set. http://www.edrm.net/resources/data-sets/edrm-enron-email-data-set
14. Grubbs, P., McPherson, R., Naveed, M., Ristenpart, T., Shmatikov, V.: Breaking web applications built on top of encrypted data. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1353–1364. ACM (2016)
15. Hamlin, A., Shelat, A., Weiss, M., Wichs, D.: Multi-key searchable encryption, revisited. Cryptology ePrint Archive, Report 2018/018 (2018). https://eprint.iacr.org/2018/018
16. Kamara, S., Lauter, K.: Cryptographic cloud storage. In: Sion, R., et al. (eds.) FC 2010. LNCS, vol. 6054, pp. 136–149. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14992-4_13
17. Kamara, S., Moataz, T.: Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10212, pp. 94–124. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56617-7_4
18. Kamara, S., Moataz, T., Ohrimenko, O.: Structured encryption and leakage suppression. Cryptology ePrint Archive, Report 2018/551 (2018). https://eprint.iacr.org/2018/551
19. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 965–976. ACM (2012)
20. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic attacks on secure outsourced databases. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1329–1340. ACM (2016)
21. Kiayias, A., Oksuz, O., Russell, A., Tang, Q., Wang, B.: Efficient encrypted keyword search for multi-user data sharing. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9878, pp. 173–195. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45744-4_9
22. Naveed, M., Kamara, S., Wright, C.V.: Inference attacks on property-preserving encrypted databases. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 644–655. ACM (2015)

23. Ohrimenko, O., Goodrich, M.T., Tamassia, R., Upfal, E.: The melbourne shuffle: improving oblivious storage in the cloud. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8573, pp. 556–567. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43951-7_47
24. Pappas, V., et al.: Blind seer: a scalable private DBMS. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP 2014, pp. 359–374. IEEE Computer Society (2014)
25. Patel, S., Persiano, G., Yeo, K.: CacheShuffle: an oblivious shuffle algorithm using caches. arXiv preprint arXiv:1705.07069 (2017)
26. Patel, S., Persiano, G., Yeo, K.: Symmetric searchable encryption with sharing and unsharing. Cryptology ePrint Archive, Report 2017/973 (2017). https://eprint.iacr.org/2017/973
27. Popa, R.A., Zeldovich, N.: Multi-key searchable encryption. Cryptology ePrint Archive, Report 2013/508 (2013)
28. Pouliot, D., Wright, C.V.: The shadow nemesis: inference attacks on efficiently deployable, efficiently searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, pp. 1341–1352 (2016)
29. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of the 2000 IEEE Symposium on Security and Privacy, pp. 44–55 (2000)
30. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS, vol. 71, pp. 72–75 (2014)
31. Uthus, D.: Ubuntu chat corpus. http://daviduthus.org/UCC/
32. Van Rompay, C., Molva, R., Önen, M.: A leakage-abuse attack against multi-user searchable encryption. Proc. Priv. Enhanc. Technol. **2017**(3), 168–178 (2017)
33. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of file-injection attacks on searchable encryption. Cryptology ePrint Archive, Report 2016/172 (2016). http://eprint.iacr.org/2016/172