ObliviousDB: Practical and Efficient Searchable Encryption with Controllable Leakage

Shujie Cui, Muhammad Rizwan Asghar, and Giovanni Russello

The University of Auckland, New Zealand

Abstract. Searchable encryption scheme allows a client to carry out keyword search on encrypted data. However, almost all the schemes proposed in the literature so far either fail to avoid the leakage of extra information which leads to the scheme vulnerable to inference attacks or are inefficient to support multi-user read and write operations independently.
We propose an sublinear searchable encryption scheme that achieves the least leakage so far and efficiently supports multi-user access. In particular, our scheme could resist against inference attack and allows multiple users to carry out complex SQL-like queries in sublinear search time on the encrypted data without sharing any secret key or re-encrypting operation. Finally, the construction is implemented and showed 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 ensuring data confidentiality. Once the data is outsourced to the cloud environment, the data owner lacks a valid mechanism for protecting the data from unauthorised accesses. This poses serious confidentiality and privacy concerns on data being stored in the cloud. To mitigate this problem, the hybrid cloud computing approach is getting more popular among large enterprises [1]. In a hybrid cloud approach, the organisation maintains sensitive data and services within their infrastructure and outsource the rest to a public cloud. However, the issue of identifying sensitive assets is not an easy task and once the data and services leaves the internal infrastructure, there is no turning back, in particular if not properly protected.

In recent years, several searchable encryption schemes have been proposed to gradually reduce confidentiality barrier in cloud computing. These schemes allow the cloud to perform encrypted search operations on encrypted data. Unfortunately, most of the existing searchable schemes suffer from the issues listed below.

Information Leakage. During data search and updates, there is information that is leaked to the cloud server just by analysing encrypted data and operations performed by the authorised users. A recent study by Cash *et al.* [2] shows that information leakage can be exploited by a determined attacker to break the encryption scheme. As shown by Naveed *et al.* [3], using statistical analysis, it is possible to successfully attack the deterministic encryption scheme used in CryptDB [4] and recover nearly 100% of the queries within seconds.

To minimise such leakage, one might employ protection techniques based on Oblivious RAM (ORAM) [5–7] or Private Information Retrieval (PIR) [8, 9]. However, these schemes are very costly and/or can only be applied in static settings, meaning they do not scale well when dealing with dynamic data updates and delete operations.

Forward and Backward Privacy. Concerning the information leakage issues, most of the work in literature do not support *forward privacy* property: if a query q is performed and later a new record r is added (or updated) to match the query q, the cloud server does not learn that r matches any query q used in the past. On the contrary, *backward privacy* means the cloud server is unable to match queries over deleted documents. Supporting these properties is fundamental to limit the power of the cloud server to collect information on the data stored in the database and queries performed by the authorised users. From the existing schemes, only [10] is able to support forward privacy but no scheme is able to support both properties simultaneously.

Linear Search. To provide an efficient data retrieval, database engines use indexing techniques. However, supporting indexing of encrypted data is not trivial. Besides, using an index might leak extra information to the cloud server.

Lack of support for Fully-Fledged Multi User Scheme. In a Fully-Fledged Multi User (FFMU) scheme, any authorised user is able to read and write data from and to the database, respectively, without requiring any key sharing [11]. A FFMU scheme better supports the needs of modern organisations, where users need to access and update data. The vast majority of existing approaches supports either Single User (SU) or Semi-Fledged Multiple User (SFMU) schemes: in the former case, a single key is used for reading and writing data to the database; in the latter case, there is a single key for writing data and the rest of the users might have separate keys for read operations. Both of these options are impractical in modern organisations, where users should be able to join and leave their organisation or role at any time, potentially without impacting the rest of the users.

In this paper, we present *ObliviousDB*, an encrypted searchable database for the hybrid cloud environment that is able to overcome all the issues discussed above. *ObliviousDB* is based on our earlier work [11]. As such, it is an encrypted search scheme that supports the FFMU management. However, compared to our previous work, *ObliviousDB* has been extended to take advantage of the hybrid cloud computing approach. At the core of *ObliviousDB*, there is the *Oblivious Proxy Server (OPS)* that is deployed in the private infrastructure of an organisation. The OPS plays a major role in ensuring confidentiality of the data and manages the data structures for supporting sub-linear search operations. In terms of its functionality, the OPS is similar to the proxy server used in CryptDB [4]. Unlike CryptDB, we have designed the OPS to be robust against attacks: a compromised OPS will not reveal sensitive data to adversaries. This paper provides several fundamentally novel contributions listed as follows:

1. *ObliviousDB* does not leak information to the cloud server when executing operations. Through the use of the OPS, we can prevent the cloud server to recognise if two queries correspond to related search terms – achieving search and access pattern privacy – by dynamically shuffling the locations of records within the database and re-randomising the encrypted data. To achieve operation pattern privacy, where

the cloud server is not able to distinguish between select, delete and update queries, the OPS ensures that every query contains both read and write operations;
2. *ObliviousDB* supports both forward and backward privacy by using nonces in the searchable encryption so that a search query cannot be repeated at a later time or new queries cannot be executed over deleted records.
3. Our scheme achieves sub-linear search efficiency. For each query, the OPS generates an optimised query so that the cloud server only needs to search a group of records, instead of the whole database. Still, our indexing scheme will not reveal sensitive information to the cloud server.

To the best of our knowledge, we are the first to propose a scheme that solves all aforementioned issues while supporting the FFMU management. To show feasibility of our approach, we have implemented *ObliviousDB* and measured its performance.

The rest of this paper is organised as follows. In Section 2, we define some preliminaries. The related work is reviewed in Section 3. In Section 4, we provide an overview of *ObliviousDB*. Solution and construction details of *ObliviousDB* can be found in Sections 5 and 6, respectively. In Section 7, we analyse security of *ObliviousDB*. Section 8 reports the performance. Finally, we conclude this paper in Section 9.

2 Preliminaries

In this section, in regards to *ObliviousDB*, we set the context and informally define some of the properties that are supported by *ObliviousDB*.

Often, encrypted searchable schemes are associated with Searchable Symmetric Encryption (SSE), where documents can be encrypted and associated with a set of keywords that are also encrypted. An encrypted search is performed by matching encrypted keywords with keyword tokens that constitute encrypted queries. However, in SSE, the document collection cannot be changed once it has been encrypted and encrypted keywords are generated. Oblivious RAM (ORAM) [5–7] has been proposed in this context. It offers a great level of privacy but it is very costly and static. In contrast, several works have been proposed for Dynamic SSE (DSSE), where the document and keywords can be inserted and/or deleted. The main disadvantage is that it leaks information to the cloud server.

ObliviousDB is a searchable encryption scheme that supports complex SQL-like queries including conjunctions, disjunctions and range queries. Nevertheless, *ObliviousDB* could also be used to store encrypted documents (as objects in a database) and support simple keyword queries to search and retrieve documents.

Definitions of information leakage that have been proposed in the literature are mostly capturing the SSE or DSSE schemes, where queries are represented by a simple keyword search [12]. In *ObliviousDB*, these definitions need to be adapted to include more general queries with complex WHERE clauses. In the following, we will informally define some of the properties for reducing leakage of information. Later, in Section 3, we used these properties to classify related work.

The first property is **Search Pattern Privacy (SPP)**, which requires that the cloud server should not be able to distinguish if two (or more) queries are the same or not. This

property can be achieved if the scheme used for encrypting the query is semantically secure, where the encryption of the same queries should generate different ciphertexts.

However, even if a searchable scheme achieves SPP, the cloud server could learn if two queries are the same or not just by looking at the identifiers of the matching encrypted records in the result set. If the result sets return records with the same identifiers, the cloud server can guess with a high probability that the two queries are the same.This is referred in literature as *access pattern* leakage [12]. Therefore, a scheme achieves **Access Pattern Privacy (APP)** if the cloud server is not able to infer anything by just looking at the records identifiers in the result set of a query.

Size pattern [12] refers to the size of the result set in our context. A scheme achieves **Size Pattern Privacy (SzPP)** if the cloud server is not able to infer the real size of the result set for a query.

In the context of databases, a query is not just a retrieving operation but it might be an insert, update or delete. When the cloud server is able to learn the operation executed by a query, we refer to this information leakage as *operation pattern*. Hence, **Operation Pattern Privacy (OPP)** is supported if the scheme is able to hide the operation pattern from the cloud server.

Forward and backward privacy were first introduced in [10] and defined in the context of SSE with a simple keyword-document setting. In this work, we stress that we are focusing on more general database settings, where forward privacy means that the cloud server does not learn if a new or updated record r matches a query q executed in the past. Backward privacy means that the cloud server is not able to executed queries on records that have been deleted or modified.

Table 1. The comparison of searchable encryption schemes.

Schemes	**Search pattern privacy**	**Access pattern privacy**	**Size pattern privacy**	**Operation pattern privacy**	**Forward privacy**	**Backward privacy**	**Sub-linear search**	**Key management**
Hang *et al.* [13]	×	×	×	×	×	×	×	○
Ferretti *et al.* [14]	×	×	×	×	×	×	×	○
Sarfraz *et al.* [15]	×	×	×	×	×	×	×	●
Sun *et al.* [16]	✓	×	×	×	×	×	×	○
Asghar *et al.* [11]	×	×	×	×	×	×	×	●
Stefanov *et al.* [10]	×	×	×	×	✓	×	✓	○
Hahn *et al.* [17]	×	×	×	×	×	×	✓	○
Kamara *et al.* [18]	×	×	×	×	×	×	✓	○
Cao *et al.* [19]	✓	×	×	–	Static	Static	×	◐
Naveed *et al.* [20]	×	×	✓	✓	×	×	×	○
Wang *et al.* [21]	✓	✓	✓	–	Static	Static	×	◐
Naveed [22]	✓	✓	✓	–	Static	Static	–	○
Our work	✓	✓	✓	✓	✓	✓	✓	●

✓ and × indicate that the property is achieved or not achieved, respectively. ○ represents a single user scheme. ◐ represents a semi-fledged multi-user scheme. ● represents a full-fledged multi-user scheme. Static means it is not possible to insert or delete the data.

3 Related Work

In this section, we discuss the approaches presented in the literature. Since the seminal paper by Song *et al.* [23], many searchable schemes have been proposed and the research in this area has been extended in several directions. In this work, we focus mainly on three aspects of the encrypted search: information leakage, search efficiency and key management. As such, the following discussion on related work is organised around these three aspects. Table 1 categories the literature based on these aspects.

Several works have concentrated on supporting multi-user access and simplifying key management. Hang *et al.* [13] present a scheme that supports complex queries in a multi-user scenario. This scheme also supports a collusion-resistant mechanism by encrypting the data with different access rights using different encryption keys. However, these keys have to be shared among the authorised users. This means a single compromised user will require all others users to get a new set of keys, thus making this SU scheme inefficient in handling user revocation.

Ferretti *et al.* [14] introduce a scheme that is resistant against collusions. They propose a hierarchical encryption mechanism to indirectly share the secret key among multiple users. However, it requires multiple re-encryption operations to deal with user revocation. Not only the data should be re-encrypted with a new secret key, but also each layer in the hierarchical structure should be re-encrypted. Although the keys are shared in a indirect way, strictly speaking, [14] is the SU scheme.

Sarfraz *et al.* [15] design the FFMU searchable scheme with a fine-grained access control by leveraging the attribute based group key management scheme as introduced in [24]. Instead of assigning the key to users, they store them into a proxy. Since the user never knows the underlying encryption key, it does not require to change the key for user revocation. The problem is that this mechanism requires the proxy to be online for performing operations on behalf of the users. As a result, the proxy represents a single point of failure: an attacker that compromises the proxy will gain access to all the logged-in users' keys and data.

Sun *et al.* [16] utilise a CP-ABE mechanism to achieve a scalable and searchable FFMU scheme that supports multi-user read and write operations without sharing any key. However, for user revocation, the data has to be re-encrypted with a new access structure and secret keys of all the other users need to be updated with a new attribute set. Strictly speaking, this scheme is also the SU scheme.

Asghar *et al.* [11] propose the FFMU scheme with an efficient and flexible key management method, where each user has her own key and does not require any re-encryption when an authorised user is revoked. This key management mechanism is integrated in the work presented in this paper.

All of the above schemes neither provides any protection against information leakage nor supports sub-linear search.

Both [10] and [17] present sub-linear SSE schemes. In [10], the authors proposed a dynamic sub-linear searchable scheme based on the hierarchical data-structure and it achieves forward privacy by dynamically encrypting the data with fresh keys. This focus mainly on supporting a high-rate query throughput and exploits the data in RAM and parallel computation. The scheme achieves high level performance but compromises on

information leakage, despite it supports forward privacy. The scheme only supports the SU key management mechanism, where all the users share a single key.

The scheme presented in [17] achieves asymptotically optimal search efficiency by learning the inverted index from the access pattern. Kamara *et al.* leverage the advances in multi-core architectures and proposed a sub-linear dynamic parallel searchable scheme [18]. Unfortunately, both approaches do not take any measures to hide SPP, APP, OPP and SzPP, and failed to achieve forward and backward privacy. Moreover, all of them are SU schemes, where the secret keys have to be shared among users.

Several recent works have addressed the issue of information leakage. Cao *et al.* [19] design a scheme that supports multi-keyword ranked searches. The scheme achieves SPP by hiding the trapdoor linkability. However, they do not protect the access pattern from which the search pattern can still be inferred. Although they provide an index for speeding up search operations, this requires a static building procedure and is very expensive. As key management, this scheme is the SFMU scheme, where users have different access rights linked to the key they own. Here, the data owner does not share the secret key with the other users, but it has to generate the search tokens and decrypt the search results for them.

In [20], Naveed *et al.* achieve SzPP. The basic idea is to divide each document into a set of blocks. When a document is requested, a larger set of blocks will be downloaded and decrypted by the client. This scheme also achieves OPP since the cloud server only sees uploads and downloads of data blocks. However, it aggravates the computation and storage overheads on the client side. Moreover, it fails to achieve both SPP and APP, since the same query always requests the same block set. Besides, this scheme is the SU scheme.

Wang *et al.* [21] propose a public multi-keyword searchable encryption scheme that achieves SzPP, SPP and APP. However, it suffers from the same problems as [19]. First, the construction is static and does not support insert and delete operations. Also, it is the SFMU scheme. For each query, the reader has to get the encrypted keywords and decrypted search results from the writer. Besides, the search efficiency of this scheme is linear.

Naveed [22] analyse the applicability of ORAM to SSE. He emphasise that it is necessary to stream the entire outsourced data to achieve SPP. Otherwise, the cloud server can distinguish one query from another, since the sizes of the retrieved results are different. However, this point is not tenable if the retrieved data size is variable for all queries, no matter they are same or not.

From our discussion on related work, it is clear that none of the reviewed approaches are able to limit information leakage, support sub-linear search or a flexible key management mechanism that is FFMU. We stress here that our approach is the first to address all of the three aspects.

4 Overview of *ObliviousDB*

In this section, we provide an overview of the proposed approach.

4.1 System Model

The system involves five main entities shown in Figure 1:

- **Cloud Server (CS)**: A CS is part of the infrastructure provided by a cloud service provider, such as Amazon S3 [25] and Google Drive [26]. It stores the encrypted data and access policies to regulate access on the data. If an access policy is satisfied, it executes encrypted queries on the encrypted data.
- **Database Administrator (DBA)**: The DBA is responsible for the management of databases (including creation of tables and dropping them), access control policies for regulating access to tables in the database (including selecting, inserting, updating and deleting records in the table) and database users.
- **Database User (DBU)**: It represents the authorised user who can execute select, insert, update and delete queries over the encrypted data. After executing encrypted queries, a DBU can retrieve the result, if any, and decrypt it.
- **Oblivious Proxy Server (OPS)**: It is employed for achieving more security and search efficiency. It serves as a proxy between DBUs and the CS. In order to hide sensitive information, it pre-processes DBU queries and filters out the result, if any, for DBUs. For improving the performance, it stores some indexing information. Technically, it 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 revokes the DBU, when she is compromised or no longer part of the database.

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 and a new DBU is created or existing one is removed from the system. For normal operations it can be taken off-line. In this way the organisation can easily secure the KMA from external attacks.

We consider that DBUs are responsible for keeping their keys (and decrypted data) securely. DBUs can collude together as well as DBUs and the CS can also collude, but they do not learn more than what each of them can learn individually.

The CS is modelled as 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.

We assume the OPS is semi-trusted. In particular, the OPS is employed to strengthen data privacy. However, it could be compromised given it is responsible for the communication with external world. Therefore, the data stored on the OPS is possibly 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, 27] can be utilised in *ObliviousDB*.

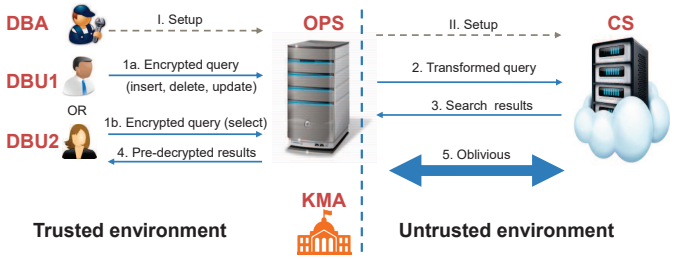


Fig. 1. Overview of *ObliviousDB*: A DBU is responsible for running setup (Step I then Step II). A DBU can insert the data (Step 1a) or execute a select query (Step 1b) to receive matching records (Step 4). Regardless of the query type, to control information disclosure, the OPS always transforms the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

4.2 Proposed approach

ObliviousDB represents the first practical encrypted scheme for database that support efficient search with controllable leakage. Using *ObliviousDB*, queries can be executed without any pattern leakage including SPP, APP, SzPP and OPP. To achieve SPP, we encrypt queries using a semantically secure encryption scheme. APP, SzPP and OPP are achieved by running the Oblivious algorithm explained in Section 5.4. In order to achieve efficiency, an indexing mechanism is implemented. Technically, we divide the data into groups, which enables sub-linear search. To promise forward and backward privacy, *ObliviousDB* uses nonces.

In *ObliviousDB*, a DBA initialises the system by setting up the OPS (Step I) and the CS (Step II) as illustrated in Figure 1. After the system is initialised, the DBA can add DBUs. For each DBU, keying material is generated by bringing online the KMA. After a DBU receives her keying material, she can execute encrypted queries. In case of an insert query (Step 1a), a DBU encrypts the query and sends to the OPS and does not need to expect any result set[1] However, if the query is select (Step 1b), a DBU receives some pre-decrypted results, which are finally decrypted by the DBU using her private key. Once the OPS receives an encrypted query, no matter the operation type, the OPS and the CS run a predefined protocol (consisting of Steps 2, 3 and 5) in order to achieve OPP. The OPS first performs transformation and then sends the transformed query to the CS (Step 2). The purpose of this transformation is to rewrite the query, without performing decryption, based on indexing information stored by the OPS.

It is important to note that, although the data is encrypted, the CS can still infer some statistical information from the encrypted data. For preventing such inference, a number of dummy records are inserted by the OPS. Therefore, when the OPS receives search results back from the CS (Step 3), it can easily filter out dummy records. After

[1]The actual implementation of the protocol sends back an acknowledgement that the insert operation has succeeded or an error code, otherwise. However, these steps are not shown here as they do not require any encryption/decryption operations.

receiving the results back, the OPS can achieve obliviousness by executing update, delete and insert as part of the oblivious algorithm (Step 5).

In *ObliviousDB*, the OPS knows the query type and number of records in the result set; however, it cannot distinguish queries or records, which are encrypted using randomised encryption and are explained in Sections 5 and 6. Unlike existing solutions (such as CryptDB [4]), we assume that the OPS can also be compromised, but the adversary can neither learn group information, which is encrypted, nor she can distinguish between real and dummy entries, unless she colludes with the CS. However, such collusion does not reveal content of the database.

5 Solution Details

ObliviousDB limits information leakage, offers forward and backward privacy, implements indexing mechanism to enable sub-linear search and supports full-fledged multi-user access. In this section, we provide details of how we achieve desired features.

5.1 Key Management

One of the main aspects of *ObliviousDB* is to employ a flexible key management approach, which not only supports FFMU access, but also supports efficient DBU registration and revocation. In this regard, we build on top of [11] in which we assume a simplistic model involving two main entities: namely a DBU and a CS, where the latter one manages the server side keys for the proxy encryption [28]. Unfortunately, the collusion between a single compromised DBU and the curious CS will render the system-wide master secret key, thus rendering the encryption useless. In contrast, in *ObliviousDB*, we limit such collusion attacks by managing the server side keys at the OPS.

The KMA is initialised with some security parameters in order to generate public parameters and a master secret key MSK, which is securely stored by the KMA. After the KMA is initialised, it generates and distributes the key material. For each DBU i, the KMA splits MSK, generates a pair of keys (K_{U_i}, K_{P_i}) and distributes them to the DBU and the OPS, respectively. The DBU securely keeps her user side key K_{U_i}. The OPS stores the corresponding OPS side key K_{P_i} in the key stored managed by the OPS. The DBU performs encryption before storing the data or processing query over encrypted data on the CS. Basically, all the data stored on the CS gets encrypted under MSK, which is only known to the KMA. This is what makes *ObliviousDB* FFMU, where entities do not share any key and each DBU holds a unique key. Any registered DBU can retrieve the encrypted data. To do so, the OPS fetches the data from the CS, performs the pre-decryption using K_{P_j} for the DBU j, who can finally decrypt the data using her private key K_{U_j}.

In *ObliviousDB*, if two or more DBUs collude putting together their user side keys they cannot obtain the master key. To do so, a DBU should collude with a DBA to put together the user side and the corresponding key on the OPS: possible but a very unlike event. An adversary cannot learn sensitive information from the data exchanged between the DBU and the OPS as well as the OPS and the CS. Compromising the

CS will not reveal any information. However, compromising the OPS will only reveal limited indexing information about the data being accessed by DBUs that are logged in. If no DBU is logged in, compromising the OPS will not put any data at risk. Considering the OPS is trusted, we do not consider the collusion between the malicious DBU and the OPS.

A DBU (say a compromised one) can be removed from the system. In order to revoke her access, the OPS is instructed to remove the OPS side key corresponding to the DBU. Consequently, the revoked DBU would not be able to store, retrieve the data or execute any query due to missing her OPS side key.

Table 2. A sample database table (a) *staff* (which is viewed by DBUs) and (b) its indexing information (only a logical view). (c) indexing information of encrypted data is stored by the OPS. Whereas, the CS stores (d) the encrypted *staff* table.

(a) *Staff*

ID	Name	Age
1	Alice	25
2	Anna	30
3	Bob	27

(b) Groups

GID	Index List
gid_1	$\{1,3\}$
gid_2	$\{2\}$
gid_a	$\{1,2\}$
gid_b	$\{3\}$

(c) Encrypted groups

GID	Index List
$\{gid_1\}_{GE}$	$(n_1,\{1,4\}),(n_3,\{3\})$
$\{gid_2\}_{GE}$	$(n_2,\{2\})$
$\{gid_a\}_{GE}$	$(n_1,\{1,4\}),(n_3,\{2\})$
$\{gid_b\}_{GE}$	$(n_3,\{3\})$

(d) Encrypted *Staff*

ID	$\{Name\}_{SE}$	$\{Name\}_{DE}$	$\{Age\}_{SE}$	$\{Age\}_{DE}$
1	$\{Alice\}_{SE_{n_1}}$	$\{Alice\}_{DE}$	$\{25\}_{SE_{n_1}}$	$\{25\}_{DE}$
2	$\{Anna\}_{SE_{n_2}}$	$\{Anna\}_{DE}$	$\{30\}_{SE_{n_2}}$	$\{30\}_{DE}$
3	$\{Bob\}_{SE_{n_3}}$	$\{Bob\}_{DE}$	$\{27\}_{SE_{n_3}}$	$\{27\}_{DE}$
4	$\{Alice\}_{SE_{n_1}}$	$\{XYZ\}_{DE'}$	$\{25\}_{SE_{n_1}}$	$\{00\}_{DE'}$

5.2 Data Representation

Using *ObliviousDB*, DBUs can store or retrieve data while preserving confidentiality in the cloud. To achieve this, we employ the Data Encryption (DE) scheme. Since DBUs should be able to perform encrypted search, we use the Searchable Encryption (SE) scheme. Both DE and SE are based on the proxy encryption scheme [28]. Table 2 illustrates an example of how we represent and store the data. Let us assume that we have a table Staff (Table 2(a)) containing name and age attributes. The CS store an encrypted version of this, which is illustrated in Table 2(d), where each attribute is encrypted under SE as well as DE. Similarly, we encrypt each value in the table. Note that SE and DE representations do not leak information about encrypted values. Although SE is semantically secure, using it for encrypting data or query may leak information on number of matching records returned by the CS. That is, SE alone does not guarantee SzPP. In order to achieve SzPP, the OPS adds some dummy records. The idea is that the CS should not be able to distinguish between a dummy and a real record when any search is performed. Therefore, the OPS picks dummy records (the SE part only) from existing records already stored on the CS. For instance, the last record in Table 2(d) is a dummy record and its SE part is generated based on the first record.

Efficiency is another important concern for searchable scheme. In order to improve the search efficiency, we support indexing, which enables sub-linear search. Technically, we divide the data into groups and performs search within a group instead of the whole table. For example, we can group the values of Name in the Staff table based on the first letter. That is, both Alice and Anna belong to group gid_a; whereas, Bob is part

of group gid_b, as illustrated in Table 2(b). Similarly, we can divide age into two groups: say both 25 and 27 are in group gid_1 while 30 is in group gid_2. The group function that is used should be made public to all the DBUs. An alternate approach is to manage the group information at the OPS (or even the CS), which could only be retrieved by the DBUs. In the former case, since group information is public, it could reveal information about the data if not protected, in particular if the OPS gets compromised. In order to avoid an extra round between the DBU and the OPS to exchange group information, in this paper, we focus on publishing group functions and protect group information by employing Group Encryption (GE). Basically, we use GE to encrypt *gid*, but indices, belonging to a group, are stored in cleartext, as illustrated in Table 2(c). If we compare Table 2(b) and Table 2(c), we can notice that index list in former one is smaller than that of the latter one. The reason behind that is introduction of dummy records (*i.e.,* record 4 in Table 2(d)) added by the OPS to achieve SzPP.

It is worth mentioning that *ObliviousDB* also achieves both forward and backward privacy. That is, the CS is unable to run previous queries on new records, or run new queries on deleted records. To achieve both forward and backward privacy, the OPS includes a one-time nonce in each value in a record (or query). This nonce is updated [2] only for those values that are processed as a result of query execution. Only the query with a correct (*i.e.,* latest) nonce can generate valid results. Once the query is executed on the CS, the OPS updates the results' nonces. The executed query and the matched records marked with old nonce will not be valid anymore. The nonce is also managed by and stored on the OPS. As we can see in Table 2(c), indexes of the records marked with the same nonce are aggregated into a sublist. For instance, the index list $\{1,3,4\}$ for group gid_1 is divided into two sublists, $\{1,4\}$ for n_1 and $\{3\}$ for n_3. As shown in Table 2(d), the values encrypted under SE are augmented with nonces.

5.3 Query Execution

In this subsection, we discuss in details the steps executed when a query is processed in *ObliviousDB*. Figure 2 illustrates these steps. Let assume that the DBU wants to execute the following SQL-like query Q: *SELECT * FROM Staff WHERE Name= Alice AND Age*= 25. As illustrated in Stage 1 of Figure 2, the DBU client encrypts under SE the following: table name, attribute names and values in the query, similar to the approach proposed in [11]. DE is only used for insert and update queries to encrypt the new data. The DBU also calculates and encrypts group information using GE. The outcome of Stage 1 is an encrypted query EQ, which is sent to the OPS.

In Stage 2, the OPS transforms the encrypted query and gets two index lists IL and UL First, it checks if the encrypted group information is found in Table 2(c). Since the query includes two conditions, involving two encrypted groups $\{gid_a\}_{GE}$ and $\{gid_1\}_{GE}$, the lookup returns two different index lists: $(n_1,\{1,4\}),(n_3,\{2\})$ and $(n_1,\{1,4\}),(n_3,\{3\})$, respectively. As both conditions are conjuncted with *AND*, *Group-Match* outputs intersection IL and union set UL of both index lists, which are $\{n_1,\{1,4\}\}$ and $\{1,2,3,4\}$, respectively. IL is used to set the search range on the CS. UL will be

[2] Technically, updating a nonce and re-randomising an encrypted value achieve the same objective, though the former is for DE and the latter one is for SE, respectively.

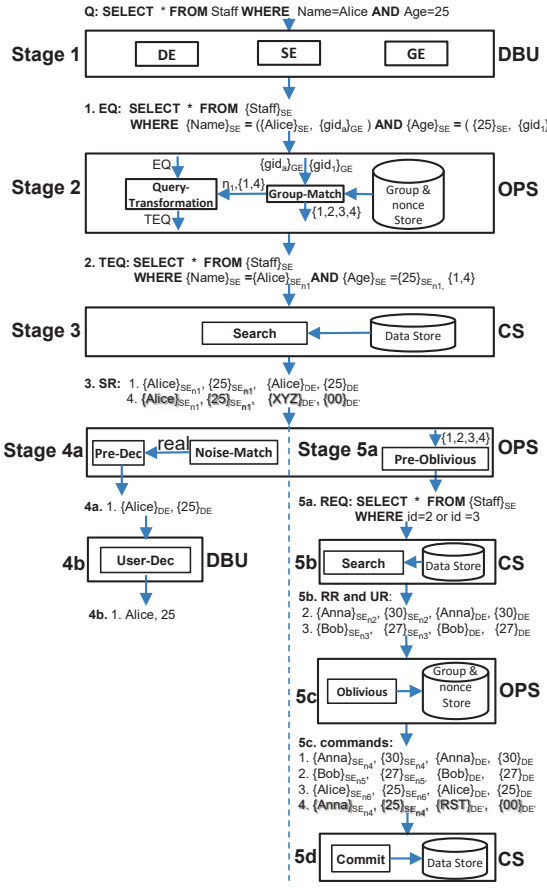


Fig. 2. An example query in process.

Algorithm 1 Oblivious

Input: The search result SR, the random result RR, the unmatched record set UR in UL, the encrypted query EQ, dummy records number du, real records number re and the threshold t indicating a ratio between du and re.

Output: A set of records to be updated UP, an indexes list of records to be deleted DEL, an insert query INS.

for each record rcd in SR **do**
  if $Noise\text{-}Match(rcd) = true$ **then**
    if $du/re < t$ **then**
      rcd =$Re\text{-}Randomise(rcd)$
      rcd =$Nonce\text{-}Update(rcd)$
      $UP = UP \cup rcd$
    else
      $DEL = DEL \cup id(rcd)$
  else
    if $EQ.type = select$ or $EQ.type = insert$ **then**
      rcd =$Re\text{-}Randomise(rcd)$
    if $EQ.type = update$ **then**
      rcd=update(rcd)
    if $EQ.type = delete$ **then**
      rcd =$Noise\text{-}Gen(rcd)$
    rcd =$Nonce\text{-}Update(rcd)$
    $UP = UP \cup rcd$
for each record rcd in RR **do**
  rcd =$Re\text{-}Randomise(rcd)$
  $UP = UP \cup rcd$
Shuffle UP
if $EQ.type = insert$ **then**
  EQ =$Nonce\text{-}Update(EQ)$
  $INS = EQ$
else
  Pick a record s from $RR \cup SR$ randomly
  rcd =$Noise\text{-}Gen(s)$
  rcd =$Nonce\text{-}Update(rcd)$
  $INS = insert(rcd)$
for each record rcd in UR **do**
  rcd =$Nonce\text{-}Update(rcd)$
  $UP = UP \cup rcd$
return UP, DEL and INS to the CS

used in the oblivious algorithm. As we can see in Figure 2, the nonce (*i.e.,* n_1) is used by *Query-Transformation* to update values in the query that are encrypted under SE. If the query is *INSERT*, a new nonce is generated and Table 2(c) is updated accordingly. As we already explained in Section 5.2, these nonces ensure forward and backward privacy. The OPS generates a transformed encrypted query TEQ that includes indices where the requested information could be located.

Stage 3 is to execute TEQ on the CS. For each index, the CS searches if the *WHERE* clause evaluates to *true*. The CS matched records that includes real (*i.e.,* $\{1\}$) and some dummy ones (*i.e.,* $\{4\}$). The CS returns search results SR to the OPS.

If the original query issued by the DBU is *SELECT*, the OPS will filter out dummy elements from the SR by running *Noise-Match* and pre-decrypt the encrypted results as illustrated in Stage 4a. The pre-decrypted results can only be decrypted by the DBU. The DBU client runs *User-Dec* to finally decrypt the results (see Stage 4b).

No matter the query type, an Oblivious algorithm is run between the OPS and the CS in Stage 5. Before that, in Stage 5a, for achieving both SPP and APP, the OPS runs *Pre-Oblivious* to ask for two sets of records UR and *RR* from the CS. UR is the unmatched record set in UL. *RR* is a set of random records, which has the same size as *SR*.

One can argue why we need to execute a select again when one (*i.e.,* TEQ) is already executed after Stage 2. There are two main reasons. First, if the original query is select, the DBU should not experience high latency. That is, the OPS will immediately send back results to the DBU after Stage 2. Second, REQ is issued to better achieve APP and SzPP *i.e.,* by maximising number of records that are shuffled. It is important to note that REQ is always fake, but it does not leak information on the original query, which could be one of select, insert, update and delete. REQ contains some valid indices (*i.e.,* 2 and 3) that can be located on the CS. After executing REQ, in Stage 5b, the CS returns results RR. In Stage 5c, the OPS runs an Oblivious algorithm, which is explained in Section 5.4 in detail. The output of Stage 5c is a list of commands that will be committed by the CS in Stage 5d.

5.4 Oblivious Algorithm

The Oblivious algorithm ensures APP, OPP and SzPP as well as forward and backward privacy. To achieve SzPP, the OPS adds a set of dummy records. OPP is guaranteed by executing all four queries (*i.e.,* select, insert, update and delete) no matter what the original query is. Every time after a query is executed, shuffling of matched records (including real and dummy ones) together with re-randomisation ensures APP. The nonce in each query and each record in the result set promises both forward and backward privacy.

The algorithm takes as input the search result *SR*, the return result *RR*, the unmatched records UR in UL and the encrypted query EQ. The OPS also keeps the total number of dummy records du, the total number of real records re and the threshold t indicating a ratio between dummy and real records. It also takes as input du, re and t. It outputs a set of records to be updated UP, an index list of records to be delete DEL and an insert query INS.

The algorithm is executed as follows. First, for each record in *SR* (Line 1), it evaluates if the record is a dummy one (Line 2). If so, it checks if the ratio between *du* and *re* is less than t (Line 3) so that we can control du. In case of yes, it re-randomises the record (Line 4), updates nonces on the record (Line 5) and adds the record to the UP list (Line 6). Ultimately, we achieve APP by re-randomising the record including DE and SE. At the same time, updating nonce (Line 5) ensures forward and backward privacy. Otherwise, (Line 7), it adds the record to the DEL list (Line 8) so that we remove unnecessary volume of dummy records. If the record is real (Line 10), there are four cases: select, insert, update and delete. If the query is select or insert (Line 10), we re-randomise the record (Line 11) to achieve APP. If the query is update (Line 12), the SQL update is run on the record (Line 13). Since the record is update, we do not need to re-randomise because we automatically achieve APP. If the query is delete (Line 14), the algorithm generates a dummy record out of the original one (Line 15) and achieves SzPP. Consequently, the CS cannot learn if the original query is delete. No matter the query type, in order to ensure forward and backward privacy, it updates the nonce (Line 16) before adding the record to the UP list (Line 17). After all records in *SR* are processed, to achieve APP, the algorithm re-randomises each record in *RR* (Line 19) and adds re-randomised records to the UP list (Line 20), which are finally shuffled (Line 21). If the original query is insert (Line 22), we update the nonce (Line 23) and then mark the record as *INS* (Line 24). Otherwise (Line 25), to achieve OPP as well as SzPP, a record is chosen randomly from *SR* or *RR* (Line 26) in order to generate a dummy record (Line 27). Next, its nonce is updated (Line 28) and it becomes the new SQL insert (Line 29). To achieve SPP, we need to update all unmatched records UR of the group that was part of the query (Lines 30-32).

By running all four types of queries, we achieve OPP. Finally, the algorithm returns UP, DEL and *INS* (Line 30).

6 Definition And Construction Details

This section gives the definitions and details of algorithms used by different modules including the KMA, the DBU and the OPS.

6.1 Definitions

The proposed scheme consists of the following algorithms:

- **Init**(1^k). It is a probabilistic algorithm run by the KMA. It takes as input a security parameter k and outputs the public parameter *Param* and the master secret key set MSK.
- **Key-Gen**(MSK, i). It is a probabilistic algorithm run by the KMA to generate keying material for all entities. For each DBU i, using MSK, it generates two key sets K_{U_i} and K_{P_i}, where K_{U_i} is the user key set for the DBU i and K_{P_i} is its corresponding OPS side key.
- **GID-Gen**(D). It is a deterministic algorithm run by the DBU i to generate the gid for data D. It takes as input the data D, and outputs gid.

- **GE-Enc**(gid). It is a deterministic algorithm run by the DBU to encrypt the group information. It takes as input *gid* and outputs the encrypted group information $GE(gid)$. In the following, we simplify $GE(gid)$ as $GE(D)$ for clear description.
- **SE-Enc**(D, K_{U_i}). It is a probabilistic algorithm run by the DBU i to encrypt D. It takes as input the data D and the key K_{U_i}. It outputs the encrypted data $SE(D)$, which is used for keyword search.
- **DE-Enc**(D). It is a probabilistic algorithm run by the DBU to encrypt the records. It results in $DE(D)$ from which the original data element D can be retrieved.
- **Nonce-Update**$(SE(D))$. It is a probabilistic algorithm run by the OPS to add or update the nonce on $SE(D)$, It takes as input $SE(D)$, and outputs the nonce n and $SE'(D)$.
- **Noise-Gen**$(GE(D), SE(D))$. It is a randomised algorithm run by the OPS to generate a dummy record. It takes as input the real encrypted data $SE(D)$ and $GE(D)$, and outputs a dummy record $(SE'(D), DE(D'))$ that can be matched with interested keyword D but indistinguishable from real data by the CS.
- **Re-Randomise**$(SE(D), DE(D))$. It is a probabilistic algorithm run by the OPS to update the encrypted data stored in the CS. It takes as input the encrypted record $(SE(D), DE(D))$ and outputs $(SE'(D), DE'(D))$.
- **Group-Match**$(GE(Q), GE(D))$. The OPS runs this deterministic algorithm to check if the DBU generated trapdoor $GE(Q)$ matches with the stored encrypted group information $GE(D)$. This algorithm outputs *true* if there is a match and *false* otherwise.
- **Data-Match**$(SE(Q), SE(D))$. The CS runs this deterministic algorithm to check if the encrypted data $SE(D)$ stored in the CS matches with the query $SE(Q)$. It returns *true* if there is a match and *false* otherwise.
- **Search**$(Indexlist, SE(Q))$ It is a probabilistic algorithm run by the CS to get the matched records for query Q. With the input $(Indexlist, SE(Q))$, it returns the result set R to the OPS.
- **Noise-Match**$(GE(Q), DE(D))$. This deterministic algorithm is run by the OPS to check if an encrypted record $DE(D)$ is dummy or not. It outputs *true* if $DE(D)$ is dummy and *false* otherwise.
- **OPS-Pre-Dec**$(DE(D), K_{P_j})$. The OPS runs this deterministic algorithm to partially decrypt the encrypted data $DE(D)$ for the DBU j. It takes $DE(D)$ and K_{P_j} as input and results in the pre-decrypted data $DE_j^*(D)$.
- **User-Dec**$(DE_j^*(D), K_{U_j})$. The DBU j runs this algorithm to finally decrypt the data $DE_j^*(D)$. It takes as input $DE_j^*(D)$ and K_{U_j}, and returns the plaintext data D.
- **Revoke**(i). The DBA runs this algorithm to revoke DBU i from *ObliviousDB*.

6.2 Concrete Construction

- **Init**(1^k). The KMA takes as input the security parameter k. It outputs a prime number p, two multiplicative cyclic groups $\mathbb{G}$ and $\mathbb{G}_\mathbb{T}$ of order p. It defines a bilinear map: $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_\mathbb{T}$, which has the properties of *bilinearity*, *computability* and *non-degeneracy* [29]. Let g be the generator of $\mathbb{G}$. It chooses a random x from Z_p^* and outputs $h = g^x$. Next, it chooses a collision-resistant keyed hash function H, two pseudorandom functions ψ and f and a random key s for f. It also initialises

the key store managed by the OPS. That is, $K_S \leftarrow \phi$. Finally, it publishes the public parameters $Params = (e, \mathbb{G}, \mathbb{G}_\mathbb{T}, p, g, h, H, f, \psi)$ and keeps securely the master secret key $MSK = (x, s)$.

- **Key-Gen**(MSK, i). For the DBU i, the KMA splits MSK into two values x_{i1} and x_{i2}, where $x = x_{i1} + x_{i2}$ and $x_{i1}, x_{i2} \in Z_p^*$. Then, the KMA transmits $K_{U_i} = (x_{i1}, s)$ and $K_{P_i} = (i, x_{i2})$ securely to the DBU i and the OPS, respectively. The OPS adds K_{P_i} to its key store: $K_S \leftarrow K_S \cup K_{P_i}$.
- **GID-Gen**(D). The DBU i generates gid for D by computing $gid \leftarrow \psi(D)$.
- **GE-Enc**(gid). The DBU encrypts gid by computing $GE(gid) \leftarrow f_s(gid)$. Finally, $GE(gid)$ is sent to the OPS.
- **SE-Enc**(D, K_{U_i}). For the data D, the DBU i first computes $\sigma \leftarrow f_s(D)$, then chooses a random r from Z_p^*. Next, the DBU i computes $SE(D) = (c_1, c_2)$, where $c_1 = g^r$ and $c_2 = g^{\sigma r}$. Finally, the DBU i sends $SE(D)$ to the OPS.
- **DE-Enc**(D). For the data D, the DBU chooses a random r from Z_p^* and calculates $DE(D) = (e_1, e_2)$, where $e_1 = g^r$ and $e_2 = h^r D$. Finally, $DE(D)$ is sent to the OPS.
- **Nonce-Update**$(SE(D))$. With the input $SE(D) = (c_1, c_2)$, the OPS get $SE'(D) = (c_1', c_2')$ by computing $c_1' = c_1^n, c_2' = c_2$, where n is random from Z_p^*. $SE'(D)$ is sent to the CS. n is stored in the OPS.
- **Noise-Gen**$(GE(D), SE(D))$. With the input $SE(D) = (c_1, c_2)$ and $GE(D)$, the OPS runs this algorithm to generate a dummy record $(SE'(D) = (c_1', c_2'), DE(D') = (\mathfrak{e}_1, \mathfrak{e}_2))$, where $c_1' = c_1^r$ and $c_2' = c_2^r$, r is a random from Z_p^*, $\mathfrak{e}_2$ is a random chosen from $\mathbb{G}$ and $\mathfrak{e}_1 = H_{GE(D)}(\mathfrak{e}_2)$.
- **Re-Randomise**$(SE(D), DE(D))$. The OPS chooses a random r from Z_p^* to update the encrypted record. The updated $SE'(D) = (c_1', c_2')$, where $c_1' = c_1^r$, $c_2' = c_2^r$. The updated $DE'(D) = (e_1', e_2')$. If this record is real, $e_1' = e_1 \cdot g^r$ and $e_2' = e_2 \cdot h^r$. Otherwise, e_2' is new random, and $e_1' = H_s(e_2')$.
- **Group-Match**$(GE(Q), GE(D))$. When receiving $GE(Q)$ from the DBU, the OPS fetches the $GE(D)$ from the group store. This algorithm checks if $GE(Q) \stackrel{?}{=} GE(D)$. The OPS runs this algorithm to insert the index of a new record into its corresponding index list or to get the index list to narrow down the search scope.
- **Data-Match**$(SE(Q), SE(D))$. This algorithm, which is run by the CS, is used to determine if the encrypted data $SE(D) = (c_{1_D}, c_{2_D}) = (g^{rn_i}, g^{r\sigma_D})$ is matched the query $SE(Q) = (c_{1_Q}, c_{2_Q}) = (g^{rn_j}, g^{r\sigma_Q})$ by checking if $e(c_{1_D}, c_{2_Q}) \stackrel{?}{=} e(c_{1_Q}, c_{2_D})$. On successful match, it returns *true* and *false* otherwise.
- **Search**$(Indexlist, SE(Q))$. The CS runs this algorithm to find which record in *indexlist* matches with $SE(Q)$. *Data-Match* is invoked to check if the related field in a record match with the corresponding condition node in $SE(Q)$. The record matches with $SE(Q)$ will be returned.
- **Noise-Match**$(GE(Q), DE(D))$. $DE(D) = (e_1, e_2)$ is the returned result. The OPS runs this algorithm to filter out the dummies for decryption. It is performed by checking if $e_1 \stackrel{?}{=} H_{GE(Q)}(e_2)$. It yes, it means this record is dummy; otherwise, it is a real record.[3]

[3]There is a negligible probability that a real record is classified as a dummy one. To address this issue, we can check if $e_1 \stackrel{?}{=} H_{GE(D)}(e_2)$ when $DE(D) = (e_1, e_2)$ is generated. If yes, we can re-randomise $DE(D)$ and check again.

- **OPS-Pre-Dec**$(DE(D), K_{P_j})$. The OPS runs this algorithm to partially decrypt the data using K_{P_j}. The ciphertext $DE(D)$ is decrypted as $\widehat{e_2} = e_2 \cdot (e_1)^{-x_{j2}} = g^{r^{`} \cdot x_{j1}} D$. The OPS sends $DE^*_j(D) = (\widehat{e_1}, \widehat{e_2})$ to the DBU $j$, where $\widehat{e_1} = g^{r^{`}}$.
- **User-Dec**$(DE^*_j(D), K_{U_j})$. The DBU j decrypts the cihpertext as $D = \widehat{e_2} \cdot (\widehat{e_1})^{-x_{j1}} = g^{rx_{j1}} \cdot D \cdot g^{-rx_{j1}}$.
- **Revoke**(i). Given DBU i, the DBA removes K_{P_i} from the key store managed by the OPS. That is, $K_S \leftarrow K_S \backslash K_{P_i}$.

6.3 Correctness

- **Data-Match**. The encrypted data used for search is $SE(D) = (c_{1_D}, c_{2_D})$. The query is encrypted as $SE(Q) = (c_{1_Q}, c_{2_Q})$. For each query, the CS performs the match between $SE(D)$ and $SE(Q)$ by checking if

$$e(c_{1_D}, c_{2_Q}) \stackrel{?}{=} e(c_{1_Q}, c_{2_D}) \tag{1}$$

where

$$\begin{aligned} e(c_{1_D}, c_{2_Q}) &= e(g^{r_D n_i}, g^{r_Q \sigma_Q}) \\ &= e(g,g)^{r_D r_Q n_i \sigma_Q} \\ e(c_{1_Q}, c_{2_D}) &= e(g^{r_Q n_j}, g^{r_D \sigma_D}) \\ &= e(g,g)^{r_Q r_D n_j \sigma_D} \end{aligned}$$

It is true, iff $n_i = n_j$ and $\sigma_Q = \sigma_D$, where $\sigma_D \leftarrow f_s(D)$ and $\sigma_Q \leftarrow f_s(Q)$. Namely, there a match between the record and the query only when the related data are same and marked with the same nonce.
- **Noise-Match**. The OPS distinguishes a dummy record from a real one though their DE part. For a dummy record, $DE(D') = (\mathfrak{e}_1, \mathfrak{e}_2)$, where $\mathfrak{e}_2$ is a legal random and $\mathfrak{e}_1 = H_{GE(D)}(\mathfrak{e}_2)$. Noise-Match is performed by checking if

$$e_1 \stackrel{?}{=} H_{GE(Q)}(\mathfrak{e}_2) \tag{2}$$

Namely,

$$H_{GE(D)}(\mathfrak{e}_2) \stackrel{?}{=} H_{GE(Q)}(\mathfrak{e}_2) \tag{3}$$

It is true if $GE(D) = GE(Q)$. Since GE is deterministic, it needs $Q = D$. $GE(D)$ suggests the SD part of the dummy record is borrowed and re-randomised from data D. Q is the query being processed currently. According to Equation 1, only when $Q = D$ the record will be returned. Therefore, Equation 3 is always true if the record is dummy.
For a real record, $DE(D) = (g^r, h^r D)$, however, it is possible that $g^r = H_{GE(D)}(h^r D)$. In order to avoid the false negative, we can check if $e_1 \stackrel{?}{=} H_{GE(D)}(e_2)$ when $DE(D) = (e_1, e_2)$ is generated, and then re-randomise it and check again if yes. Fortunately, the false detection possibility is negligible.

7 Security Analysis

In this section, we analysis the security of *ObliviousDB*. Particularly, we prove SPP, APP, SzPP, OPP, forward and backward privacy.

SPP. Informally speaking, SPP means the CS can not learn if the queries are same or not from their ciphertext. *ObliviousDB* supports complex query, however, we did not take measures to hide the query structure, including the query length, the conjunctions and disjunctions. These leakage can be avoided by padding and reordering queries into same structure. Here we only define and prove SPP for the queries with same structures. Here we give its formal definition.

Definition 1 ***(SPP)***. *Let k be the security parameter of ObliviousDB over group $\mathbb{G}$. The SPP game between a Probabilistic Polynomial-Time (PPA) adversary $\mathscr{A}$ and a challenger $\mathscr{B}$ is described as below*

- ***Setup***. *The challenger $\mathscr{B}$ runs Init (1^k) and to get Params and MSK. Then he runs Key-Gen(MSK, i) to generate the secret key pair (K_{U_i}, K_{P_i}). He publishes Params to the adversary $\mathscr{A}$, and keeps MSK and (K_{U_i}, K_{P_i}) secret.*
- ***Phase 1***. *$\mathscr{A}$ requests $\mathscr{B}$ to encrypt a set of queries $\boldsymbol{Q} = \{q_1, ...q_t\}$. The only restriction is that all the queries should have the same structure. $\mathscr{B}$ generates the encrypted query and the index list (teq_j, l_j) for each query $q_j \in [1,t]$ by running SE, Nonce-Update and Group-Match. Then, $\mathscr{B}$ sends them to $\mathscr{A}$.*
- ***Challenge***. *$\mathscr{A}$ sends two pairs of queries $Q_0 = (q_0, q_0)$ and $Q_1 = (q_0, q_1)$. q_0 and q_1 can belong to $\boldsymbol{Q}$, but should have the same structure. $\mathscr{B}$ flips a coin $b \in \{0,1\}$, returns $< (teq_0, il_0), (teq_b, il_b) >$ to $\mathscr{A}$, where*

$$\begin{aligned} teq_0 &\leftarrow Nonce - Update(SE(q_0, K_{U_i})) \\ il_0 &\leftarrow Group - Match(SE(q_0, K_{U_i})) \\ teq_b &\leftarrow Nonce - Update(SE(q_b, K_{U_i})) \\ il_b &\leftarrow Group - Match(SE(q_b, K_{U_i})) \end{aligned}$$

- ***Phase 2***. *Same as phase 1.*
- ***Guess***. *$\mathscr{A}$ submits his guess b' of b.*

The advantage of $\mathscr{A}$ in this SSP game is defined as $Adv_{\mathscr{A}}^{SSP}(1^k)$. We say ObliviousDB achieves SPP, if for all PPA adversaries have at most negligible advantage in the above game:

$$Adv_{\mathscr{A}}^{SSP}(1^k) = Pr[b' = b] - 1/2 \leq negl(k) \tag{4}$$

where $negl(k)$ denotes as a negligible function in k.

Proof. The encrypted queries get by $\mathscr{A}$ are

$$< teq_0, teq_b > = < (c_1, c_2), (c_1', c_2') > = < (g^{nr}, g^{r\sigma}), (g^{n'r'}, g^{r'\sigma'}) > \tag{5}$$

where n and n' are the nonces stored on the OPS, $\sigma \leftarrow f_s(q_0)$, $\sigma' \leftarrow f_s(q_b)$. Since r and r' are random numbers that picked from Z_p^* uniformly. They are also independent of σ,

σ' and the nonces. This means just from the cipertext, $\mathscr{A}$ does not have advantage to success.

Alternatively, $\mathscr{A}$ could guess by checking if

$$e(c_1, c_2') \stackrel{?}{=} e(c_1', c_2) \tag{6}$$

As described in Equation 1, it is true only when $n' = n$ and $q_b = q_0$. According to *ObliviousDB*, after each query, the nonce in the corresponding group is updated. Since the nonce is random number picked from Z_p^* uniformly. The probability of getting same nonce is negligible. So in this way, we still have that

$$Adv_{\mathscr{A}}^{SSP}(1^k) = Pr[b' = b] - 1/2 \leq negl(k) \tag{7}$$

Besides, $\mathscr{A}$ also get the index list for search. In *ObliviousDB*, after the oblivious algorithm, not only the group size but also the index inside this group is refreshed. As a result, if there is an interaction between two index lists, $\mathscr{A}$ is uncertain if the two queries are in the same group.

SzPP. If the real size of the search result is unknown to the CS, the scheme achieves SzPP. In this work, the search result size is hidden by a random number of dummy records. If the CS wants to learn the size pattern, it should know which records in the search result set are real. Therefore, the SzPP prove could be reduced to prove the indistinguishability between dummy and real records. In order to achieve the SzPP, the dummy record should be indistinguishable from the real one in terms of both the DE and SE parts.

Proof. On the CS, a real record D and a dummy record are represented by $(SE(D), DE(D))$ and $(SE'(D), DE(D'))$, where

$$\begin{aligned} SE(D) &= (g^r, g^{r\sigma}) \\ SE'(D) &= (g^{r'}, g^{r'\sigma}) \\ DE(D) &= (g^r, h^r D) \\ DE(D') &= (H_{GE(D)}(\mathfrak{e}_2), \mathfrak{e}_2) \end{aligned}$$

Since r and r' are random numbers picked from Z_p^* uniformly, c_1, c_1', c_2, c_2', e_1 and e_2 could uniformly be any element in group $\mathbb{G}$. e_2' is randomly picked from $\mathbb{G}$. Although e_1' is determined by e_2', it can be any element in $\mathbb{G}$ since e_2' is random. So the dummy records and the real one are indistinguishable.

APP. The leakage of the matched items is inevitable if the search is performed by the CS. The key point is he CS should not learn any useful information from them. The CS nearly learn nothing from an independent search result. However, from the relationship between search results, it could infer the search pattern. We say APP is achieved if the CS is unable to infer the search pattern from the search result. Here we give the formal definition of APP.

Definition 2 ***(APP)***. *Let k be the security parameter of ObliviousDB over group $\mathbb{G}$. The APP game between a Probabilistic Polynomial-Time (PPA) adversary $\mathscr{A}$ and a challenger $\mathscr{B}$ is described as below*

- ***Setup***. *Same as the* ***Setup*** *in Def. 1.*
- ***Phase 1***. $\mathscr{A}$ *requests the search results from* $\mathscr{B}$ *for a set of queries* $\boldsymbol{Q} = \{q_1, ...q_t\}$. $\mathscr{B}$ *searches the result* SR_j *and sends it to* $\mathscr{A}$ *for each query* $q_j, j \in [1,t]$.
- ***Challenge***. $\mathscr{A}$ *sends two queries* q_0 *and* q_1*, which can belong to* $\boldsymbol{Q}$. $\mathscr{B}$ *flips a coin* $b \in \{0,1\}$*, returns* SR_b *to* $\mathscr{A}$.
- ***Phase 2***. *Same as phase 1.*
- ***Guess***. $\mathscr{A}$ *submits his guess* b' *of* b.

The advantage of $\mathscr{A}$ *in this APP game is defined as* $Adv_{\mathscr{A}}^{APP}(1^k)$. *We say ObliviousDB achieves APP, if for all PPA adversaries have at most negligible advantage in the above game:*

$$Adv_{\mathscr{A}}^{APP}(1^k) = Pr[b' = b] - 1/2 \leq negl(k) \tag{8}$$

where $negl(k)$ *denotes as a negligible function in k.*

In *ObliviousDB*, we achieve APP with two techniques. On the one hand, we change the index of each record in the result set by shuffling them with a number of unmatched records. On the other hand, we randomise the shuffled records to make them look different and untraceable. Consequently, the search results will be completely different for the same query.

Proof. Assume the search result for query Q is $SR = \{id_1, ..., id_m\}$, where $m \geq 1$. According to the oblivious algorithm, SR will shuffle with another m unmatched records $RR = \{id'_1, ..., id'_m\}$ randomly. After the shuffling, the record with index $id_i, i \in [1,m]$ could be one of the ever matched records or unmatched record with the same probability. That is, if $id_i, i \in [1,m]$ returns to another query, it could be same to Q with 50% probability. Furthermore, both SE and DE parts of all the shuffled records are re-randomised. $\mathscr{A}$ is unable to trace the record according to its ciphetext. Besides, we should note that since the dummy records are refreshed dynamically, even the search result for all the requested queries are different, $\mathscr{A}$ does not have any advantage to win the game.

OPP. We hide the operation pattern by extending all the types of query into two select queries, a set of update queries, a set of delete queries and one insert query, as described in Algorithm 1. From none of these queries, the CS could learn the operation pattern.

Only when the original query is select the first select query is real. If the original query is update or delete, we change it into 'select *' and keep their where-clause. To some extent, they are still real queries. For insert query, the fake query can be obtained by re-randomising an executed query. Hence, the fake query is indistinguishable from the real one.

The second select query is to fetch a number of random records. No matter what the original query is, this query will be always executed. Which and how many records should be fetched is not affected by the type of the original query. Therefore, from the second select query, the CS gets nothing about the type of the original query.

Similarly, a set of update queries will be always executed by the CS. The difference is when the original query is update, the search result of the first query will be updated into the new encrypted value. Otherwise, they are re-randomised. The re-randomised

value is indistinguishable from the original encrypted data. So this operation neither leaks the type of the original query.

The size of the delete operation set is depend on the number of dummy records. It could be empty or not whatever the original query is. When the original query is delete, the search result of the first query will be changed into dummy records. Since the dummy record is indistinguishable from the real one, the CS could not perceive if the original query is delete or not.

When the original query is insert, the real record will be inserted. Otherwise, a dummy record is inserted. Since the dummy record is indistinguishable from the real one, the real insert query is protected.

Forward and backward privacy. Forward and backward privacy should be guaranteed if a scheme aims to achieve SPP and APP. Assume the CS could run the executed queries on the updated database or run queries on deleted data. The same result set will be returned for the same query. Consequently, the CS can infer the search pattern.

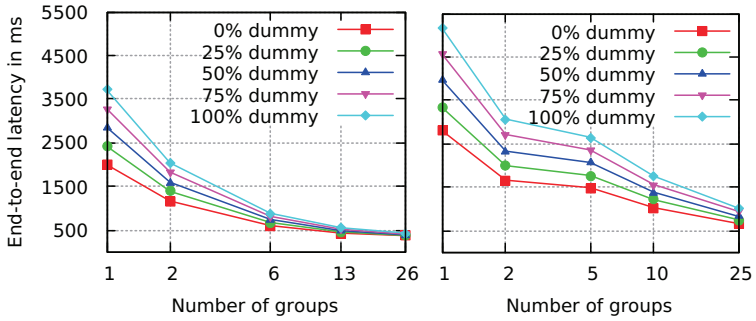
We use nonce to ensure the forward and backward privacy As shown in Equation 1, only when the query marked with a valid nonce, the record will be matched. In the oblivious algorithm, the nonces marked in the matched records will be replaced with a new one. Since the executed query is still marked with the old nonce, it will get nothing if the CS run this query arbitrarily. Similarly, assume the CS keeps a copy of the records before committing the update operation, it is unable to get the search result.

8 Performance Analysis

In this section, we evaluate the performance of *ObliviousDB*. We implemented the scheme in C using MIRACL 7.0.0 library 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 cluster with 64 Intel *i*5 3.3 GHz processor with 256 GB RAM, running Ubuntu 14.08 Linux system. For our experimentation, we considered the RAM-based database. In our testing scenario, all operations ran on one cluster and ignore the network latency occurring in a real deployment. In the following, all the results are averaged over 10 trials.

We first present the results of end-to-end latency measured at the DBU when performing a search operation on a database with $100,000$ real records and with a result set of $1,000$ real records. In this experiment, we want to measure the effect of the number of groups as well as the ratio between dummy and real records on latency.

The graphs in Figure 3 illustrate the latency in millisecond (ms). The graph in Figure 3(a) shows the results for the query 'select $*$ from *staff* where name=Alice'. The graph in Figure 3(b) provides the latency for performing a range query on a numerical attribute: 'select $*$ from *staff* where $20 < age < 31$'. In both graphs, the X-axis shows different group numbers: 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 was executed 5 times, each time changing the ratio (from 0% to 100%) between dummy and real records, represented by different lines in both graphs.

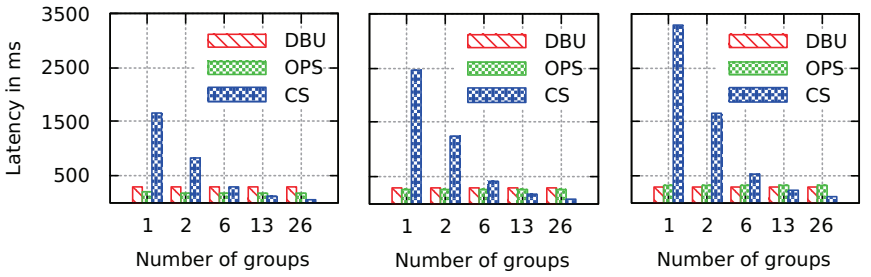


(a) Single keyword query (b) Range query

Fig. 3. End-to-end latency on the DBU for getting 1,000 real records from the database with 100,000 real records. The database size goes up to 200,000 with the increase of the ratio between dummy and real records. The tested single keyword and range queries are: 'select $*$ from *staff* where name=Alice' and 'select $*$ from *staff* where $20 < age < 31$', 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 means less records within a given group. This reduces searching time on the CS and in turn reduces the latency on the DBU.

On the other hand, increasing the ratio between dummy and real records degrades the performance. For both queries, for a given group size, the latency increases when we go from 0% dummy records (in other words, only real records) to a ratio of 100% (as many dummy records as real ones). This is explained mainly by two facts: i) with a higher ratio, the CS has to retrieve more records (including real and dummy ones), and ii) the OPS needs to filter out more records before sending the real records to the DBU.



(a) 0% dummy records (b) 50% dummy records (c) 100% dummy records

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.

It is worth mentioning here that, for the DBU latency experiment, we did not measure the time the OPS spends in executing the oblivious protocol. The main reason is

that the OPS will forward to the DBU the result sets and then initiates the oblivious protocol.

In the following, we want to provide some details on the time each entity, namely the DBU, the OPS and the CS, spend to execute a single keyword query. Figure 4 shows the graphs for the execution of the same select query. For each graph, the ratio is kept constant while we vary only the number of groups (shown on the X-axis). As we can see, the increasing of the number of groups affects mainly the performance on the CS while time taken by the DBU and the OPS remains constant. However, increasing the ratio between dummy and real records increases the latency on the OPS and the CS.

9 Conclusions and Future Work

In this work, we propose the first sub-linear searchable scheme for outsourced database, which achieves forward privacy and the privacy of search pattern, access pattern, size pattern and operation pattern without streaming the whole database and supports multi-user to carry out complex SQL-like queries on encrypted data.

As future work, we plan to optimize and parallelise the oblivious algorithm to improve the the performance of the OPS and the CS. Secondly, the size of each query is unprotected in our scheme, which is probably a potential useful leakage for the CS. Thus, we are going to take some measures to hide this kind of size pattern from the CS.

Acknowledgment

The authors would like to thank Steven D. Galbraith for his insightful comments in order to improve the quality of this work.

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. D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 668–679.
3. M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 644–655.
4. R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 85–100.
5. R. Ostrovsky, "Efficient computation on oblivious RAMs," in *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, ser. STOC '90. New York, NY, USA: ACM, 1990, pp. 514–523.
6. O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, May 1996.

7. E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple Oblivious RAM protocol," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 299–310.
8. B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, no. 6, pp. 965–981, November 1998.
9. P. Williams and R. Sion, "Usable PIR," in *NDSS*. The Internet Society, 2008.
10. E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage." *IACR Cryptology ePrint Archive*, vol. 2013, p. 832, 2013.
11. M. R. Asghar, G. Russello, B. Crispo, and M. Ion, "Supporting complex queries and access policies for multi-user encrypted databases," in *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '13. New York, NY, USA: ACM, 2013, pp. 77–88.
12. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 2006, pp. 79–88.
13. I. Hang, F. Kerschbaum, and E. Damiani, "ENKI: Access control for encrypted query processing," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 183–196.
14. L. Ferretti, F. Pierazzi, M. Colajanni, and M. Marchetti, "Scalable architecture for multi-user encrypted sql operations on cloud database services," *Cloud Computing, IEEE Transactions on*, vol. 2, no. 4, pp. 448–458, 2014.
15. M. I. Sarfraz, M. Nabeel, J. Cao, and E. Bertino, "DBMask: Fine-grained access control on encrypted relational databases," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '15. New York, NY, USA: ACM, 2015, pp. 1–11.
16. 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 Proceedings IEEE*. IEEE, 2014, pp. 226–234.
17. F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 310–320.
18. S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, A.-R. Sadeghi, Ed. Springer Berlin Heidelberg, 2013, vol. 7859, pp. 258–274.
19. N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 222–233, 2014.
20. M. Naveed, M. Prabhakaran, C. Gunter *et al.*, "Dynamic searchable encryption via blind storage," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 639–654.
21. 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*. IEEE, 2015, pp. 2092–2100.
22. M. Naveed, "The fallacy of composition of oblivious RAM and searchable encryption," Cryptology ePrint Archive, Report 2015/668, 2015.
23. D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*. IEEE, 2000, pp. 44–55.

24. M. Nabeel and E. Bertino, "Poster: towards attribute based group key management," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 821–824.
25. "Amazon S3," last accessed: January 28, 2016. [Online]. Available: https://aws.amazon.com/s3/
26. "Google Drive," last accessed: January 28, 2016. [Online]. Available: https://www.google.co.nz/drive/
27. 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/.
28. C. Dong, G. Russello, and N. Dulay, "Shared and searchable encrypted data for untrusted servers," *Journal of Computer Security*, vol. 19, no. 3, pp. 367–397, 2011.
29. D. Boneh and M. Franklin, "Identity-based encryption from the weil pairing," in *Advances in Cryptology - CRYPTO 2001*, ser. Lecture Notes in Computer Science, J. Kilian, Ed. Springer Berlin Heidelberg, 2001, vol. 2139, pp. 213–229.