ObliviousDB: Practical and Efficient Searchable Encryption with Controllable Leakage

Shujie Cui, Muhammad Rizwan Asghar, Steven D. Galbraith and Giovanni Russello

The University of Auckland, New Zealand

Abstract. Searchable encryption allows users to execute encrypted queries over encrypted databases. Several encryption schemes have been proposed in the literature but they leak sensitive information that could lead to inference attacks. We propose *ObliviousDB*, a searchable encryption scheme for an outsourced database that limits information leakage. Moreover, our scheme allows users to execute SQL-like queries on encrypted data and efficiently supports multi-user access without requiring key sharing. We have implemented *ObliviousDB* and show its practical efficiency.

1 Introduction

Cloud computing is a successful paradigm offering companies virtually unlimited data storage and computational power at very attractive costs. Despite its benefits, cloud computing raises new challenges for protecting data.

Motivation. Once the data is outsourced to the cloud environment, the data owner lacks a valid mechanism for protecting the data from unauthorised access. This poses serious confidentiality and privacy concerns to the outsourced data. To mitigate this problem, the hybrid cloud computing approach is getting more popular among large enterprises [1,2]. In a hybrid cloud approach, the organisation maintains sensitive data and services within their infrastructure and outsources the rest to a public cloud. However, identifying sensitive assets is not an easy task and once the data and services leave the internal infrastructure, there is a risk of compromising the confidentiality of the assets if no proper security mechanisms have been put in place.

Problem. In recent years, Searchable Encryption (SE) schemes have been proposed to partially overcome the confidentiality issue in cloud computing. These schemes allow the cloud to perform encrypted search operations on encrypted data. Most of them focus on improving the search efficiency and functionality. A thorough survey with a comparative analysis of existing SE schemes can be found in our recent work [3].

Unfortunately, researchers have paid little attention to the information the cloud provider can learn during search and match operations even if performed on encrypted data. Some recent works [4–6] have shown that even a minor leakage can be exploited to learn sensitive information and break the scheme.

In [5], Naveed *et al.* recover a vast majority of data in CryptDB [7] by using frequency analysis. Zhang *et al.* [6] further investigate the consequences of leakage by injecting chosen files or records into the encrypted database. Based on the information

learned by looking at which encrypted data is accessed by a given query, referred to as *access pattern* leakage, they could recover a very high fraction of the searched keywords by injecting a small number of known files or records into the database. The cloud provider can also infer if two or more queries are equivalent or not, referred to as *search pattern* leakage. A recent study by Cash *et al.* [4] also shows that given small leakage a determined attacker (including a malicious cloud provider) could break the encryption scheme.

Matters are even worse for *dynamic* SE schemes where insert and delete operations are also supported. Most of the dynamic SE schemes do not support *forward privacy* and *backward privacy* properties. Lacking forward privacy means that the cloud provider can learn if newly inserted data or updated data matches previously executed queries; lack of support for backward privacy means that the cloud provider can learn if deleted data matches new queries. Basically without forward and backward privacy, a cloud provider executing a dynamic SE scheme is able to learn the evolution of the data over time. Only a few of the existing dynamic schemes [8–10] support forward privacy, but no scheme is able to support both properties simultaneously.

A possible solution could be to employ Oblivious Random Access Memory (O-RAM) or Private Information Retrieval (PIR) schemes. However, current ORAM and PIR schemes are prohibitively costly and impractical.

Our Solution. In this paper, we present *ObliviousDB*, an SE scheme for databases for hybrid cloud environments, that is able to overcome all the issues discussed above.

Based on proxy-encryption given in [11], *ObliviousDB* is an encrypted search scheme that supports the full-fledged multiple user management. Moreover, *ObliviousDB* exploits the hybrid cloud computing approach and minimises information leakage. In our approach, the organisation is not required to make decisions on how to split its data between the private and public infrastructure. The public infrastructure is used for storing all the data while the private infrastructure is used mainly for running our *Oblivious Proxy Service (OPS)*, a proxy service for maintaining metadata information about the data stored in the public infrastructure.

The OPS plays a major role in ensuring the confidentiality of the data and manages the data structures for achieving search efficiency. In terms of its functionality, the OPS is similar to the proxy server used in CryptDB [7]. However, unlike CryptDB, we have designed the OPS to be robust against attacks *i.e.,* a compromised OPS will not reveal sensitive data to adversaries.

Contributions. This paper makes the following novel contributions:

1. *ObliviousDB* minimises the information leaked to the cloud provider when executing queries by (i) dynamically re-randomising the encrypted data, (ii) shuffling the locations of records within the database, and (iii) introducing and varying a random number of dummy records, necessary for achieving *search and access pattern privacy*.
2. To achieve *operation pattern privacy*, where the cloud server is not able to distinguish between select, insert, delete and update queries, the OPS obfuscates the actual operations executed by the users by inserting additional queries and combining queries into the shuffle operation.

3. *ObliviousDB* supports both forward and backward privacy by randomising data and query through the use of fresh nonces. In this way, even if the cloud provider stores a search query, it cannot be matched with new data. Likewise, new queries cannot be executed over deleted records.

To show the feasibility of our approach, we have implemented *ObliviousDB* and measured its performance.

2 Overview of *ObliviousDB*

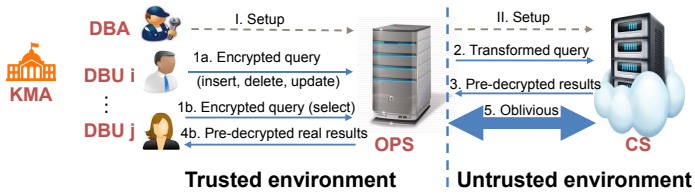


Fig. 1. Overview of *ObliviousDB*: A DBA is responsible for running setup (Step I then Step II). A DBU can insert, delete and update the data (Step 1a) or execute a select query (Step 1b) to receive matching records (Step 4b). Regardless of the query type, to control information disclosure, the OPS transforms the query (Step 2) to perform the search (Step 3) followed by an oblivious algorithm (Step 5).

In the remainder of the paper, we set the context and informally describe the properties used in our categorisation. **Search Pattern Privacy (SPP)** refers to the property where the cloud provider is not able to distinguish if two (or more) queries are the same or not. **Access Pattern Privacy (APP)** means the cloud provider is unable to infer if two (or more) result sets contain the same records or not. **Size Pattern Privacy (SzPP)** is achieved if the cloud provider is unable to learn the size of returned (real) records. **Operation Pattern Privacy (OPP)** is achieved if the cloud provider is unable to discover if the issued query is select, insert, delete or update. **Forward Privacy** means the cloud provider does not learn if a new or updated record matches a query executed in the past. **Backward Privacy** means the cloud provider is unable to executed queries on records that have been deleted or modified.

2.1 System Model

The system involves five main entities shown in Figure 1:

- **Database Administrator (DBA)**: A DBA is responsible for management of the database, its users and access control policies for regulating access to tables.
- **Database User (DBU)**: It represents an authorised user who can execute select, insert, update and delete queries over encrypted data. After executing encrypted queries, a DBU can retrieve the result set, if any, and decrypt it.

- **Oblivious Proxy Server (OPS)**: It provides greater security and search efficiency. It serves as a proxy between DBUs and the cloud server. In order to hide sensitive information about queries, it pre-processes the queries submitted by the DBU. It also filters out dummy records from the result set returned to the DBU. To improve performance, it manages some indexing information. Technically, the OPS is part of the private cloud in the hybrid cloud environment, which is linked with a more powerful public cloud infrastructure.
- **Key Management Authority (KMA)**: This entity is responsible for generating keying material once a new DBU joins the system. Furthermore, the KMA removes the DBU, when she is compromised or no longer part of the database.
- **Cloud Server (CS)**: A CS is part of the public cloud infrastructure provided by a cloud service provider. It stores the encrypted data and access control policies and enforces those policies to regulate access to the data.

Threat Model. We assume that the KMA is fully trusted. The KMA does not need to be online all the times. In particular, it has to be online only when the system is initialised, a new DBU is created or an existing one is removed from the system. In this way, the organisation can easily secure the KMA from external attacks. DBUs are only considered to keep their keys (and decrypted data) securely.

We consider that the CS is honest-but-curious. More specifically, the CS would honestly perform the operations requested by the DBA and DBUs according to the designated protocol specification; however, it is curious to analyse the stored and exchanged data so as to learn additional information. We assume that the CS will not mount active attacks, such as modifying the message flow or denying access to the database.

The OPS is deployed in the private cloud, which is owned by the organisation. Hence, we assume the OPS is trusted. However, it is responsible for communicating with the external world. Thus, it could be the target of attackers and get compromised, which means the data stored on the OPS could possibly be exposed to attackers.

In this work, we assume that there are mechanisms in place for data integrity and availability. Last but not least, access policy specification is out of the scope of this paper, but the approach introduced in [11, 12] can be utilised in *ObliviousDB*.

3 Solution Details

3.1 Data Representation

Table 1 illustrates an example of how we represent and store the data on the OPS and CS. Let us assume that we have a table Staff (Table 1(a)) containing *Name* and *Age* fields. The CS stores an encrypted version of this, which is *EDB* and illustrated in Table 1(c), where each data element is encrypted under Data Encryption (DE) and Searchable Encryption (SE), where DE ensures the confidentiality of the retrievable data, and SE makes the data searchable (the implementation details are given in Algorithm 1). Similarly, we encrypt each value in the table.

To improve search efficiency and reduce the communication overhead, we support indexing. Technically, we divide the data into groups and build an index for each group

Table 1. Data representation on each entity.

(a) *Staff*

Name	**Age**
Alice	25
Anna	30
Bob	27

(b) GDB

GID	**Nonce**	**Index List**
g_1	n_{20}	$\{1,3,4\}$
g_2	n_{30}	$\{2\}$
g_3	n_a	$\{1,2,4\}$
g_4	n_b	$\{3\}$

(c) EDB

ID	$\{\textbf{Name}\}_{SE}$	$\{\textbf{Name}\}_{DE}$	$\{\textbf{Age}\}_{SE}$	$\{\textbf{Age}\}_{DE}$
1	$SE_{n_a}(Alice)$	$DE(Alice)$	$SE_{n_{20}}(25)$	$DE(25)$
2	$SE_{n_a}(Anna)$	$DE(Anna)$	$SE_{n_{30}}(30)$	$DE(30)$
3	$SE_{n_b}(Bob)$	$DE(Bob)$	$SE_{n_{20}}(27)$	$DE(27)$
4	$SE_{n_a}(Alice)$	$DE(xyz)$	$SE_{n_{20}}(25)$	$DE(13)$

Table (a) is a sample table viewed by DBUs. Table (b) is the group information stored on the OPS. We have $g_1 = GE(25) = GE(27)$, $g_2 = GE(30)$, $g_3 = GE(Alice) = GE(Anna)$ and $g_4 = GE(Bob)$. The group ID is encrypted, since the OPS could be compromised. Each group has a nonce to ensure forward and backward privacy and a list of IDs indicating the records in the group. The CS stores Table (c), where each value is encrypted with *SE* and *DE* for data search and retrieval, respectively. Each *SE* value is bound with the nonce of its group. The last record, consisting of normal *SE* and fake *DE* parts, is dummy.

maintained by the OPS. When a query is received, the OPS sends to the CS the corresponding list of indices to be searched. Table 1(b), called *GDB*, shows an example of the group information. Note that each field of the database corresponds to a different group, so that in a complex query the OPS would identify all the groups corresponding to fields in the query and send to the CS the union or intersection of the indices (depending on whether the query is a disjunction or conjunction). The group identifiers are concealed with *GE* (the detail is given in Algorithm 1, Section 3.3). The secret key for *GE* is only known to DBUs. This means that if the OPS gets compromised then an attacker is unable to learn the actual data items that correspond to a group.

SE and DE representations do not leak information about encrypted values. However, the CS can easily learn the number of matching records during the search process. In *ObliviousDB*, the OPS adds dummy records. Note that the CS is not able to distinguish between a dummy and a real record. To make sure that the dummy records match with the queries generated by DBUs, the OPS generates dummy records such that the SE fields correspond to real data values. Specifically, the OPS generates dummy records by sampling SE terms from the set of real records in the corresponding group. The OPS also maintains a list of indices that contain dummy records (not shown in Table 1(b)); this could be an N-bit string $flags$, where N is the total size of the database on the CS. We have $flags[id] = 0$ or 1 if the record is a dummy or real record, respectively.

To ensure that dummy records are not delivered to the DBU, the OPS filters them out from the search result. A large number of dummy records can ensure a high level of privacy but at the cost of poor performance since the CS has to search over more records and the OPS has to filter out more dummy records before returning the result to the DBU. For controlling dummy records, the DBA sets a threshold t at the initialisation time, which is the ratio between dummy and real records. In reality, the value of t can be set according to the practical requirements for security and performance, and depending on the type of data being stored in the database. For example, in a different context, Cash *et al.* [4] suggested taking $t = 0.6$ to resist against size pattern based attacks on the Enron emails dataset.

ObliviousDB achieves both forward and backward privacy. That is, even if the CS holds old queries, they cannot be matched with new records. Similarly, if the CS holds deleted records, they cannot be matched with new queries. To achieve both properties, the OPS re-encrypts each record and query with nonces. Nonces are generated and

maintained on groups, meaning all the records with the same group are under the same nonce. When a query is executed over a given group, the OPS will generate a new nonce and the data will be updated accordingly. The nonces are not revealed to the CS. The OPS maintains the information between groups and nonces using the *GDB* table, as shown in Table 1(b).

3.2 Setup

The system is set up by the KMA by taking as input a security parameter λ. The output is a prime number p, three multiplicative cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of order p, such that there is a "Type 3' bilinear map [13] $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, which has the properties of *bilinearity*, *computability* and *non-degeneracy*, but there is no symmetric bilinear map defined on $\mathbb{G}_1$ alone or $\mathbb{G}_2$ alone. Let g_1 and g_2 be the generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. The KMA chooses a random x from $\mathbb{Z}_p$ and returns $h = g_1^x$. Next, it chooses a collision-resistant keyed hash function H, a pseudorandom functions f and a random key s for f. It also initialises the key store managed by the CS. That is, $K_S \leftarrow \phi$. Finally, it publishes the public parameters $Params = (e, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, h, H, f)$ and keeps securely the master secret key $MSK = (x, s)$.

Building on top of proxy encryption [11, 14], *ObliviousDB* supports multi-user access with efficient DBU registration and revocation. Specifically, when the DBU i joining the system, the KMA splits MSK into two values x_{i1} and x_{i2}, where $x = x_{i1} + x_{i2}$ mod p and $x_{i1}, x_{i2} \in \mathbb{Z}_p$. Then, the KMA transmits $K_{U_i} = (x_{i1}, s)$ and $K_{S_i} = (i, x_{i2})$ securely to the DBU i and the CS, respectively. The CS adds K_{S_i} to its key store: $K_S \leftarrow K_S \cup K_{S_i}$. With K_{U_i}, DBU i could issue a query. For revoking a DBU, we just need to remove K_{S_i} on the CS.

3.3 Query Execution

ObliviousDB supports SQL-like queries consisting of a set of equalities and inequalities, which are connected with conjunctions (*i.e.,* and) and disjunctions (*i.e.,* or), such as 'select * from staff where name=Alice and age$>$ 30'. To support range queries, we use the same approach presented in [11].

Every query executed in *ObliviousDB* is performed with the cooperation of the DBU, the OPS and the CS. The details of the steps performed by each entity are described in Algorithm 1.

The DBU encrypts the query with key K_{U_i} (Lines 1-11, Algorithm 1). For an insert or update query, each data element d is encrypted under GE, SE and DE. For a select query, a keyword k in the WHERE-clause is encrypted only under GE and SE. The encrypted query EQ and group information GQ are sent to the OPS. Both DE and SE are semantically secure because of the random numbers r, which prevents the CS to infer the search pattern from EQ, or learn any frequency information in EDB.

On the OPS, the original query EQ is cached temporarily (Line 14), and the real insert, delete and update operations will be performed by the oblivious algorithm later. To hide the operation pattern, the OPS always sends an encrypted select query to the CS. So, the OPS first transforms EQ into a select query (Lines 15-17). Specifically, if EQ is update or delete query, the OPS just changes the terms 'delete' and 'update' into

Algorithm 1 $Query(Q)$

1: $\underline{DBU_i(Q)}$:
2: **for** each data element d in Query Q **do**
3: $\sigma \leftarrow f_s(d)$
4: $GE(d) \leftarrow LSB_k(\sigma)$ {the least significant k bits of σ}
5: $r \leftarrow Z_p^*, SE(d) \leftarrow (c_1 = g_1^r, c_2 = g_1^{\sigma r})$
6: $r \leftarrow Z_p^*, DE(d) \leftarrow (e_1 = g_2^r, e_2 = h^r d)$
7: **for** each keyword k in WHERE-clause of Q **do**
8: $\sigma \leftarrow f_s(k)$
9: $GE(k) \leftarrow LSB_k(\sigma)$
10: $r \leftarrow Z_p^*, SE(k) \leftarrow (t_1 = g_2^r, t_2 = g_2^{\sigma r})$
11: Send the encrypted query $EQ = (SE(Q), DE(Q))$ and its group information $GQ = GE(Q)$ to the OPS
12: $\underline{OPS(EQ, GQ)}$:
13: $IL \leftarrow \emptyset$
14: Cache a copy of EQ
15: **if** EQ is an insert query **then**
16: Generate a fake select query as EQ
17: **if** EQ is a delete or update query **then**
18: Change the type of EQ into *select*
19: **for** each $GE(k) \in GQ$ and $SE(k) \in EQ$ **do**
20: $(n, il) \leftarrow GDB(GE(k))$
21: $IL \leftarrow IL \star il$, where $\star$ is the conjunction in GQ
22: $SE_n(k) \leftarrow (t_1 \leftarrow t_1^n = g_2^{rn}, t_2)$
23: Send $(IL, SE_n(Q), i)$ to the CS, where i is the identifier of the DBU
24: $\underline{CS(IL, SE_n(Q), i)}$:
25: $SR \leftarrow \emptyset$
26: **for** each $id \in IL$ **do**
27: **if** $Match(EDB(id), SE_n(Q))$=true **then**
28: Add all the required $DE(d)$ in $EDB(id)$ into SR
29: **for** each $DE(d) = (e_1 = g_1^r, e_2 = h^r d) in SR$ **do**
30: $DE'(d) \leftarrow (e_1, e_2' = e_2 * e_1^{-x_2} = g_1^{x-x_2} d = g_1^{x_1 r} d)$, where x_2 is the CS side key for DBU_i
31: Send SR to the OPS
32: $\underline{OPS(SR)}$:
33: Remove all dummy records from SR by checking flags
34: Send SR to the DBU
35: $\underline{DBU_i(SR)}$:
36: **for** each $DE'(d) = (e_1 = g_1^r, e_2' = g_1^{x_1 r} d) \in SR$ **do**
37: $d \leftarrow e_2' * e_1^{-x_1} = g_1^{x_1 r} d * g_1^{-x_1 r}$

Algorithm 2 $Match(rcd, SE_n(Q))$

1: **for** each $SE_n(k) = (t_1, t_2) \in SE_n(Q)$ **do**
2: Get $SE_n(d)$ in the same field from rcd
3: **if** $e(c_1, t_2) \neq (c_2, t_1)$ **and** $* =$ 'and' **then**
4: **return** false
5: **if** $e(c_1 t_2) = e(c_2, t_1)$ **and** (*='or') **or** k is the last keyword **then**
6: **return** true
7: **return** false

'select *'. If EQ is insert, a random number of values in $SE(Q)$ are used to assemble the WHERE-clause of the fake select query.

Second, to improve search efficiency, the OPS gets the search range IL for the C-S, which is populated by merging the index lists of involved groups according to the conjunctions and disjunctions in GQ (Lines 19-21). Meanwhile, to ensure forward and backward privacy, each SE in EDB is bound to the nonce of this group (Line 22). Only the query bound with the same nonce could match the record. Both the index list IL and the transformed query $SE_n(Q)$ are sent to the CS.

Finally, the CS checks each record in IL with $SE_n(Q)$ by performing the pairing map operation (Lines 26-28). The match operation is described in detail in Algorithm 2. Assume the searched data element is $SE_n(d) = (c_1 = g_1^{r'n'}, c_2 = g_1^{\sigma' r'})$ and the encrypted keyword in $SE_n(Q)$ is $SE_n(k) = (t_1 = g_2^{rn}, t_2 = g_2^{\sigma r})$. The equality check between them is performed by checking whether $e(c_1, t_2) = e(c_2, t_1)$. Note that

$$e(c_1, t_2) = e(c_2, t_1) \iff e(g_1^{r'n'}, g_2^{r\sigma}) = e(g_1^{r'\sigma'}, g_2^{rn})$$
$$\iff e(g_1, g_2)^{r'n'r\sigma} = e(g_1, g_2)^{rnr'\sigma'}$$

and so if $\sigma = \sigma'$ and $n = n'$ then equality holds, while inequality holds with negligible probability if $\sigma \neq \sigma'$ and $n \neq n'$ That is, the record matches the query only when $k = d$

and they are bound to the same nonce. Finally, the search result *SR* is sent to the OPS. Yang *et al.* introduce a similar method to perform the equality check for SE schemes in [15]. However, their method leaks the search pattern and frequency information of records to the CS, since the pairing map they use is symmetric. That is, the CS could still infer if they are the same or not by running the bilinear map operation between two records or two queries, although they are encrypted with a probabilistic algorithm.

The search result is recovered in two rounds of decryption (Lines 29-37, Algorithm 1). Before sending *SR* to the OPS, the CS first pre-decrypts the DE parts of each matched record with DBU_i's CS side key x_2 (Lines 29-30), and sends the pre-decrypted *SR* to the OPS. Next, the OPS filters the dummy records out (Lines 33-35). Finally, with the pre-decrypted real search result get from the OPS, the DBU can recover the plaintext with its DBU side key x_1 (Lines 36-37).

Algorithm 3 $Oblivious(EQ, GQ, t)$

1: $Rcds \leftarrow \emptyset$
2: **for** each $GE(k) \in GQ$ **do**
3: $(n, il) \leftarrow GDB(GE(k))$
4: $rcds \leftarrow EDB(il)$ {Get from CS all the records indexed by il}
5: $n \leftarrow nn'$, where $n' \xleftarrow{\$} Z_p^*$
6: **for** each record $rcd \in rcds$ **do**
7: $SE_n(d) = (c_1 \leftarrow c_1^{n'}, c_2)$ {d is in the same field as k}
8: **for** each $(SE_n(d), DE(d))$ pair in rcd **do**
9: $r \xleftarrow{\$} Z_p, SE_n(d) = (c_1 \leftarrow c_1^r, c_2 \leftarrow c_2^r)$
10: $r \xleftarrow{\$} Z_p, DE(d) = (e_1 \leftarrow e_1^r, e_2 \leftarrow e_2^r)$
11: **for** each dummy record $rcd \in rcds$ **do**
12: **if** $du/re > t$ **then**
13: delete it, $du--$
14: **else**
15: $SE(d') = (c_1, c_2) \xleftarrow{\$} rcds$
16: $SE_n(d) \leftarrow (c_1^{rn}, c_2^r)$, $r \xleftarrow{\$} Z_p^*$ {d is in the same field as k}
17: $Rcds \leftarrow Rcds \cup rcds$
18: **for** each matched real record $rcd \in Rcds$ **do**
19: **if** EQ is an update query **then**
20: **for** each $SE(d) \in SE(Q)$ **do**
21: $SE_n(d) \leftarrow (c_1^n, c_2), n \leftarrow GDB(GE(d))$
22: Update rcd with $SE_n(Q)$ and $DE(Q)$
23: **if** EQ is a delete query **then**
24: Invert its flag
25: $du++, re--$
26: **if** EQ is an insert query **then**
27: Assign an id to the new record
28: **for** each $SE(d) \in SE(Q)$ **do**
29: **if** $\emptyset \leftarrow GDB(GE(d))$ **then**
30: $n \xleftarrow{\$} Z_p^*$
31: $GDB(GE(d)) \leftarrow (n, id)$
32: **else**
33: $(n, il) \leftarrow GDB(GE(d))$
34: $il \leftarrow il \cup id$
35: $SE_n(d) \leftarrow (c_1 \leftarrow c_1^n, c_2)$
36: $Rcds \leftarrow Rcds \cup (SE_n(Q), DE(Q))$
37: $re++$
38: $flags[id] = 1$
39: **else**
40: Assign the id to a new dummy record
41: **for** each field in rcd **do**
42: $SE(d') = (c_1, c_2) \xleftarrow{\$} Rcds$
43: $r \xleftarrow{\$} Z_p$, $n \leftarrow GDB(g)$, $SE_n(d) \leftarrow (c_1^{nr}, c_2^r)$
44: $e_1, e_2 \xleftarrow{\$} G_1, DE(d) \leftarrow (e_1, e_2)$
45: $il' \leftarrow GDB(GE(d'))$
46: $il' \leftarrow il' \cup id$
47: $Rcds \leftarrow Rcds \cup rcd$
48: $du++$
49: $flags[id] = 0$
50: Shuffle $Rcds$ and upload to CS
51: Update the index list in GDB and update $flags$

3.4 Oblivious Algorithm

To hide the access, size and operation patterns and ensure forward and backward privacy, in the oblivious algorithm, the OPS shuffles and re-randomises all the records included in the searched the groups every time a query is executed.

To ensure a high level of security, it is possible to shuffle all the records in the database. However, this degrades the system performance. The more records are shuf-

fled, the more difficult it is for the CSPs to infer the access pattern, but it is worse in terms of the system performance. In this work, we shuffle all the records in the searched groups. In this case, the CS can only recognise if two queries are performed within the same group or not. In practice, the number of records to be shuffled can be set according to the performance and security requirements. Note that, at this stage, the user has already obtained the search results from the CS and does not need to wait for the shuffle operation to be completed.

There are four main steps in the oblivious algorithm. In the first step (Lines 3-10, Algorithm 3), for each group involved in the query, the OPS updates its nonce and re-encrypts the associated SE parts of the records in this group with the new nonce (Line 7). Consequently, the queries bound to previous nonces cannot match the re-encrypted records, as illustrated in (3.3). Similarly, a new query bound to the latest nonce can not match stale records. That is, both forward and backward privacy are ensured. Meanwhile, the OPS re-randomises both the SE and DE parts of all the records to be shuffled to make them untraceable (Lines 8-10).

In the second step (Lines 11-16), all the dummy records in each searched group are updated. The OPS controls the number of dummy records to ensure the performance of the system (Lines 12-13). To this end, the OPS counts the total number of real records re in the index list IL and the total number of dummy records du in IL. When the ratio of dummy records exceeds the threshold t, some of them are deleted. The OPS updates the SE parts of the remained dummy records (Lines 14-16). Although the dummy record protects the real size pattern, if the fake size pattern for the same query never changes, the CS could make some assumptions on whether two queries are equivalent or not by checking the number of the records in the result set. To protect the search pattern, it is necessary to make the size pattern for all queries variable. Note that considering the correctness of the system it is impossible to change the number of matched real records when there is no insert, delete or update operation. In our work, the OPS replaces the SE parts of dummy records with randomly chosen ones from non-dummy records in the group (lines 15-16).

Recall that to protect the operation pattern the OPS only sends select queries to the CS. The real insert, update and delete operations are executed in the third step (Lines 18-48). If the type of the original query is update (Lines 19-22), the OPS re-encrypts the cached SE parts with the latest nonce stored in GDB, and replaces both the SE and DE parts of the matched records with new values. If the type of the original query is delete (Lines 23-25), in order to protect the operation pattern, the OPS converts them into dummy records by inverting the flag into dummy instead of deleting them directly. In this way, no matter what type the original query is, the number of records returned by the oblivious algorithm is only affected by de, re and t. On the contrary, if we delete them, fewer records will be sent to the CS. For other types of queries, a number of dummy records are probably deleted. However, the CS could learn the type of the original query must not be delete when it gets more records or the same number of records after the oblivious algorithm. If the type of the original query is insert (Lines 26-38), the OPS re-encrypts the cached SE parts with the latest nonces and adds it to the records set. In case if the original query is not insert, the OPS generates a dummy record (Lines 39-49) by re-randomising the SE parts that picked from $Rcds$ randomly. Otherwise, the

CS could infer the type of the original query must be insert when receiving one more record after the oblivious algorithm.

Finally, the OPS shuffles all the updated records set and sends them to the CS (Line 50). Note that their flags and *id*s stored in the index list are updated at the same time. Because of the shuffling and re-randomising, if the same query is executed again, the search result will be totally different from the previous ones in terms of the store locations on the CS, size and appearance, which means the CS is unable to infer if different search results contain the same records or not. That is, the access pattern is protected.

4 Security Analysis

In this section, we formally define and prove SPP, APP, SzPP, OPP, forward and backward.

Leakage *ObliviousDB* aims at minimising information leakage. At the same time, we require *ObliviousDB* to be efficient. To meet these two conflicting requirements, we consider a trade-off between security and performance. In this work, we achieve SPP and APP for those queries involved in the same groups. To optimise the performance of the system, we divide the data into groups and only perform the search and oblivious operations within groups. Consequently, the CS could learn if some records and interested keywords are in the same groups or not. Second, considering we do not encrypt the conjunctions between predicates, the CS could learn the number of predicates and their conjunctions in complex queries. In addition, the CS could learn if two queries are searched over the same fields or not, since we do not re-randomise the field names and shuffle the columns. Given these leakages, we formalise our security definition below.

Definition In our security definition, we only consider the queries with the same structure. Any two queries Q_0 and Q_1 have the 'same structure' if they satisfy the following criteria:

- Both SQL queries have the same logical structure (all WHERE-clauses are the same equalities and inequalities with respect to the same data fields, and the sentences are formed from the clauses using the same conjunctions or disjunctions in the same order). This can be achieved by padding and reordering the WHERE clause. Note that the queries could be insert, select, update or delete.
- The groups involved by each equality/inequality in Q_0 should be same to the one involved in the corresponding equality/inequality in Q_1. This will ensure that both index lists are the same, *i.e.,* $IL_0 = IL_1$.
- The number of matched real records $|RR|$ for each equality/inequality in Q_0 and Q_1 should be similar in size, namely $||RR_0| - |RR_1|| \leq \theta * \min\{|RR_0|, |RR_1|\}$, where θ is a parameter specified when the scheme is set up.

The CS is modelled as the Probabilistic Polynomial-Time (PPT) adversary $\mathscr{A}$, which means $\mathscr{A}$ honestly follows the protocols and gets all the messages the CS sees.

The scheme is considered to be secure if an adversary could break it with not more than a negligible probability. Formally, it could be defined as follows:

Definition 1 ***(Negligible Function)****. A function f is negligible if for every polynomial $p(.)$ there exits an N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.*

Definition 2 *Let $\prod$ =(Setup, Query, Oblivious) be ObliviousDB, λ be the security parameter, and t be the threshold indicating the ratio between dummy and real records. $\mathscr{A}$ is a PPT adversary, and $\mathscr{C}$ is a challenger. The game between $\mathscr{A}$ and $\mathscr{C}$ in $\prod$ is described as below:*

- ***Setup*** *The challenger $\mathscr{C}$ first initialises the system by generating Params and MSK. Then, she generates the secret key pair (K_U, K_S). The adversary $\mathscr{A}$ is given Params and K_S.*
- ***Bootstrap*** *$\mathscr{A}$ submits a database Δ[1]. Assume Δ contains n records with a certain number of fields. $\mathscr{C}$ encrypts Δ and divides the data in each field into groups. Moreover, $\mathscr{C}$ generates a number of dummy records for each group, such that the total number of dummy records is $t \cdot n$. The encrypted database EDB is sent to $\mathscr{A}$. The encrypted groups information GDB is securely kept by $\mathscr{C}$.*
- ***Phase 1*** *$\mathscr{A}$ can make polynomially many SQL queries Q in plaintext. All the queries are in the same structure but could be of different types. $\mathscr{C}$ encrypts and transforms each query Q to $SE_n(Q)$, and generates the index list IL, as would be done by the DBU and the OPS. With $SE_n(Q)$ and IL, $\mathscr{A}$ searches over EDB to get the search result SR. After that, $\mathscr{C}$ and $\mathscr{A}$ engage in the oblivious algorithm to update EDB. So, for each query, $\mathscr{A}$ sees $SE_n(Q)$, IL, SR and the records set Rcds returned by the oblivious algorithm. Note that, the $\mathscr{A}$ could cache $SE_n(Q)$ and execute it again independently at any time.*
- ***Challenge*** *$\mathscr{A}$ sends two queries Q_0 and Q_1 to $\mathscr{C}$ that have the same structure, which can be those already issued in phase 1. Note that if Q_0 or Q_1 is insert or update query, the data elements included in them should already exist in Δ. $\mathscr{C}$ responds the request as follows: it chooses a random bit $b \in \{0,1\}$ and transforms query Q_b, as done by the DBU and OPS, to $SE_n(Q)$ and IL. Then, $\mathscr{C}$ and $\mathscr{A}$ perform the full protocol, so that $\mathscr{A}$ learns SR and Rcds.*
- ***Phase 2*** *$\mathscr{A}$ continues to adaptively request polynomially many queries, which could include the challenged queries Q_0 and Q_1.*
- ***Guess*** *$\mathscr{A}$ submits her guess b'.*

The advantage of $\mathscr{A}$ in this game is defined as:

$$Adv_{\mathscr{A},\prod}(1^\lambda) = Pr[b' = b] - \frac{1}{2}.$$

We say ObliviousDB achieves SPP, APP, SzPP, OPP, forward and backward privacy, if all PPT adversaries have negligible advantage in the above game.

In this game, $\mathscr{A}$ is very powerful. She knows the plaintext of all the real records and queries, could arbitrarily generate and issue any kind of queries as long as they are in the same structure, and has the full access to *EDB*. That is, she could learn the real search result of all the issued queries, and could adaptively run the encrypted queries over *EDB* to get the encrypted search results at any time. If one of the search, access, size, operation patterns and forward and backward privacy is not protected, $\mathscr{A}$ could

[1]For simplicity, we assume there is only a single table in Δ and regard Δ as a table. Without loss of generality, our proofs will hold for a database containing a set of tables.

infer b easily. For example, if the search pattern is not protected (*i.e.,* $\mathscr{A}$ can learn if the terms involved in two queries are the same or not), she could select one of the queries issued in phrase 1 as either Q_0 or Q_1 and win the game by checking if $SE_n(Q)$ is same to one of those get in phase 1; if the size pattern is not protected (*i.e.,* $\mathscr{A}$ can learn the number of real records in SR), she could win the game by setting Q_0 and Q_1 with different numbers of matched records; if the operation pattern is not protected (*i.e.,* $\mathscr{A}$ can learn the type of Q_b), she could set two different types of query as Q_0 and Q_1 and win the game from the type of $SE_n(Q)$. In other words, if with these abilities, $\mathscr{A}$ still can not win the game with non-negligible advantage, it means *ObliviousDB* achieves all the properties.

Theorem 1. *Let the SE and DE schemes have semantic security. Let t (the proportion of dummy records) be chosen sufficiently large relative to θ. If the SE and DE schemes have semantic security, ObliviousDB achieves SPP, APP, OPP, SzPP, forward and backward privacy.*

Proof. (Sketch) We show that the bit b chosen by $\mathscr{C}$ is information-theoretically hidden from the view of $\mathscr{A}$, assuming that both SE and DE are semantically secure.

Consider the view of $\mathscr{A}$ in the game. $\mathscr{A}$ chooses an arbitrary database Δ and uploads this to $\mathscr{C}$. In Phase 1, $\mathscr{A}$ makes queries that are answered correctly by $\mathscr{C}$ by following the protocols.

In the challenge round, $\mathscr{A}$ sends two queries Q_0 and Q_1. $\mathscr{A}$ receives a list of $SE_n(k)$ terms corresponding to the literals in the predicate defining the query. By definition, the two queries have the same structure. Hence, the same number of literals, each of the same type, will be received by $\mathscr{A}$ for either query. Since SE is semantically secure, $\mathscr{A}$ cannot distinguish the query terms given the ciphertexts $SE_n(k)$.

$\mathscr{A}$ also receives a list IL of database indexes to be searched by the CS, and by definition, this is the same list for all queries. Hence, no information about the queries can be leaked by IL.

Each group involved in IL is accompanied by a nonce n. With overwhelming probability, these nonces are distinct and unrelated to the values used in previous queries. Previously encrypted search keywords can no longer be used to query these indexes, and $SE_n(Q)$ can not be executed over stale records, since the nonces do not match. Hence, there is no way to link information from previous search queries to these records, indicating forward and backward privacy is achieved.

The adversary $\mathscr{A}$ may try to guess b from the search result SR. Although the numbers of real records matched with Q_0 and Q_1 are known to $\mathscr{A}$ since all the queries and real records in plaintext are set by her, a number of dummy records are inserted into EDB in order to hide the number of real records that are matched. Since SE and DE are semantically secure, the dummy records are indistinguishable from the real ones, if t is sufficiently large compared to θ then the probability distributions of the result set sizes $|SR_0|$ and $|SR_1|$ are statistically close and $\mathscr{A}$ cannot distinguish them from a single query. Therefore, the CS is unable to distinguish the two queries from the size of SR, indicating SzPP is achieved.

Even if the queries Q_0 and/or Q_1 have previously been executed by $\mathscr{A}$, the refreshing of dummy records, together with the shuffling and re-randomising performed in the

oblivious algorithm, imply that $\mathscr{A}$ cannot distinguish the two queries by comparing *SR* with previous search results, indicating APP is achieved.

Finally, $\mathscr{A}$ gets *Rcds* from the oblivious algorithm. Some records in *Rcds* may be updated with new values or turned into dummy records, and one of them is newly added. Due to the semantic security of *SE* and *DE*, *Rcds* leaks nothing to $\mathscr{A}$, indicating the operation pattern is concealed, *i.e.,* OPP is achieved.

The game continues in Phase 2. $\mathscr{A}$ may repeat Q_0 and/or Q_1. If Q_0 and Q_1 are different types of queries, *e.g.,* insert and delete. $\mathscr{A}$ may run a related select query to test the search result. Again, due to the refreshing of dummy records, the shuffling and re-randomising operations, the number, the ciphertext and store locations of matched records for Q_0 and/or Q_1 will be different from *SR*. Similarly, the nonce updating does not allow records to be linked to records found in previous search queries. Hence, the future state of the database and the queries in Phase 2 are independent of the query made in the challenge round.

Since $\mathscr{A}$ has no information to distinguish the bit b, the scheme satisfies the definition. ■

5 Performance Analysis

We implemented the scheme in C using the MIRACL 7.0.0 library, necessary for cryptographic primitives. The implementation of the overall system including the functions on the DBU, the OPS and the CS was tested on a single machine with 64 Intel *i*5 3.3 GHz processor and 8 GB RAM running Ubuntu 14.08 Linux system. In our testing scenario, we ignored network latency that could occur in a real deployment. In the following, all the results are averaged over 10 trials.

The tested database contains one table with 3 fields. Considering the search operation can be performed in each field, we encrypted each field with SE separately. However, each record was encrypted by DE as a whole since we only tested 'select *' queries. In the following, we tested how the three controllable parameters namely the number of dummy records, the number of groups and the number of shuffled records, affect the performance of the system.

We first present the results of end-to-end latency measured at the DBU when performing a search operation on a database consisting of $100,000$ real records with a result set of $1,000$ real records. Note that, in the DBU latency experiment, we did not measure the time the OPS spends in executing the oblivious algorithm. The reason is that the OPS will forward to the DBU the result sets before initiating the oblivious algorithm.

The graphs in Figure 2 illustrate latency in seconds. In particular, Figure 2(a) shows the results for a simple select query. Figure 2(b) reports latency for performing a range query on a numerical field. In both graphs, the X-axis shows the number of groups: that is, we change the granularity of the indexing going from no indexing (where all the records are part of one group) to a more fine-grained indexing. For a given number of groups, the same experiment was executed 5 times, each time changing the ratio t, represented by different lines in both graphs.

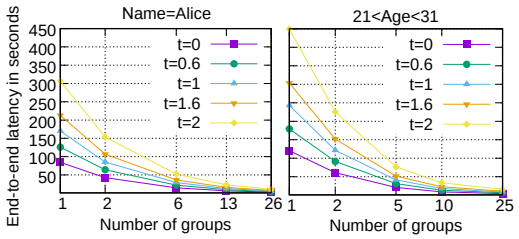


Fig. 2. End-to-end latency on the DBU for getting 1,000 real records from the database consisting of 100,000 real records. The database size goes up to 300,000 with the increase of t, the ratio between dummy and real records.

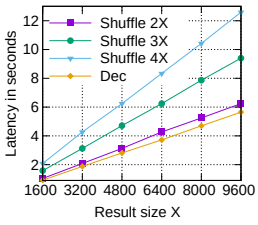


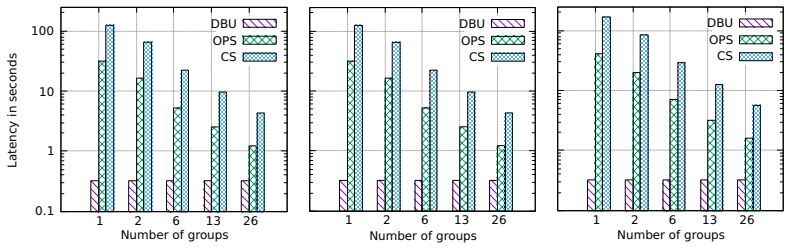
Fig. 3. Oblivious latency with $t = 0.6$. *Shuffle 2X, 3X, 4X* mean the number of records to be shuffled are 2, 3, 4 times of the result size, respectively.

As we expected, for both queries, increasing the number of groups reduces the DBU latency. For a given size of a database, more groups mean fewer records within a given group. This reduces searching time on the CS and in turn reduces latency on the DBU.

On the other hand, increasing the value of t degrades the performance. For both queries, for a given group size, there is a slight latency increase when we go from $t = 0$ dummy records (*i.e.,* only real records) to a ratio of $t = 2$ (*i.e.,* 2 times dummy records of real ones). This is explained mainly by two facts: a) with more dummy records, the CS has to retrieve more records (both real and dummy ones), and b) the OPS needs to filter out the dummy records before sending the real records to the DBU: the higher the percentage of dummy records, the longer it takes for the OPS to remove them from the result set (to be returned to the DBU). Recall that $t = 0.6$ is the minimum value to ensure security reported by Cash *et al.*.

Second, we measured the effect on the oblivious algorithm when varying the number of records to be shuffled. As shown in Figure 3, the latency of the oblivious algorithm goes up linearly with the increase in result size, which affects the size of shuffled records in the test setting. Fortunately, the oblivious algorithm can be executed in parallel with decryption operations since they are independently executed by different entities. From Figure 3, we can observe that, if we shuffle the search result with the unmatched records (where the number of unmatched records is same as the number of records in the search result), the latency of the oblivious algorithm is close to the decryption time, indicating the oblivious algorithm does not severely degrade the throughput of *ObliviousDB*.

Next, we want to provide some details on the time each entity, namely the DBU, the OPS and the CS, spends for executing a query. Figure 4 shows the graphs for the execution of the select query (same as the one for the graph in Figure 2(a)). In this experiment, for each graph, we shuffled all the records in the searched group and kept the ratio constant while we changed only the number of groups (shown on the X-axis). As we can see, the more groups we introduce, the better performance we achieve on the CS (while for the DBU and the OPS, there is no big variation). Increasing the ratio between dummy and real records slightly increases latency on the OPS and the CS.



(a) t=0 (b) t=0.6 (c) t=1

Fig. 4. Latency on the DBU, the OPS and the CS for executing 'select $*$ from *staff* where name=Alice' with three different ratios of dummy records.

From our experiments, we can see that latency, although substantially higher than a less secure scheme like CryptDB, for DBUs using *ObliviousDB* is still usable especially when introducing more groups. Also, by comparing Figures 2(a) and 2(b), we can see that the performance of numerical range queries is not much different from simpler single keyword queries.

At the same time, to ensure data confidentiality, it is necessary to maintain some dummy records in the data set. However, our experiments show that the burden of maintaining dummy records does not impact latency on the DBU, in particular when a large number of groups are used. The cost of maintaining the dummy records is offloaded to the OPS and the CS, which are likely to be deployed on more powerful machines than the one used by the DBU.

6 Conclusions and Future Work

In this work, we propose *ObliviousDB*, a searchable scheme for hybrid outsourced databases. ObliviousDB is the first full-fledged multi-user scheme that does not leak information about search pattern, access pattern, size pattern and operation pattern. It is also the first scheme that achieves both forward and backward privacy, where the CS cannot reuse cached queries for checking if new records have been inserted or if records have been deleted. We have implemented *ObliviousDB* and shown that it is capable of performing numerical range queries with 1000 results on a database of 200,000 records in around 4 seconds.

As future work, we plan to carry out a thorough security analysis for identifying a right balance between real and dummy records for achieving a sustainable level of security without degrading performance. Another area we want to explore is to investigate sub-linear data structure to achieve more efficiency.

References

1. "Gartner expects five years for hybrid cloud to reach productivity," last accessed: February 19, 2016. [Online]. Available: http://www.cloudcomputing-news.net/news/2015/aug/18/gartner-expects-hybrid-cloud-reach-productivity-five-years-are-they-right/

2. "Rightscale 2016 state of the cloud report," last accessed: July 3, 2016. [Online]. Available: https://www.rightscale.com/lp/state-of-the-cloud
3. S. Cui, M. R. Asghar, S. D. Galbraith, and G. Russello, "Secure and practical searchable encryption: A position paper," in *ACISP 2017, Part I*, ser. Lecture Notes in Computer Science, J. Pieprzyk and S. Suriadi, Eds., vol. 10342. Springer, 2017, pp. 266–281.
4. 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. ACM, 2015, pp. 668–679.
5. 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. ACM, 2015, pp. 644–655.
6. 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*. USENIX Association, 2016, pp. 707–720.
7. 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. ACM, 2011, pp. 85–100.
8. E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage." in *NDSS 2013*, vol. 71, 2014, pp. 72–75.
9. 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. ACM, 2016, pp. 1143–1154.
10. Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *ACNS 2005*, ser. Lecture Notes in Computer Science, J. Ioannidis, A. D. Keromytis, and M. Yung, Eds., vol. 3531, 2005, pp. 442–455.
11. 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. ACM, 2013, pp. 77–88.
12. M. R. Asghar, "Privacy preserving enforcement of sensitive policies in outsourced and distributed environments," Ph.D. dissertation, University of Trento, Trento, Italy, December 2013, http://eprints-phd.biblio.unitn.it/1124/.
13. S. D. Galbraith, K. G. Paterson, and N. P. Smart, "Pairings for cryptographers," *Discrete Applied Mathematics*, vol. 156, no. 16, pp. 3113–3121, 2008.
14. C. Dong, G. Russello, and N. Dulay, "Shared and searchable encrypted data for untrusted servers," in *DBSec 2008*, ser. Lecture Notes in Computer Science, V. Atluri, Ed., vol. 5094. Springer, 2008, pp. 127–143.
15. G. Yang, C. H. Tan, Q. Huang, and D. S. Wong, "Probabilistic public key encryption with equality test," in *CT-RSA 2010*, ser. Lecture Notes in Computer Science, J. Pieprzyk, Ed., vol. 5985. Springer, 2010, pp. 119–131.