P-McDb: Privacy-preserving Search using Multi-cloud Encrypted Databases

Shujie Cui, Muhammad Rizwan Asghar, Steven D. Galbraith, Giovanni Russello
The University of Auckland, Auckland, New Zealand
Email: scui379@aucklanduni.ac.nz, {r.asghar, s.galbraith, g.russello}@auckland.ac.nz

***Abstract*—Searchable Symmetric Encryption (SSE) allows users to execute encrypted queries over encrypted databases. A large number of SSE schemes have been proposed in the literature. However, most of them leak a significant amount of information that could lead to inference attacks.**

In this work, we propose an SSE scheme for a Privacy-preserving Multi-cloud encrypted Database (*P-McDb*), which aims at preventing inference attacks. *P-McDb* allows users to execute queries in an efficient sub-linear manner without leaking search, access and size patterns. We have implemented a prototype of *P-McDb* and show its practical efficiency.

I. Introduction

Searchable Symmetric Encryption (SSE) schemes allow users to perform encrypted queries over encrypted data. SSE is very useful in a scenario where sensitive data is outsourced to a Cloud Service Provider (CSP): by performing encrypted queries over encrypted data, the CSP never gets access to the data (and queries) in cleartext.

In the literature, a long line of recent work has investigated SSE with flexible functionality and better performance [1]–[12]. Although these schemes are secure in certain models under various cryptographic assumptions, the CSP is still able to learn some information about the data from the access patterns of users performing search operations. For instance, the CSP is able to see which encrypted data is accessed by a given query by looking at the matching records (referred to as *access pattern leakage*). The CSP can also infer if two or more queries are equivalent (referred to as *search pattern leakage*) by comparing the encrypted queries or matched records. Last but not least, the CSP can simply log the number of matched records or files returned by each query (referred to as *size pattern leakage*).

When an SSE scheme supports insert and delete operations, it is referred to as a *dynamic* SSE scheme. Dynamic SSE schemes might leak extra information if they do not support *forward privacy* and *backward privacy* properties. Lacking forward privacy means that the CSP can learn if newly inserted data or updated data matches previously executed queries. Lacking backward privacy means that the CSP learns if deleted data matches new queries.

Supporting forward and backward privacy is fundamental to limit the power of the CSP to collect information on how the data evolves over time. Only a few of the existing schemes [7], [13], [14] support forward privacy, but no scheme is able to support both properties simultaneously.

Some recent works [15]–[18] have shown that these leakages can be exploited to learn sensitive information and break the scheme. Naveed *et al.* [16] recover more than 60% of the data in CryptDB [6] using frequency analysis. Zhang *et al.* [17] further investigate the consequences of leakage by injecting chosen files or records into the encrypted database. Based on the access pattern, they could recover a very high fraction of searched keywords by injecting a small number of known files into the database. Liu *et al.* [18] demonstrate that the search pattern can be used to reveal the underlying keywords in queries.

Another issue with SSE schemes is a very inflexible key management mechanism. Some schemes, like [4], encrypt the data and queries with the key shared among all the users. Consequently, all the queries and search results issued by one user could be decrypted by all authorised users. Even worse, when one user is revoked, the single key has to be changed and the data has to be re-encrypted with the new key. In other schemes, such as [2] and [8], the keys are only known to the data owner. The users have to send the query and search results to the data owner to get the search tokens and cleartext results, which means that the data owner represents a bottleneck in the system.

In this paper, we present an SSE scheme for multi-cloud environments named Privacy-preserving Multi-cloud Database (*P-McDb*). *P-McDb* can effectively resist attacks based on the access pattern. Our key technique is to use two cloud servers, that are assumed not to collude: one server stores the data and performs the search operation, the other manages re-randomisation and shuffling of the database and coordinates the sub-linear search. (Sub-linear search means that the search operation does not have to be performed on every record in the database.) A user with access to both servers can perform an encrypted search in sub-linear time without leaking the search pattern, access pattern, or size pattern. Furthermore, we handle the situation where many users have access to the same database. Each user is able to protect her queries and search results against all the other entities. Last but not least, *P-McDb* is more efficient for user revocation than most of the existing SSE schemes: Instead of changing the key and re-encrypting the data, we only need to inform all the CSPs to stop any service for the revoked user. Although the revoked users still own the key, they are unable to issue queries or recover the data even if they collude with one of the CSPs.

The rest of this paper is organised as follows. We review

some background and related work in Section II. In Section III, we provide an overview of our approach. In Section IV, we define some notations. Solution and construction details can be found in Section V. Section VI reports the performance. We analyse the potential limitations of our scheme and give a possible solution in Section VII. Finally, we conclude this paper in Section VIII.

II. Background and Related Work

Since the seminal paper by Song *et al.* [10], many SSE schemes have been proposed and the research in this area has been extended in several directions.

In the remainder of the paper, we use the following security notions: **Search Pattern Privacy (SPP)** means the CSP is not able to learn if two (or more) queries are the same or not. **Access Pattern Privacy (APP)** means the CSP is unable to infer if two (or more) result sets contain the same records or not. **Size Pattern Privacy (SzPP)** is achieved if the CSP is unable to learn the number of records that match the query. **Forward Privacy** means the CSP does not learn if a new or updated record matches a query executed in the past. **Backward Privacy** means the CSP is unable to executed queries on records that have previously been deleted or modified.

Curtmola *et al.* [4] introduce a multi-user scheme by combining a single-user SSE scheme with a broadcast encryption. However, the data stored on the CSP is encrypted with the key K shared among all the users, which means the revoked users can still recover all the data if they collude with the CSP. The multi-user SSE scheme proposed by Jarecki *et al.* [19] has the same problem, *i.e.,* it needs to regenerate a new key and re-encrypt the data with the new key when a user is revoked. Hang *et al.* [20] and Ferretti *et al.* [21] present two different collusion-resistant mechanisms that support multi-user access to the outsourced data. Although they support approaches without sharing the keys among users, in both, after user revocation, it is necessary to generate a new key and re-encrypt the data.

Asghar *et al.* [1] propose a multi-user scheme with an efficient and flexible key management method, where each user has her own private key and does not require any re-encryption when an authorised user is revoked. However, this scheme is not secure if a user colludes with the CSP.

A thorough and up-to-date survey and comparison of the current literature can be found in our paper [22].

III. Solution Overview

A. System Model

In the following, we define our system model to describe the entities involved in *P-McDb*, as shown in Figure 1:

- **Admin**: An admin is responsible for the setup and maintenance of databases, user management as well as specification and deployment of access control policies.
- **User**: It is an entity that represents a user who joins the system, if granted by the admin. A user can issue queries according to deployed access control policies.

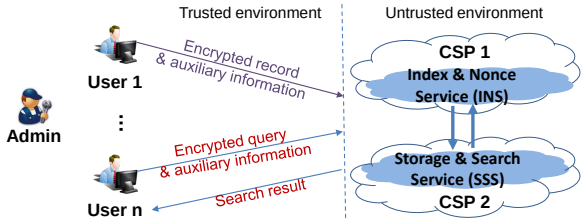


Fig. 1. An overview of *P-McDb*: Users can upload records and issue queries. The SSS and the INS represent independent CSPs. The SSS stores encrypted records and executes queries. The INS stores auxiliary information to ensure search efficiency and privacy. The INS provides auxiliary information to the SSS for performing search. After each query, the SSS sends a set of records to the INS for shuffling and re-randomising to protect patterns privacy.

- **Storage and Search Service (SSS)**: It provides encrypted data storage, executes encrypted queries and returns matching records in an encrypted manner.
- **Index and Nonce Service (INS)**: It stores and manages auxiliary information necessary to ensure search efficiency and privacy. Precisely, it stores nonces that are needed to retrieve data, performs shuffling and re-randomisation of the database, has some control over the number of dummy records, and provides an index service that enables sub-linear search. After each query, it updates the database and auxiliary information to achieve privacy. The INS has no access to the encrypted data.
- **Cloud Service Providers (CSPs)**: Each of the INS and the SSS is deployed on the infrastructure managed by a separate CSP. The CSPs have to ensure that there is a two-way communication between the INS and SSS, but our model assumes there is no collusion between the CSPs.

B. Threat Model

We assume the admin is fully trusted. All the users are semi-trusted in our model. That is, they are only assumed to securely store their keys and the data, however, malicious and revoked users may try to learn what other users are searching for and the result set, or they can collude with the SSS or INS (or the CSPs hosting them) to get access to the data they otherwise are not authorised to access.

The CSPs hosting the SSS and the INS are modelled as honest-but-curious. More specifically, they honestly perform the operations requested by users according to the designated protocol specification. However, they are curious to learn sensitive information by analysing the stored and exchanged data. We assume both the SSS and the INS are part of the public cloud infrastructures provided by different CSPs. According to the latest report given by Rightscale [23], organisations are using more than three public CSPs on average, which means the schemes based on multi-cloud are feasible for most organisations. We emphasise that the CSPs are assumed not to collude. In practice, any two competitive cloud providers, such as Amazon S3, Google Drive and Microsoft Azure, could be considered since they may be less likely to collude in an attempt to gain information from their customers. In case they

collude, pattern privacy will no longer be guaranteed, although the data still remains protected. Similarly, we assume a user cannot collude with the two CSPs at the same time. Solutions to resist these collusion attacks are given in Section VII. We do not consider active attacks and assume that there are mechanisms in place for ensuring data integrity and availability of the system.

C. Approach Overview

P-McDb aims at providing pattern privacy for minimising information leakage from queries. *P-McDb* also achieves both backward and forward privacy by using nonces in the data and queries. To achieve search pattern privacy, we encrypt queries and use nonces. Access pattern and size pattern privacy are achieved by shuffling records. In order to achieve efficiency, an indexing mechanism is implemented: we divide the data into groups and build an index for each group, which enables sub-linear search. To support forward and backward privacy, *P-McDb* uses nonces in the data and queries. In *P-McDb*, it is efficient to revoke users because no key re-generation and data re-encryption operations are needed. To combine indexing and shuffling and pattern privacy, it is necessary to have more than one CSP. We now give an overview of our approach.

To protect the search pattern, we use a semantically secure encryption algorithm. Hence, identical queries will look different once encrypted. However, the SSS may still infer the search pattern by looking at the access pattern. That is, by looking at the physical locations of the encrypted records returned by the search operation, the SSS can infer that two queries are equivalent if the same records are returned.

To address this issue, after executing each query, we shuffle the locations of the searched records. Moreover, prior to re-inserting these records, we re-randomise the ciphertexts, making them untraceable. Technically, after each search operation, we update all the records touched during the search operation. In this way, even if a query equivalent to the previous one is executed, the SSS will see a new set of records being searched and returned, and cannot easily infer the access pattern. This also implies that the search pattern remains protected.

Another form of leakage is size pattern leakage, where an adversary can learn the number of records returned after performing a query, referred to as a *count attack* in [15]. Moreover, even after shuffling and re-randomisation, an adversary could guess whether two queries are equivalent by checking the number of records in the result set. To protect the size pattern, we introduce a random number of dummy records. Dummy records look exactly as the real ones and can even match queries. The search result for each query will contain a number of dummy records making it difficult for an adversary to identify the actual number of real records returned by a query. Moreover, after each query we update a random number of dummy records, thus varying the size of the result set of consecutive queries. Increasing the number of dummy records will make it harder for an adversary to identify the actual real records; this however will incur an increase in processing time.

Furthermore, to achieve forward and backward privacy, we use nonces in the encrypted records and queries. After each search operation, the records are re-randomised using fresh nonces. Only queries that include the current nonce will be able to match the records. In this way, even if a malicious CSP tries to use previously executed queries with old nonces, she will not be able to match the records in the data set, ensuring forward privacy. Similarly, deleted records (with old nonces) will not match newly issued queries because they use different nonces. The use of nonces also limits the records that a revoked user could be able to access even if she is able to access the encrypted data.

Finally, to achieve sub-linear search complexity and minimise the communication overhead, we use an indexing technique. Technically, we divide the data into groups and require one of the CSPs to keep track of all the data locations for each record in the group. The search is then performed only on the records within a group rather than across the whole dataset.

The details and algorithms of our scheme will be discussed in the following section.

IV. Notation

In this work, a database $DB = \{rcd_1, rcd_2, \ldots\}$ is a set of records. Each record $rcd = (D_1, \ldots, D_M)$ has M fields. Each element D_m for $1 \leq m \leq M$ could be any allowed value, such as an integer, string or date. Data elements are collected into groups, which are used for the indexing. There is a function G that returns the group of a data element as $gid_m = G(D_m)$. Hence, associated to each record is an M-tuple $Grcd = (gid_1, \ldots, gid_M)$ that represents the group information for each data item in the record.

EDB is the encrypted DB stored on the SSS, consisting of a set of encrypted records, *i.e.,* $EDB = \{Ercd_1, Ercd_2, \ldots\}$ where $Ercd = (SE(D_1), \ldots, SE(D_M), tag)$ represents an encrypted record (here $SE(D_m)$ is the encrypted element and tag is used to mark if the record is dummy or real). The implementation of SE and tag is given in the next section.

The INS has a nonce database $NDB = \{Nrcd_1, Nrcd_2, \ldots\}$, storing nonces related to EDB. Each nonce record $Nrcd = (n_1, \ldots, n_{M+1}, flag)$ is a list of nonces, where the nonce n_m is used to encrypt D_m and is included in $SE(D_m)$. The nonce n_{M+1} is used to hide the value of tag from both the SSS and INS, and $flag$ is used to mark a dummy records for the INS.

In DB, EDB and NDB, each record is identified by a unique id, which can be thought of as being the physical location of the record. Hence, we will use the notation $rcd \leftarrow DB(id)$, $Ercd \leftarrow EDB(id)$ and $Nrcd \leftarrow NDB(id)$ to indicate selecting the corresponding information from these databases.

The INS also stores a matching of groups to indices, in the array GDB. So for each $1 \leq m \leq M$ and gid the array GDB stores the list of all ids of records such that the m-th data element lies in group gid.

Let λ be the security parameter in our system. We define the set of binary strings of length λ as $\{0,1\}^\lambda$, the set of all

finite binary strings is denoted as $\{0,1\}^*$. Sampling uniformly random from a set X is denoted as $x \xleftarrow{\$} X$. Given two binary strings a and b, the concatenation of them is denoted as $a||b$, and the xor operation between them is denoted as $a \oplus b$. Given two sets of binary strings, *i.e.,* $S_1 = \{a_1, \cdots, a_I\}$ and $S_2 = \{b_1, \cdots, b_J\}$, assume $I \leq J$, we define the xor operation between them as $S_1 \oplus S_2 = \{a_1 \oplus b_1, \cdots, a_I \oplus b_I\}$.

We use a symmetric encryption scheme $Enc : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^*$, such as AES-ECB. This is sufficient for our searchable encryption functionality; we do not require any special features of the encryption scheme. Function G is defined as $G : \{0,1\}^* \times \{0,1\}^\lambda \rightarrow \{0,1\}^\iota$, where ι is the length of gid.

V. Solution details

The system is setup by the admin by generating the secret key k. k is shared among users and is used to protect the data (including queries and records) from both the SSS and the INS.

A. Insert Query

Algorithm 1 Insert(rcd)

1: User(rcd):
2: **for** $m = 1$ to M **do**
3: $n_m \xleftarrow{\$} \{0,1\}^{|D_m|}$, $SE(D_m) \leftarrow Enc_k(D_m) \oplus n_m$
4: $gid_m \leftarrow G(D_m)$
5: **if** rcd is a real record **then**
6: $tag \xleftarrow{\$} \{0,1\}^{2\lambda}$, $n_{M+1} \xleftarrow{\$} \{0,1\}^{2\lambda}$
7: $flag \leftarrow 1$
8: **else**
9: $S \xleftarrow{\$} \{0,1\}^\lambda$, $n_{M+1} \xleftarrow{\$} \{0,1\}^{2\lambda}$, $tag \leftarrow (H_k(S)||S) \oplus n_{M+1}$
10: $flag \xleftarrow{\$} \{0,1\}$
11: Send $Ercd = (SE(D_1), \ldots, SE(D_M), tag)$ to the SSS
12: Send $Nrcd = (n_1, \ldots, n_{M+1}, flag)$ and $Grcd = (gid_1, \ldots, gid_M)$ to the INS

13: $SSS(Ercd)$:
14: Determine the next available index id for the record
15: $EDB(id) \leftarrow Ercd$
16: Send id to the INS

17: $INS(Nrcd, Grcd, id)$:
18: $NDB(id) \leftarrow Nrcd$
19: **for** $m = 1$ to M **do**
20: Add id to $GDB(m, gid_m)$

In this section, we describe the steps involved in inserting a record in *P-McDb*. The actual details of each operation are provided in Algorithm 1.

To insert a (real or dummy) record rcd, first each data element D_m in rcd is encrypted using the symmetric encryption algorithm Enc, and then XORed with the corresponding nonce n_m (Line 3, Algorithm 1). The use of the nonce n_m makes $SE(D_m)$ semantically secure. The user does not have to remember the nonce; this is the job of the INS.

To improve the search efficiency, data items are associated with groups, via $gid_m = G(D_m)$. For each group, the INS maintains an index list il containing the record's id. $Grcd$ is sent to the INS to build il.

To protect the size pattern privacy, while inserting new records, the user also generates and inserts together with the real records a number of dummy records. Each data element in a dummy record can be either copied from the real records or picked from the value space randomly, and encrypted using SE, which ensures that the dummy records may match with queries and are indistinguishable from real ones. Consequently, the size pattern is concealed.

To mark if a record is dummy or real, a tag tag is generated as shown in Lines 6 and 9, Algorithm 1, where $H : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ is a keyed hash function. With the secret key k, the dummy records can be efficiently filtered out by users before decrypting the search result by checking if:

$$lhtag \stackrel{?}{=} H_k(rhtag), \text{ where } lhtag||rhtag \leftarrow tag \oplus n_{M+1}$$

To make the size of the search results different, even when the same query is executed twice, we reveal a subset of dummy records to the INS by associating a flag with each record (Line 7 and 10). When the INS sees $flag = 0$, it knows the record is dummy. However, $flag = 1$ does not mean a record is real. After each query, the INS randomly updates the records with $flag = 0$ (see Algorithm 3).

The number of dummy records must be controllable since it affects the security and performance of *P-McDb*. The higher the percentage of dummy records with respect to real ones, the harder for the SSS and the INS to infer the search pattern. However, this also implies that more records should be searched for each query. In *P-McDb*, we set two parameters, δ_1 and δ_2, to control the minimum and maximum proportion of dummy records, respectively. The values of these two parameters can be set according to the practical requirements for security and performance. Specifically, when a user inserts x records, y dummy records are generated such that, on average, $\delta_1 x \leq y \leq \delta_2 x$ (this is easily done using standard random variable sampling methods). The study by Cash *et al.* in [15] suggests that 1.6 is the minimum ratio between dummy and real records to resist against size pattern-based attacks. A thorough security analysis for identifying a right balance between real and dummy records for achieving a sustainable level of security and performance is part of our future work.

To summarise, for each record rcd, the user generates (1) an encrypted record $Ercd$ that is sent to the SSS to be inserted into the encrypted database EDB; (2) a set of nonces $Nrcd$ that is sent to the INS and inserted into the database called NDB; (3) and the group information $Grcd$ is also sent to the INS that inserts it into another database named GDB.

Table I provides an example of EDB, GDB and NDB for a *Staff* table shown in cleartext in Table I(a). In our example, the *Staff* table has two fields, *Name* and *Age*, and contains two records.

EDB (shown in Table I(b)) is stored on the SSS and contains the encrypted records. In addition to the database fields (*i.e., Name* and *Age*), EDB also contains the tag field for tag. Note that EDB contains two extra records with id 3 and id 4: they are two dummy records inserted by the user. The INS knows the last record is dummy, since its $flag$ is set to be 0 in NDB.

Table I(c) represents GDB stored on the INS. It contains unique group ids (gid) and the list of records ids belonging

TABLE I

(A) A SAMPLE *staff* TABLE. (B) THE ENCRYPTED *staff* TABLE STORED ON THE SSS. EACH ENCRYPTED DATA ELEMENT $SE(D_m) = Enc_k(D_m) \oplus n_m$. EACH RECORD HAS A TAG, ENABLING USERS TO DISTINGUISH BETWEEN DUMMY AND REAL RECORDS. (C) INDEX MAINTAINED BY THE INS. THE *names* AND *ages* ARE DIVIDED INTO GROUPS ACCORDING TO THE FIRST LETTER AND RANGE, RESPECTIVELY. PRECISELY, $gid_{20} = G(25)$, $gid_{30} = G(32)$, $gid_a = G(Alice)$, $gid_b = G(Bob)$. (D) NONCES ARE STORED ON THE INS. FOR EACH DATA ELEMENT D_m, THE INS STORES A NONCE n_m THAT IS BOUND TO $SE(D_m)$. MOREOVER, FOR EACH RECORD, IT STORES ONE MORE NONCE FOR THE tag, AND A 1-BIT FLAG TO MARK IF IT IS DUMMY.

(a) *Staff*

id	Name	Age
1	Alice	25
2	Bob	32

(b) EDB on the SSS

id	$SE(name)$	$SE(age)$	tag
1	$SE(Alice)$	$SE(25)$	tag_1
2	$SE(Bob)$	$SE(32)$	tag_2
3	$SE(Bob)$	$SE(25)$	tag_3
4	$SE(Alice)$	$SE(25)$	tag_4

(c) GDB on the INS

gid	Index list
gid_{20}	$\{1, 3, 4\}$
gid_{30}	$\{2\}$
gid_a	$\{1, 4\}$
gid_b	$\{2, 3\}$

(d) NDB on the INS

id	n_1	n_2	n_3	$flag$
1	n_{11}	n_{21}	n_{31}	1
2	n_{12}	n_{22}	n_{32}	1
3	n_{13}	n_{23}	n_{33}	1
4	n_{14}	n_{24}	n_{34}	0

to a given group. For instance, in our example, the group with gid^a contains the id of all the records where the values of the field $Name$ starting with the letter A. The INS also stores in NDB the nonce set $Nrcd$ associated with each record stored in EDB. Table I(d) represents NDB for our toy example.

Finally, the field names in Tables I(b) and I(d) (as well as in queries) are encrypted as $SE(name) = Enc_k(name)$.

B. Select Query

For simplicity, we only explain our solution for queries whose predicate is a single equality statement $D_m = C_m$. For example, 'select * from staff where name=Alice'. From this building block, it is possible to support complex queries and range queries; for details, see the full version of the paper [24]. EQ represents the encrypted query. The group is $gid_m = G(C_m)$.

Algorithm 2 Search(m, C_m)

1: User(m, C_m):
2: **for** $i = 1$ to M **do**
3: $\quad \eta_i \xleftarrow{\$} \{0,1\}^{|D_i|}$
4: Set $\mathcal{N} = (\eta_1, \ldots, \eta_{M+1})$ where $\eta_{M+1} \xleftarrow{\$} \{0,1\}^{2\lambda}$
5: $SE(C_m) \leftarrow Enc_k(C_m) \oplus \eta_m$
6: $gid_m \leftarrow G(C_m)$
7: Send $EQ = SE(C_m)$ to the SSS
8: Send the nonce set $\mathcal{N}$ and gid_m to the INS

9: $INS(m, gid_m, \mathcal{N})$:
10: $EN \leftarrow \emptyset$
11: $il_m \leftarrow GDB(m, gid_m)$
12: $IL \leftarrow il_m \cup$ a set of ids selected from other groups randomly
13: **for** each $id \in IL$ **do**
14: $\quad EN(id) \leftarrow NDB(id) \oplus \mathcal{N}$
15: Send the encrypted nonce set EN and the index list IL to the SSS

16: $SSS(m, EQ, IL, EN)$:
17: **for** $i = 1$ to M **do**
18: $\quad n'_i \xleftarrow{\$} \{0,1\}^{|D_i|}$
19: Set $\mathcal{N}' = (n'_1, \ldots, n'_{M+1})$ where $n'_{M+1} \leftarrow \{0,1\}^{2\lambda}$
20: $SR \leftarrow \emptyset$, $Trcds \leftarrow \emptyset$
21: **for** each $id \in IL$ **do**
22: $\quad Trcd \leftarrow EDB(id) \oplus EN(id)$
23: $\quad$ **if** $Trcd(m) = EQ$ **then**
24: $\quad\quad SR \leftarrow SR \cup Trcd$
25: $\quad Trcds(id) \leftarrow Trcd \oplus \mathcal{N}'$
26: Send the search result SR to the user
27: Send the transformed record set $Trcds$ to the INS for shuffling

For performing a select query, *P-McDb* requires the cooperation between the INS and the SSS. The details of the steps performed by the user, the INS and the SSS are shown in Algorithm 2.

The user first encrypts the query '$D_m = C_m$' using k and a nonce set $\mathcal{N}$ to get EQ. The nonce ensures that the encrypted query EQ is semantically secure. Also, the user generates group information gid_m for C_m. Finally, the user sends the encrypted query EQ to the SSS, and the group information set GQ together with the nonce set $\mathcal{N}$ to the INS.

The INS determines the search range from GQ (Line 11, Algorithm 2). In addition to the queried group, a set of random ids from other groups are selected (the number of additional ids is another parameter). The SSS will end up searching over all these indices, so it is hard to know exactly which indices correspond to the group. For each record in the search range IL, the INS creates an encrypted nonce vector by retrieving the nonces from NDB (using the record id in IL) and XORing them with the nonces $\mathcal{N}$ generated by the user (Line 14). The set EN encrypted nonces is then sent to the SSS together with the search range IL.

For each search operation, the SSS creates a set $\mathcal{N}'$ with fresh nonces (this is needed to hide the data from the INS during the shuffling operation). It also creates two empty sets: SR that will contain search results and $Trcds$ that will contain transformed records. The SSS performs the search operation within the ids contained in IL. For each id in IL, the SSS XORs the record in EDB with the corresponding element in EN. This operation is necessary because the nonce in the encrypted query is different from the nonces in the records stored in EDB. Therefore, it would not be possible for the SSS to match the encrypted values in the query with the encrypted values of the fields in the records. By XORing the encrypted record with the corresponding element in EN, we basically XOR the records with the same nonce as the query without revealing its content to the SSS. In details, recalling that the elements in EN are $n_m \oplus \eta_m$, the operation in Line 22 is:

$$Trcd(m) \leftarrow Enc_k(D_m) \oplus \eta_m = (Enc_k(D_m) \oplus n_m) \oplus (n_m \oplus \eta_m)$$

where η_m represents the nonce in the query. Similarly, the tag is XORed as: $tag \oplus (n_{M+1} \oplus \eta_{M+1})$.

At this stage, the SSS can perform the matching operation. In case of a match, the SSS adds $Trcd$ to SR (Line 24).

Only the user issuing the query knows $\mathcal{N}$ and is able to decrypt the records in SR by computing $Enc_k^{-1}(Trcd \oplus \mathcal{N})$, where Enc^{-1} is the inverse of Enc. Similarly, only the user

issuing the query will be able to distinguish between real and dummy records because she is the only one who knows η_{M+1}.

All the records listed in IL are touched by the SSS during the search (even if they do not match the query). To protect the access pattern, all these records in IL need to be shuffled and re-encrypted. The shuffling and re-encryption are performed by the INS as explained in Section V-C. Each $Trcd$ in $Trcds$ is now encrypted under k and the nonce set $\mathcal{N}$. If the INS and the user collude then they could retrieve all the records in $Trcds$, which could contain more records than SR. To avoid this, all the records in $Trcd$ are XORed with the new set of nonces $\mathcal{N}'$ that is known only to the SSS (Line 25).

C. Shuffling and Re-randomisation

Algorithm 3 Shuffle($Trcds$)

1: $INS(Trcds, IL)$
2: **for** each $id \in IL$ **do**
3: **for** $i = 1$ to M **do**
4: $r_i \xleftarrow{\$} \{0,1\}^{|D_i|}$
5: Set $R = (r_1, \ldots, r_{M+1})$ where $r_{M+1} \leftarrow \{0,1\}^{2\lambda}$
6: **if** $NDB(id).flag = 0$ **then**
7: **for** each $SE(D_m) \in Trcds(id)$ **do**
8: $SE(D'_m) \xleftarrow{\$} Trcds$, $SE(D_m) \leftarrow SE(D'_m) \oplus r_m$
9: $tag \leftarrow tag \oplus r_{M+1}$
10: **else**
11: $Trcds(id) \leftarrow Trcds(id) \oplus R$
12: $NDB(id) \leftarrow \mathcal{N} \oplus R$
13: Shuffle $Trcds$ and update GDB and NDB
14: Send $Trcds$ to the SSS.
15:
16: $SSS(Trcds, IL)$
17: **for** each record $id \in IL$ **do**
18: $EDB(id) \leftarrow Trcds(id) \oplus \mathcal{N}'$

In most of the existing searchable schemes, the ciphertext and location of the data are not changed, unless there is a delete or update query. Consequently, if the same search results are returned, an adversary can infer that the two queries are logically equivalent, thus leaking the access pattern. To protect the access pattern, after executing every query, we shuffle the search results with other records and re-randomise all of them. In this way, having the same ids in two search results does not mean the corresponding queries are equivalent.

Ideally, one could shuffle all the records in the database. However, this would have a significant impact on performance. In this work, we only shuffle all the records in IL, representing the set of records touched by the SSS.

The INS is responsible for shuffling and re-randomisation of the records in $Trcds$ as described in Algorithm 3. Note that, at this stage, the user has already obtained the search results from the SSS and does not need to wait for this operation to be completed.

For each record in $Trcds$, which contains all the records in IL, the INS first generates a new set of nonces R. To make sure that the size of the matching records is different for every query (*e.g.,* size pattern privacy), the INS updates the subset of dummy records, where $flag = 0$. Specifically, for each data element $SE(D_m)$ in the dummy record, the INS updates it with another value $SE(D'_m)$ that is picked from $Trcds$ randomly, and then re-randomises it with a nonce (Line 6-9, Algorithm 3). For the record with $flag = 1$, it is just re-randomised with the nonce set R (Line 11). Formally, the final re-randomised data element is:

$$SE(D_m) \leftarrow Enc_k(D_m) \oplus \eta_m \oplus r_m$$

while the tag is:

$$tag \leftarrow tag \oplus \eta_{M+1} \oplus r_{M+1}$$

The new value $Nrcd$ corresponding to the re-randomised record is:

$$Nrcd = (\eta_1 \oplus r_1, \ldots, \eta_{M+1} \oplus r_{M+1})$$

and it will be stored in NDB. Finally, the list $Trcds$ is permuted randomly, or in other words shuffled. Because of the shuffling and updating operations, the group information should be updated in GDB (Line 13). The shuffled and re-randomised records $Trcds$ are then sent to the SSS. Before the records are stored in EDB, the nonce set $\mathcal{N}'$ is removed from each record by XORing it again (Line 18, Algorithm 3). Finally, the shuffled data is re-written by the SSS into the same memory locations specified by IL. The point is that the SSS knows that these and only these indices have been changed, but it does not know the permutation that was applied to the data.

By using a new set of nonces R during the shuffling, we are able to achieve both forward and backward privacy. If the SSS tries to execute an old query, it will not be able to match any records without the new nonce set, which is known only to the INS. Similarly, the SSS cannot learn if deleted records match new queries.

Due to page limit, the security analysis of ***P-McDb*** is not provided in this paper. For a security analysis of ***P-McDb***, an interested reader is referred to the extended version [24].

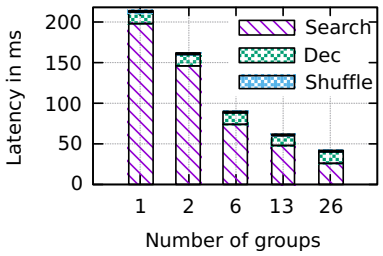
D. Delete and Update Queries

On the user and the INS, delete queries are processed in the same way as select queries. On the SSS, all the matched records are deleted as usual. Meanwhile, the SSS sends the ids of the matched records and informs the INS to delete the corresponding nonces. The remaining records in IL will be shuffled and re-randomised by the INS. Update queries are executed by deleting the stale data first and inserting the new value later.

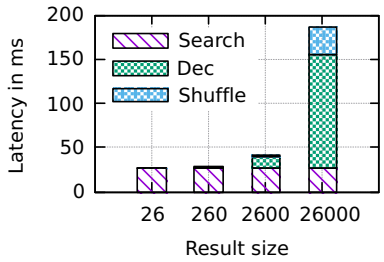
E. User Revocation

Because of the nonces bound to SE, without the assistance of the INS and the SSS, the revoked user is unable to recover the query and records only with k. Therefore, for user revocation, we just need to manage a revoked user list at the INS as well as at the SSS. Once a user is revoked, the admin informs the INS and the SSS to add this user into their revoked user lists. When receiving a query, the INS and the SSS will first check if the user has been revoked. If yes, they will reject the query. In case a revoked user colludes with either the SSS or INS, she cannot get the search results, since such operation requires the cooperation of both the user issuing the query, the INS and the SSS.

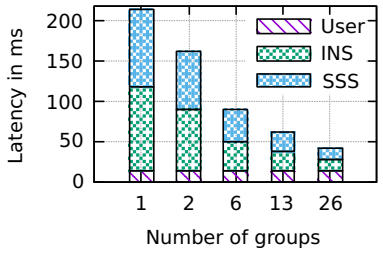
VI. Performance Analysis



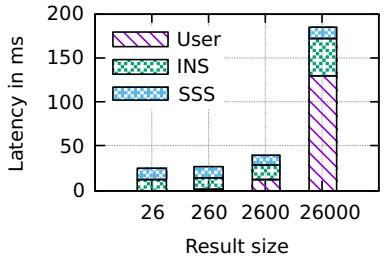
(a) Latency of each phase with different number of groups. The result size is 2600 (1000 real records).



(b) Latency of each phase with different result sizes and 26 groups.



(c) Latency at each entity with different number of groups. The result size is 2600 (1000 real records).



(d) Latency at each entity with different result sizes and 26 groups.

Fig. 2. Query time for a single keyword search with 2.6 million records, where 1 million are real records.

We implemented *P-McDb* in C consisting of 1000 lines of code using MIRACL 7.0 library for cryptographic primitives. The performance of all the entities was evaluated on a desktop machine running Intel i5 3.3 GHz 4-core processor with 8GB of RAM. In the experiments, we ignored network latency that could occur in a real deployment. We created a *Staff* table with five fields, *name*, *age*, *gender*, *position* and *email*. The database contains 2.6 million records of which 1.6 million are dummy records. The ratio between real records and dummy records is same as the suggested by Cash *et al.* in [15]. In the following, all the data points in the graphs were averaged over 10 trials.

In the following experiments, we distinguish three phases when a query is executed: *Search*, *Dec* and *Shuffle*. The Search phase includes all the operations shown in Algorithm 1. In the Dec phase, the user removes the dummy records from the results set and decrypts the real ones. In the Shuffle phase, instead of shuffling all the records in IL as described in Algorithm 3, the INS shuffles and re-randomises the same number of searched records together with the result set.

Figure 2(a) shows the effect on each phase when we change the number of groups and fix the size of the search result. As we can see, the latency is mostly spent on the Search phase. We observe that increasing the number of groups reduces the search time as the SSS has fewer records to search through. Due to the efficient XOR operation, even without grouping the data, the user can get the cleartext search result in 200 milliseconds (ms). The Dec and Shuffle phases instead are not affected by the group size but by the search results. This is more clearly shown in the following experiment.

Figure 2(b) shows how the latency varies when we fix the group size and use different sizes for search results. As we can see, the Search latency remains constant, but the time of the Dec and Shuffle phases increases with the increase in the result size. Similarly, the shuffling time is affected significantly by the result size. The worst case is to shuffle a single group or the whole database, where the latter ensures a higher security level. When the result size is 26000 (10000 real records), the shuffle time becomes the main overhead. It should be noted that, the Dec phase on the user can be started without waiting for the completion of the Shuffle phase.

An important aspect of an outsourced service is that most of the intensive computations should be off-loaded to the CSPs. To quantify the workload on each of the entities, we have measured the latency on the user, the INS and the SSS for processing a result set of 2600 records with different group sizes. The results are shown in Figure 2(c). We can notice that for a fixed result set of 2600 records, the latency on the user is always less than 10 ms. The latency on the user side is mainly for decrypting the returned values. This is clearly shown in Figure 2(d) where we keep the group size fixed and change the size of the returned set. As we can see, the latency on the user increases with the increase in the result set size. It should be noted however, that for a 20K result set the user requires around 120 ms to filter out dummy records and decrypt all the real ones. The increase in the result set size also affects the latency on the INS because a larger result set means more records to shuffle.

More tests reporting the performance comparison with other works, the performance of more complex queries and the latency to shuffle all the touched records can be found in the extended version of this paper [24].

VII. Discussion

In *P-McDb*, if the SSS colludes with the INS, they could learn the search and access patterns by removing the nonces included in records and queries. Furthermore, if a user colludes with both the INS and the SSS, they can decrypt all the records. To resist such collusion attacks, there are two possible solutions. The simplest one would be to introduce a proxy server between the users and the CSPs. The proxy server would blind the records and queries with its own set of nonces. As long as the proxy server is deployed on a trusted private cloud managed by the data owner, the CSPs could not decrypt the data even if they have the key k. The other approach would be to generalise the system by introducing n CSPs, assuming not all of them collude together. In this case, each data element is blinded with n nonces. Only when n CSPs collude together, the search and access patterns will be leaked, and only when the user collude with n CSPs, all the data can be recovered.

Considering field names in queries and tables are encrypted using a deterministic algorithm, *P-McDb* leaks if the queries are searched over the same fields or not. This can be protected by also encrypting field names using nonces and storing the nonces on the INS. It is also necessary to shuffle the database over columns and re-randomise encrypted field names to make them untraceable on the INS after executing each query.

For each query, the INS has to send all the nonces indexed by IL to the SSS. When IL is large, the communication overhead will be computationally intensive. This is because each record in *P-McDb* is bound with a unique nonce record $Nrcd$. Although the encryption function Enc is deterministic, considering statistical information is already protected by dummy records, the records can be bound with same nonces. The INS could use the same nonce set to re-randomise the records in the same group during the shuffle operation. In this way, only one encrypted nonce set $ENrcd$ should be sent to the SSS.

VIII. Conclusion

In this work, we presented *P-McDb*, a dynamic searchable encryption scheme for multi-cloud outsourced databases. *P-McDb* supports sub-linear search and does not leak information about search, access and size patterns. It also achieves both forward and backward privacy, where the CSPs cannot reuse cached queries for checking if new records have been inserted or if records have been deleted. Furthermore, *P-McDb* offers a flexible key management scheme where revoking users does not require regeneration of keys and re-encryption of the data. As future work, we plan to investigate some performance optimisations to achieve more efficiency without sacrificing security guarantees offered by *P-McDb*, and do our performance analysis by deploying the scheme in the real multi-cloud setting.

References

[1] M. R. Asghar, G. Russello, B. Crispo, and M. Ion, "Supporting complex queries and access policies for multi-user encrypted databases," in *CCSW 2013* (A. Juels and B. Parno, eds.), pp. 77–88, ACM, 2013.

[2] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 222–233, 2014.

[3] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *NDSS 2014*, The Internet Society, 2014.

[4] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *CCS 2006* (A. Juels, R. N. Wright, and S. D. C. di Vimercati, eds.), pp. 79–88, ACM, 2006.

[5] M. Naveed, M. Prabhakaran, and C. A. Gunter, "Dynamic searchable encryption via blind storage," in *SP 2014*, pp. 639–654, IEEE Computer Society, 2014.

[6] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *SOSP 2011* (T. Wobber and P. Druschel, eds.), pp. 85–100, ACM, 2011.

[7] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage.," in *NDSS 2013*, vol. 71, pp. 72–75, 2014.

[8] B. Wang, W. Song, W. Lou, and Y. T. Hou, "Inverted index based multi-keyword public-key searchable encryption with strong privacy guarantee," in *INFOCOM 2015*, pp. 2092–2100, IEEE, 2015.

[9] W. Sun, S. Yu, W. Lou, Y. T. Hou, and H. Li, "Protecting your right: Attribute-based keyword search with fine-grained owner-enforced search authorization in the cloud," in *INFOCOM 2014*, pp. 226–234, IEEE, 2014.

[10] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *S&P 2000*, pp. 44–55, IEEE Computer Society, 2000.

[11] P. Rizomiliotis and S. Gritzalis, "ORAM based forward privacy preserving dynamic searchable symmetric encryption schemes," in *CCSW 2015* (I. Ray, X. Wang, K. Ren, F. Kerschbaum, and C. Nita-Rotaru, eds.), pp. 65–76, ACM, 2015.

[12] M. I. Sarfraz, M. Nabeel, J. Cao, and E. Bertino, "Dbmask: Fine-grained access control on encrypted relational databases," in *CODASPY 2015* (J. Park and A. C. Squicciarini, eds.), pp. 1–11, ACM, 2015.

[13] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *SIGSAC 2016* (E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, eds.), pp. 1143–1154, ACM, 2016.

[14] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *ACNS 2005* (J. Ioannidis, A. D. Keromytis, and M. Yung, eds.), vol. 3531 of *Lecture Notes in Computer Science*, pp. 442–455, 2005.

[15] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *SIGSAC 2015* (I. Ray, N. Li, and C. Kruegel, eds.), pp. 668–679, ACM, 2015.

[16] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *SIGSAC 2015* (I. Ray, N. Li, and C. Kruegel, eds.), pp. 644–655, ACM, 2015.

[17] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *USENIX Security 2016*, pp. 707–720, USENIX Association, 2016.

[18] C. Liu, L. Zhu, M. Wang, and Y. Tan, "Search pattern leakage in searchable encryption: Attacks and new construction," *Inf. Sci.*, vol. 265, pp. 176–188, 2014.

[19] S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Outsourced symmetric private information retrieval," in *SIGSAC 2013* (A. Sadeghi, V. D. Gligor, and M. Yung, eds.), pp. 875–888, ACM, 2013.

[20] I. Hang, F. Kerschbaum, and E. Damiani, "ENKI: access control for encrypted query processing," in *SIGMOD 2015* (T. K. Sellis, S. B. Davidson, and Z. G. Ives, eds.), pp. 183–196, ACM, 2015.

[21] L. Ferretti, F. Pierazzi, M. Colajanni, and M. Marchetti, "Scalable architecture for multi-user encrypted SQL operations on cloud database services," *IEEE Trans. Cloud Computing*, vol. 2, no. 4, pp. 448–458, 2014.

[22] S. Cui, M. R. Asghar, S. Galbraith, and G. Russello, "Secure and practical searchable encryption: A position paper," in *ACISP 2017*, Lecture Notes in Computer Science, Springer, 2017.

[23] "RightScale2016." Last accessed: June 14, 2016.

[24] S. Cui, M. R. Asghar, S. D. Galbraith, and G. Russello, "P-McDb: Privacy-preserving search in multi-cloud encrypted databases," 2017. https://www.cs.auckland.ac.nz/~asghar/papers/eprint-P-McDb.pdf.