Long White Cloud (LWC): A Practical and Privacy-Preserving Outsourced Database

Shujie Cui✉, Ming Zhang, Muhammad Rizwan Asghar, and Giovanni Russello

The University of Auckland, New Zealand
scui379@aucklanduni.ac.nz, {ming.zhang, r.asghar, g.russello}@auckland.ac.nz

Abstract. To fully benefit from a cloud storage approach, privacy in outsourced databases needs to be preserved in order to protect information about individuals and organisations from malicious cloud providers. As shown in recent studies [2, 11], encryption alone is insufficient to prevent a malicious cloud provider from analysing data access patterns and mounting statistical inference attacks on encrypted databases. In order to thwart such attacks, actions performed on outsourced databases need to be oblivious to cloud service providers. Approaches, such as Fully Homomorphic Encryption (FHE), Oblivious RAM (ORAM), or Secure Multi-Party Computation (SMC) have been proposed but they are still not practical. This paper investigates and proposes a practical privacy-preserving scheme, named *Long White Cloud* (*LWC*), for outsourced databases with a focus on providing security against statistical inferences. Performance is a key issue in the search and retrieval of encrypted databases. *LWC* supports logarithmic-time insert, search and delete queries executed by outsourced databases with minimised information leakage to curious cloud service providers. As a proof-of-concept, we have implemented *LWC* and compared it with a plaintext MySQL database: even with a database size of 10M records, our approach shows only a 10-time slowdown factor.

1 Introduction

Cloud computing provides organisations with virtually unlimited storage and computational power at attractive prices. One of the main challenges of outsourcing large databases to the cloud is to secure data from unauthorised access. A naive approach is to use standard encryption techniques for protecting the data. However, using this approach, no operations could be performed on encrypted databases.

Ideally, one would like to perform operations such as search, insert and update directly on encrypted databases without letting cloud providers learn information about both the query and the data stored in the database. For such scenarios, the Searchable Symmetric Encryption (SSE) scheme has been proposed, where symmetric keys are used for encrypting the data and queries. SSE was first introduced by Song *et al.* in [13] in 2000 and several more research efforts have been presented since then. Unfortunately, most of the existing SSE schemes suffer from one or more of the issues listed below.

- **Information Leakage** During data search, the correlation between the queries and the matched data is leaked to the cloud server. This correlation can be exploited by a determined attacker to break the encryption scheme as shown in recent studies [2, 8]. Approaches like Oblivious RAM (ORAM) [6, 12, 15] or Private Information Retrieval (PIR) [3, 16] could be used to minimise information leakage. 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.
- **Lack of Support for a Full-Fledged Multi-User Access** The vast majority of existing approaches support a very basic key distribution scheme, where all users share the same key. We refer to these as Single User (SU) schemes. Another common approach is to have a read-only key shared among all the users and one special key for inserting/updating data. We refer to these as Semi Fledged Multiple User (SFMU) schemes. In both cases, if a user misplaces a key or needs to have her access revoked, then a new key needs to be generated and the data requires re-encryption under the new key. Instead, in a Full-Fledged Multi-User (FFMU) scheme, any authorised user is able to read and write data from and to the database, respectively [1]. An FFMU scheme better fulfils needs of modern organisations, where users need to access and update data and are able to join and leave the organisation at any time without affecting rest of the users.

Our contribution is to propose a very efficient sub-linear Dynamic SSE (DSSE) scheme, named *Long White Cloud* (*LWC*), able to support a high throughput of queries while minimising information leakage. In terms of efficiency, our scheme is similar to Stefanov *et al.* [14] and Ishai *et al.* [7]. However, unlike their approach, our scheme is designed for large organisations, where users might join and leave at any time. Therefore, we want to support an FFMU scheme to simplify the user registration and revocation. At the same time, we want that each user is able to insert a new record, update existing records, and retrieve any data from the database.

The main idea in *LWC* is to use a hybrid private/public cloud approach. In such an approach, organisations maintain a private part of the infrastructure on their local premises while outsourcing the rest to a public cloud provider, such as Amazon or Google. In *LWC*, the private cloud is used to maintain data structures for speeding up the query processing and to perform operations in order to minimise information leakage. The public cloud instead is mainly used as an encrypted data store.

The rest of this paper is structured as follows: Section 2 gives an overview of the related work. Section 3 presents the system model, threat model, an abstract architecture of *LWC*. Section 4 describes key management and explains how we represent the data. Section 5 explains the database queries supported by the *LWC* scheme. Section 6 analyses security aspects of *LWC*. Section 7 evaluates the performance of *LWC* and compares it with a cleartext MySQL database. Section 8 concludes by highlighting the main contributions and results of the *LWC* scheme.

2 Related Work

Since the seminal work by Song *et al.* [13], many searchable schemes have been proposed and the research in this area has been extended in several directions. We focus mainly on three aspects of the encrypted search: key management, search efficiency and information leakage. A thorough and up-to-date survey and comparison of the current literature can be found in our paper [4]. We stress here that, with this work, we aim at proposing an FFMU scheme that minimises information leakage while supporting sub-linear search. Most importantly, our scheme is very efficient thus practical. Before discussing the rest, we first set the context and informally define the properties related to information leakage.

- **Search Pattern Privacy (SPP)** refers to the property where the cloud server is not able to distinguish if two (or more) queries are the same or not.
- **Access Pattern Privacy (APP)** means the cloud server is unable to learn if two (or more) real result sets overlap or not.
- **Size Pattern Privacy (SzPP)** is achieved if the cloud server is unable to learn the size of returned (real) records.
- **Operation Pattern Privacy (OPP)** ensures the cloud server cannot tell if the executed query is a select, update, delete, or insert.

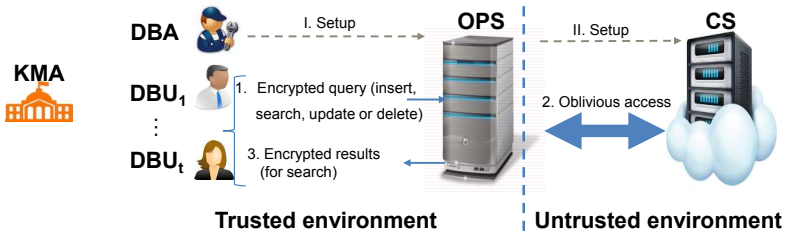

Fig. 1. System entities and their interactions in *LWC*: A DBA is responsible for running the setup (Step I then Step II). A DBU encrypts the query, which could be insert, select, update or delete. A DBU sends the encrypted query to the OPS (Step 1). The OPS communicates with the CS in an oblivious manner (Step 2). The OPS returns encrypted results (in case of select) back to the DBU (Step 3). The DBU can decrypt and get access to the requested data.

3 Overview of LWC

3.1 System Model

The system involves the following entities, also shown in Figure 1:

- A **DataBase User (DBU)** represents an authorised user who is allowed to access, retrieve and update information in the outsourced database.

- A **DataBase Administrator (DBA)** is responsible for the setup and management of the outsourced database. It is also responsible for the management of DBUs. A DBA is considered to act on behalf of the organisation that outsources the database.
- The **Operations Proxy Server (OPS)** is under the direct control of the DBA. With the help of the OPS, a DBA can grant, revoke and regulate access to the outsourced database. For database operations, the DBU interacts with the OPS to run all kinds of queries including insert, select, update and delete.
- The **Cloud Server (CS)** is part of the public infrastructure that provides cloud storage services. The CS is usually hosted and operated by a third party service with full access control over its cloud storage. It is expected to provide only two main operations: read and write.
- The **Key Management Authority (KMA)** is responsible for issuing encryption keys. Once the scheme is initialised, the KMA supplies encryption keys to new DBUs and the OPS. Typically, the KMA is under the complete control of the DBA and is online only when the DBA requests encryption keys for new DBUs. In other words, it can go offline after a DBU gets registered.

3.2 Threat Model

The KMA is assumed to be a fully trusted entity. The DBUs are responsible for securely keeping their keys (and decrypted information). We assume that the OPS is deployed on the private cloud; whereas, the CS is hosted on a public cloud infrastructure. The OPS is assumed to be trusted but may be potentially vulnerable to external attacks since it communicates with the external world. As the vast majority of the existing SSE schemes consider, including all the works listed in [4], the CS is assumed to be honest-but-curious. That is, the CS follows the protocol correctly, but may be interested to read the information or to try and discover statistical inferences on its database access and operations. DBUs may collude among themselves but they do not learn anything more than what they would learn individually. A DBU colluding with the CS reveals no useful information. An active or revoked DBU is unable to decrypt communication between other DBUs and the OPS.

3.3 System Interactions

LWC aims at minimising potential information leakage while supporting sublinear search in a multi-user setting. The interactions between the entities are shown in Figure 1. The following sequence of steps takes place to initialise and run the system. The DBA sets up the OPS (Step I) and then prepares the database on the CS (Step II). The DBA brings the KMA online to distribute encryption keys between participating DBUs and the OPS. Using her key, a DBU encrypts and issues queries including insert, search, update and delete.

As shown in Figure 1, a query is processed in two main phases. In the first phase, the DBU sends the encrypted query to the OPS (Step 1) and then the

OPS consults the local lookup table for finding locations of the records stored on the CS. In the second phase, the OPS uses these locations to fetch the data from the CS. To avoid any potential information leakage, the OPS queries the requested information in an oblivious manner (Step 2). The oblivious access hides from the CS the actual query issued by the DBU or the result set returned to the DBU. If the query is select, the OPS sends the encrypted result back to the DBU (Step 3). Finally, the DBU decrypts the encrypted results using her private key.

4 Key Management and Data Representation

4.1 Key Management

LWC supports the FFMU access. That is, if the DBU key is stolen or compromised, the system can still work without requiring re-encryption of the data with new keys and re-distribution of the new keys to the authorised users. In *LWC*, for each DBU, the KMA generates two keys (K_{DBU}, K_U). The KMA forwards both (K_{DBU}, K_U) keys to the DBU but only K_{DBU} to the OPS. The key K_U is a shared key across all DBUs and the key K_{DBU} is a DBU specific key that is shared with the OPS. The OPS stores all the DBU specific keys in a key store. Using both K_{DBU} and K_U, the DBU can issue queries and decrypt the search result. For revoking a DBU, the DBA instructs the OPS to remove the corresponding K_{DBU} key from its key store. Without loss of generality, instead of using K_{DBU}, a secure channel (say using SSL) can be established to protect communication between the DBU and the OPS.

4.2 Data Structure for the CS

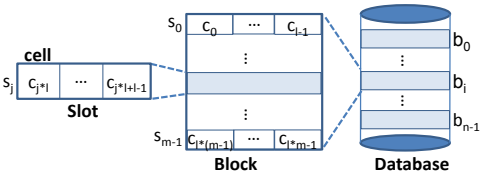

Fig. 2. Database on the CS: A database consists of a set of n blocks $\{b_0, \ldots, b_{n-1}\}$. Each block contains m slots $\{s_0, \ldots, s_{m-1}\}$, where a slot stores a single record. Each slot s_j has l cells.

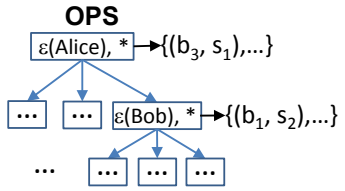

Fig. 3. A sample of B^+tree with 3 branches and 3 layers. The entry on each node is an encrypted keyword, and a pointer points to a list of (b_i, s_j) indicating the record store location on the CS.

The data stored on the CS is organised with three different data units, as shown in Figure 2. As a whole, the database is a set of n blocks $\{b_0, \ldots, b_{n-1}\}$.

The OPS uploads or downloads data to or from the CS block by block. Each block consists of m slots $\{s_0, \ldots, s_{m-1}\}$. Each slot stores a single record. A tuple of block number and slot number (b_i, s_j) uniquely identifies a stored record in the database. Specifically, we define a slot as *empty* if it holds no record. An empty slot may hold *null* or a random bit string. On the contrary, a slot is *full* if it is occupied by a DBU inserted record. Furthermore, each slot contains l cells. In other words, each encrypted record in a slot is divided into l data cells. Overall, each block is a set of $m*l$ data cells $\{c_0, \ldots, c_{m*l-1}\}$. Data re-encryption in *LWC* is implemented by permuting data cells (see Section 5.3). Assume each data cell is w-bit long, there are 2^w possible cell values, and each encrypted record is fixed to $w * l$ bits.

4.3 Data Structure for the OPS

In *LWC*, the search operation is performed on the OPS. Considering the OPS is trusted, any data structures that support sub-linear search, such as inverted index technique proposed in [5] and the red-black tree introduced in [10], could be used. In this work, we use a series of B^+trees [9] for managing indexing information on the OPS, which aims at supporting sub-linear search. Each field in a record is stored on a different B^+tree (see Figure 3) and hence it provides a simple and efficient method to query for keywords across different fields. Each entry in the tree is a tuple of an encrypted keyword and a list of (b_i, s_j) indicating the locations of the records containing this keyword on the CS. We can notice that the unique keyword number in each field determines the size of the corresponding tree, which is much less than the number of records on the CS. To further reduce the storage overhead, the OPS could hash the encrypted keyword first before inserting it into the tree.

The OPS also stores a list of flags to mark if each slot is full or empty. Using this information, the OPS can pick an empty slot and store the inserted record.

5 Query execution

In *LWC*, a DBU can issue queries for inserting, updating, searching, and deleting records. A query may include simple keywords or conjunctions/disjunctions of conditions, like "select * from Staff WHERE name = Alice AND age = 25". As mentioned in Section 3.3, the queries are mainly processed in two phases by *LWC*.

The first phase aims to get the locations of records (for a select query) or an empty slot (for an insert query) on the CS. Technically, it involves 4 steps on the DBU and the OPS. The detail of each step is shown in Figure 4.

5.1 Encryption on the DBU

The first two steps, which are part the first phase, are completed by the DBU. The system components, running on the DBU end, include the encryption and

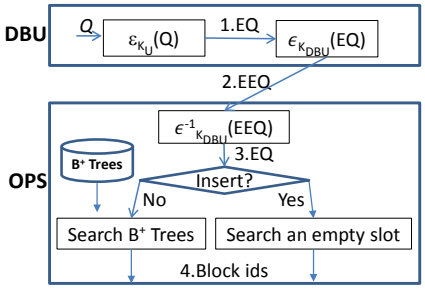

Fig. 4. Query execution: A DBU encrypts a query Q twice with K_U and then K_{DBU}. The encrypted query EEQ is sent to the OPS. The OPS decrypts it with K_{DBU} to get EQ, and then look ups an empty slot for inserting a record or searches B^+ trees according to the type of the query.

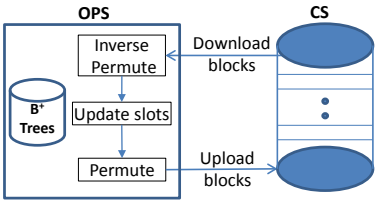

Fig. 5. Oblivious access between the OPS and the CS: The OPS first downloads a set of blocks, which contains the required slots, and then performs three steps (inverse permutation, updating empty slots, and permutation) after which the blocks are uploaded back to the CS.

decryption utilities. Each query is processed in two rounds of encryptions on the DBU using two different functions $\varepsilon : Q \rightarrow \varepsilon_{K_U}(Q)$ and $\epsilon : EQ \rightarrow \epsilon_{K_{DBU}}(EQ)$. ε is a deterministic symmetric encryption to make the encrypted data searchable and retrievable. In the first round of encryption, the query Q is encrypted using K_U, a key shared among all DBUs. On one hand, it ensures the data could be accessed by all the DBUs. On the other hand, it also means the encrypted query EQ and search pattern are not protected from other DBUs. To address this issue, we introduce the second round of encryption, where EQ gets re-encrypted under K_{DBU}. The second round of encryption ϵ is semantically secure. Because K_{DBU} is unknown to other DBUs, they are unable to learn EQ and search pattern. Without loss of generality, we could issue a password to each DBU for authentication and then use SSL/TLS to establish a secure channel between the DBU and the OPS to deliver EQ. ε^{-1} and ϵ^{-1} are the their corresponding decryption functions. Note that the first round of encryption is only performed over the keywords in Q. That is, the logical conditions and operators in EQ are in cleartext (for the OPS). For instance, "select * from Staff WHERE $\varepsilon_{K_U}(name) = \varepsilon_{K_U}(Alice)$ AND $\varepsilon_{K_U}(age) = \varepsilon_{K_U}(25)$". However, in the second round of encryption, EQ is encrypted as a whole, which means all the information in EQ is protected.

5.2 Index Search on the OPS

As shown in Figure 4, the last two steps of a query execution are performed by the OPS. The OPS is responsible for parsing the query, searching for the locations of the corresponding records on the CS database and fetching those records. In case of a select query, the OPS returns those records to the DBU. Parsing EEQ on the OPS enables *LWC* to execute any kind of query, which could

involve conjunctions, disjunctions and other logical operators for a multi-keyword search, update and delete queries. As shown in Figure 4, the components on the OPS includes the ϵ^{-1} for decrypting EEQ, B^+ trees for managing indexing information to enable sub-linear search.

In the third step, the OPS removes the outer layer encryption of EEQ to get the executable query EQ by running the inverse stream cipher function $\epsilon^{-1} : EEQ \rightarrow EQ$.

For an insert query, the next step is to check the flags list and find an empty location (b_i, s_j) to hold the record. Next, the selected location (b_i, s_j) will be added to the corresponding entry in the B^+ tree for each field (see Figure 3). If the related entry is already in the tree, we just add the (b_i, s_j) to its list. Otherwise, a new entry will be created. Note that the keywords in trees are encrypted, which ensures data confidentiality even if the OPS gets compromised.

In case of search, delete and update queries, the fourth step is to search the B^+ trees and find out which records on the CS match the query. The B^+tree search incurs $O(\log_b N)$, where N denotes the total number of nodes in the tree and b represents the branching factor of the tree. For efficiency reasons, we suggest having a separate tree for each field in the table. Recall the sample query, two predicates "$\varepsilon_{K_U}(name) = \varepsilon K_U(Alice)$" and "$\varepsilon(age)_{K_U} = \varepsilon_{K_U}(25)$" are searched over two trees separately. Once the location tuple list is searched out for each separate keyword, the final search result is the combination of them according to the logical operator between keywords, which is in cleartext on the OPS.

With these locations, the second phase is to execute (*i.e.,* insert, select, update or delete) on the CS database obliviously, which is explained in Section 5.3.

5.3 Oblivious Access

In *LWC*, we aim to protect operation pattern, size pattern, access pattern and search pattern from the CS. The OPS gets the location tuples for the query. If the OPS sends the tuples to the CS directly, the CS could learn these patterns easily. The data should be accessed in an oblivious manner. The detail of the oblivious access between the OPS and the CS is shown in Figure 5. The OPS performs the following three steps: inverse permutation, update slots and permute. The detail of each step is given below.

Recall that all the data stored on the CS is encrypted using K_U, which is a key shared among all the DBUs. This implies that K_U is still known to DBUs that have been revoked. A revoked DBU can collude with the CS to decrypt the data stored on the CS. To address the problem, instead of re-encrypting the data, we propose a more efficient approach. The idea is to shuffle the data cells between different records in each block with a pseudo-random permutation π. That is, all the data cells in each block are permuted in an invertible way, which is only known to the OPS. Consequently, the DBU is unable to decrypt the data without the assistance of the OPS. In each permuted block, the data in each slot is no longer a complete record. The permutation seed for each block is kept by

the OPS. Note that to make the permutation invertible for the CS, we should set $m * l \gg 2^w$, where $m * l$ is the number of data cells in a block and w is bit length of a data cell.

Before any operation, the OPS has to download the matched records from the CS. If the matched slots are distributed in k blocks, the CS will download γ blocks, where $k < \gamma$. k of them are the matched blocks, the rest $\gamma - k$ blocks are picked randomly. Note that all blocks from the database must be picked through a single request from the OPS. After receiving γ blocks, the OPS first performs π^{-1} over the $m * l$ data cells for each block to recover the data order. In the second step, actions (described below) are taken based on the query type:

- If EQ is a select query, all the matched slots will be sent to the DBU.
- If EQ is a delete query, the OPS just changes flags of these matched slots to mark them empty and does nothing over the records in each block.
- If EQ is an update query, the matched records will be updated with the new values.
- If EQ is an insert query, the new records will be inserted into the selected slot.

After that, the OPS updates or fills a number of empty slots with random bit strings for each block. In the third step, all the data cells in each block are permuted again with a new seed. Finally, the γ updated blocks are written back to the CS. The security of *LWC* is analysed in Section 6.

5.4 Data Decryption

In case of a select query, there is one more phase between the OPS and the DBU, *i.e.,* the decryption of the result. Once the records in each block are retrieved, the OPS will extract the records satisfying the query. Before sending them to the DBU, the OPS first encrypts it with ϵ. Since the encryption key K_{DBU} is unique for each DBU, the search result could only be decrypted by the DBU who issued the query. Considering ϵ is semantically secure, the access pattern can not be inferred from the interaction between the DBU and the OPS. To hide the size pattern, the OPS could add a set of dummy data into the result. On the DBU, two rounds of decryptions will be performed to get the cleartext records.

6 Security Analysis

In this section, we analyse how the oblivious access described in Section 5.3 could protect size pattern, access pattern and operation pattern from the CS.

Recall that we say SzPP is achieved if the CS cannot learn how many records are matched for each query. *LWC* achieves SzPP with two techniques. First, the OPS downloads data from the CS block by block, rather than slot by slot. It is unknown to the CS how many slots are matched with the query in each block. Second, γ blocks are downloaded by the OPS for each access. Here γ is a random number determined by load on the OPS, but independent from the real matched

blocks. Note that, for the same query, if load on the OPS is different, the value of γ would be different. In other words, from the number of accessed blocks, the CS is unable to infer anything about search pattern.

Similarly, access pattern is also protected at two different levels. Since the data is downloaded block by block, it is unknown to the CS which slots in each block are matched with the query. If only the matched blocks are downloaded by the OPS, given the number of matched records is unknown to the CS, in the view of the CS, each record in a block could be a matched one or not with 50% probability. Furthermore, matched blocks are also protected by random blocks. When γ blocks are downloaded for a query, there are 2^γ possible block access patterns. The probability that the CS could guess the real block access pattern successfully is $\frac{1}{2^\gamma}$. Moreover, from the relationship between downloaded blocks, the CS is unable to infer search pattern.

Theorem 1. *If the $\gamma - k$ blocks are picked randomly, LWC partially achieves SPP.*

Proof. (sketch) Since *EQ* is never sent to the CS, the CS could only infer search pattern from the relationship between size patterns and the relationship between accessed blocks. If access patterns and size patterns of two queries are the same, there is a high probability that the queries are the same, and vice versa. As we mentioned above, search pattern cannot be inferred from size pattern, since they are always variable no matter whether the queries are the same or not. In *LWC*, due to the random blocks, access patterns are also always variable for all queries. More specifically, for the same queries, $\gamma - k$ random blocks could make their block access patterns different. However, for different queries, $\gamma - k$ random blocks may cause an overlap between their block access patterns. In the view of the CS, each block could be the matched one or random with the same probability. It has no advantage to infer search pattern from the overlap between block access patterns. There are 2^m possible slot access patterns in each block. The probability that queries have the same slot access patterns in one block is $\frac{1}{2^{2m}}$. However, the CS could learn the two queries are different when the two accessed block sets are totally different. To avoid such leakage, the OPS could download the whole database each time, but it is costly.

Theorem 2. *If the encrypted records are indistinguishable from random bit strings and if π is a pseudo-random permutation, LWC achieves OPP.*

Proof. (sketch) To protect operation pattern, the OPS always performs the same operations. Specifically, no matter what type the query is, the OPS updates or fills a number of the empty slots with random bit strings. If the query is insert, the selected empty slot will be filled with the new record. If the query is update, the matched slots will be updated with new values. But, in the view of the CS, there are always some slots that are updated or filled in each block for each access. Since the encrypted record is indistinguishable from a random bit string, the CS is unable to learn if the slot is filled with a record or random bit string, making insert and update queries indistinguishable. If the query is delete, the records

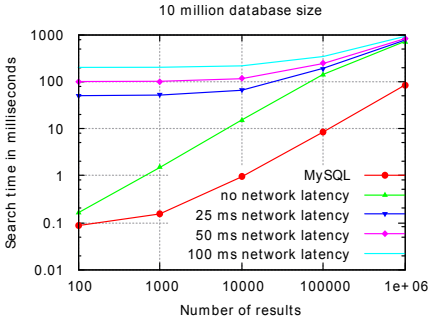

Fig. 6. End-to-end search time in *LWC*.

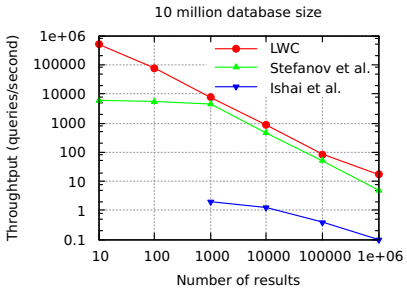

Fig. 7. Query throughput comparisons for database schemes.

are removed by changing the flags, but the values in each block are unchanged like the select query. Moreover, the $m * l$ slots in each block are permuted again with a new seed before uploading them to the CS. Consequently, which slots and how many slots are updated or filled, and if slots are updated partially or completely are also protected from the CS.

The problem is, for select and delete queries, if all the slots in accessed blocks are filled with DBU inserted records, the OPS would not be able to insert or update random bit strings. Recall that there are 2^w different cell values in the system. Even if all the data cells will be permuted again, the number of each unique cell value in each block remains unchanged for select and delete operations. With the frequency information of each data cell value, the CS could distinguish update from select and delete operations. To solve this problem, we set a block is *full* when θ out of m slots are occupied by records and do not insert any records into it anymore, where $\theta < m$. That is, each block always has at least $m - \theta$ empty slots to fill random bit strings. Therefore, whatever the query type is, the frequency of data cell values is changed all the time. In practice, we can set $\theta = \lceil 2^w/l \rceil$. In this case, the frequency of each data cell value could be changed.

Although the OPS is trusted, considering the data stored in the B^+ trees are encrypted, the queries received from the DBUs are encrypted as well. Thus, compromising the OPS will not put any data at risk. However, SPP, APP and OPP might not be guaranteed, since the data and queries are encrypted using a deterministic algorithm. We aim at further investigating these issues in our future work.

7 Experimental Evaluation

In this section, we analyse performance of *LWC*. For all the experiments, we used a PC powered by Intel i5-4670 3.40 GHz processor and 8 GB of RAM using Linux Ubuntu 15.04. Note that we have chosen a very basic PC setup to show that *LWC* can achieve high performance even when deployed on cheap hardware.

The prototype is programmed in C and is compiled using GCC version 4.9.2. No parallel operations or hyper-threading were implemented. All data structures are stored in RAM. For the experiment, we set the OPS to pick up to $2 \cdot k$ blocks from the CS for each query, where k is the number of blocks needed to execute the query. We fixed the maximum size of a record inserted at 128 bytes the number of cells in a slot was set at 256 cells.

The experiments presented in the following were set up as follows. All the entities (the DBU, the OPS and the CS) were executed on the same machine. However, where required, we used a simulated round-trip network latency for the links between each entity. In the following, the queries with a single predicate were executed on a database with 10 million records and all the results were averaged over 10 trials.

First, we measured the end-to-end time for a DBU to perform a search operation varying the number of the returned records between 100 to 1 million. The results are shown in Figure 6 in milliseconds (ms). These measurements include also the time on the DBU to encrypt the query and decrypt all the returned records. As a baseline, we executed the same experiments using a plaintext MySQL database where the client and the database were deployed on the same machine without any network latency. With no network latency, *LWC* end-to-end search time on an average was between 2 (when 100 records were returned) and 10 times slower than the plaintext MySQL. Given that in *LWC* the DBU and the CS do not interact directly but through the OPS, we have performed the same experiments but introducing a simulated round trip network latency of 25 ms, 50 ms and 100 ms. As we can see in Figure 6, the effect of network latency on search time rapidly reduces when the result size increases. In any case, with a result size of 1 million records, the end-to-end search time is under 1 second even when 100 ms network latency is introduced.

To investigate the penalty introduced by the OPS, we have compared the query throughput of *LWC* with schemes proposed by Stefanov *et al.* [14] and Ishai *et al.* [7]. To the best of our knowledge, both schemes are currently the best in term of performance when compared with other SSE schemes. The results of this comparison are presented in Figure 7. Note that, at the time of the writing of this paper, we could not get access to their implementations. The graphs for Stefanov's and Ishai's schemes in Figure 7 are plotted using the data presented in their respective papers. The comparison shown in Figure 7 is an approximation for two main reasons: (1) in Ishai's paper, there are no throughput values for result sizes smaller than 1K; (2) while Ishai's scheme has been tested on a hardware configuration very similar to ours, results by Stefanov *et al.* were collected on a top of the line hardware configuration with a maximum degree of parallelism at 32. Therefore, the results presented in Figure 7 for Stefanov's scheme have been normalised as executed on a single core.

The comparison in Figure 7 shows that *LWC* on average has a throughput about twice that of Stefanov's when no parallelism is used. For instance at 10,000 results, the throughput of *LWC* is 852 queries per second, where Stefanov's scheme achieves 450 queries per second. Though the comparisons are on the log

scale, the results on individual data points indicate that *LWC* has a significantly higher throughput. *LWC* is also seen to easily outperform the results in Ishai's scheme.

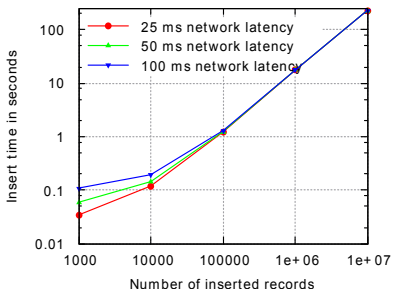

Fig. 8. Time taken for executing an insert query in *LWC*.

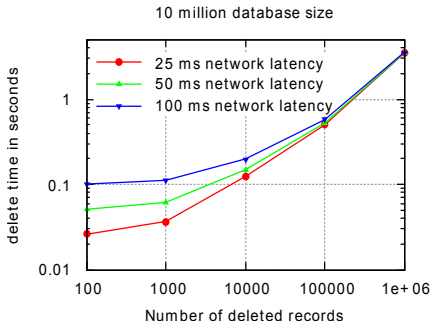

Fig. 9. Time taken for executing a delete query in *LWC*.

We also provide the latency results for insert, update and delete operations with simulated end-to-end network latencies.

Figure 8 plots the times for an insert query including network latency in seconds. It is observed that *LWC* takes around over 0.1 seconds to insert 1K records and up to 100 seconds for inserting 10M records. We also observe that the effect of network latency diminishes when a large number of records are inserted.

Figure 10 plots the time for an update query with network latency. We can observe that *LWC* can update 1M records in just under 4 seconds. Again, the network latency effect diminishes when a large number of records are updated.

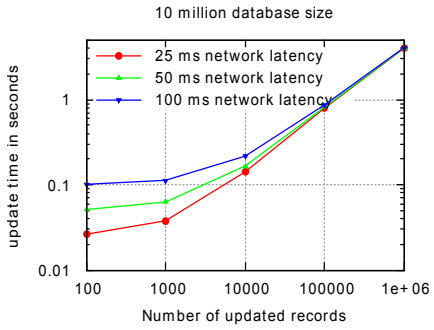

Fig. 10. Time taken for executing an update query in *LWC*.

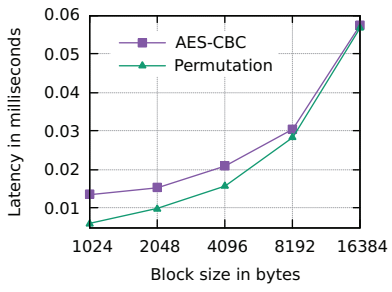

Fig. 11. The performance comparison between AES-CBC and permutation operations.

Finally, Figure 9 plots the time for a delete query including network latency. LWC can delete 1M records in under 4 seconds. Note that, in *LWC*, the steps executed for the delete operation are similar to the update. The only difference is that in a delete operation there is not replacing of the old with new value like in an update.

To achieve OPP and resist against the collusion between the DBU and CS, the OPS could also encrypt each block with other cryptographic primitives, like AES-CBC, rather than permuting data cells. However, permuting data cells is more efficient than re-encryption. We did another test and compared the performance of permutation with AES-CBC encryption. AES-CBC with 256-bit key implemented in MIRACL 7.00 C library was used for the test. For permutation, we set the size of each data cell to 1 byte. Before getting all the required blocks, the OPS could first pre-generate new seeds for accessed blocks and pre-compute the new orders. As shown in Figure 11, permutation is more efficient than AES-CBC when the block size is between 512 and 16384 bytes. When the block size is greater than 2^{16} bytes, the data cell can be set to 2 bytes, which will make the permutation more efficient.

We can conclude our analysis by discovering that even though *LWC* requires the OPS to perform most of the computation, its centralised nature does not degrade performance of the system. *LWC* achieves a high throughput performance and even with network latency simulations, returning 1 million records from a database of 10 million records takes less than a second including the time on the DBU to decrypt the returned results. When compared with a plaintext database, *LWC* results 10 times slower but achieves a high level of privacy. Finally, our comparison with other similar works, with all its current limitations, shows that *LWC* has a higher throughput for any result size while it still provides a very flexible key management scheme not supported by Stefanov's and Ishai's schemes. Moreover, *LWC* supports more privacy properties when compared with existing schemes.

8 Conclusion and Future Work

In this paper, we proposed *LWC*, a dynamic searchable encrypted scheme for hybrid outsourced databases with a full-fledged multi-user key management. *LWC* is a sub-linear scheme that does not leak information on its search pattern, access pattern, size pattern and operation pattern. The experimental results indicate that *LWC* is able to achieve high performance without requiring top of the line hardware. We have compared *LWC* with two relevant high-performance approaches: Stefanov *et al.* [14] and Ishai *et al.* [7]. *LWC* outperforms both while providing a higher level of privacy and a more flexible key management.

In our future work, we plan to extend the capabilities of the B^+ tree implementation to support range queries. It also remains to be explored how to move most of the operations to the untrusted cloud server, but without compromising on security.

References

1. M. R. Asghar, G. Russello, B. Crispo, and M. Ion. Supporting complex queries and access policies for multi-user encrypted databases. In A. Juels and B. Parno, editors, *CCSW 2013*, pages 77–88. ACM, 2013.
2. D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In I. Ray, N. Li, and C. Kruegel, editors, *SIGSAC 2015*, pages 668–679. ACM, 2015.
3. B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.
4. S. Cui, M. R. Asghar, S. Galbraith, and G. Russello. Secure and practical searchable encryption: A position paper. In *ACISP 2017*, Lecture Notes in Computer Science. Springer, 2017.
5. R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In A. Juels, R. N. Wright, and S. D. C. di Vimercati, editors, *CCS 2006*, pages 79–88. ACM, 2006.
6. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.
7. Y. Ishai, E. Kushilevitz, S. Lu, and R. Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In K. Sako, editor, *CT-RSA 2016*, volume 9610 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 2016.
8. M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*. The Internet Society, 2012.
9. J. Jannink. Implementing deletion in B+-trees. *SIGMOD Rec.*, pages 33–38, 1995.
10. S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In A. Sadeghi, editor, *FC 2013*, volume 7859 of *Lecture Notes in Computer Science*, pages 258–274. Springer, 2013.
11. M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In I. Ray, N. Li, and C. Kruegel, editors, *SIGSAC 2015*, pages 644–655. ACM, 2015.
12. R. Ostrovsky. Efficient computation on oblivious rams. In H. Ortiz, editor, *STOC 1990*, pages 514–523. ACM, 1990.
13. D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *S&P 2000*, pages 44–55. IEEE Computer Society, 2000.
14. E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *NDSS 2013*, volume 71, pages 72–75, 2014.
15. E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In A. Sadeghi, V. D. Gligor, and M. Yung, editors, *SIGSAC 2013*, pages 299–310. ACM, 2013.
16. P. Williams and R. Sion. Usable PIR. In *NDSS 2008*. The Internet Society, 2008.