Implementation of a Simple Client-Server Private Database Using Homomorphic Encryption

Ouyang Danwen, Tran Tien Dat

April 22, 2018

1 Introduction

Homomorphic encryption (HE) allows complex mathematical operations on encrypted data directly without compromising the encryption. With HE, it is thus possible to outsource the heavy computations to a database server securely as no decryption is necessary at the server side, thus protecting the privacy of the database directly. In fact, in the context of a client server model, the server is normally assumed to have more hardware resources, thus being able to handle more complex and heavier computations on the data. By making use of HE, we can therefore achieve better utilization of the computing resources by transferring more hardware demanding tasks from the client to the server.

Various methods of constructing such a private database query (PDQ) based on HE have been discussed in the literature, but most are still far from being practical, as the topic of research is very recent. Among these propositions, [1] offers a promising PDQ construction with better security and efficiency. This PDQ construction can already process and respond to conjunction, disjunction and threshold queries with equality conditions. Our main work was: to build a client-server prototype of this PDQ construction, optimize its performance by parallelization, as well as developing new functionalities such as adding new database entries, implementing limit and offset of query results, performing 0/1 sorting of query results, implementing and parallelizing a less than test.

2 Preliminaries

2.1 Homomorphic Encryption

A fully homomorphic encryption (FHE) consists of the four algorithms, $KeyGen$, Enc, Dec, and $Eval$:

- $KeyGen(\lambda) \rightarrow (pk, ek, sk)$: It takes a security parameter λ as input and outputs a public key pk, an evaluation key ek, and a secret key sk. We assume that pk, sk, and ek each include the information of both the plaintext space P and ciphertext space C.

- $Enc(pk, m) \rightarrow \overline{m}$: Given the public key pk and a plaintext message $m \in P$, it outputs a ciphertext. $\overline{m}$.

- $Dec(sk, \overline{m}) \rightarrow m^* / \perp$: Given the secret key sk and ciphertext $\overline{m}$, it outputs a message $m^* \in P$ or $\perp$.

- $Eval(ek, \varphi, \overline{m}_1, ..., \overline{m}_n) \rightarrow \overline{m}_\varphi$: It takes the evaluation key ek, a function φ: $P^n \rightarrow P$, and a set of n ciphertexts $\overline{m}_1, ..., \overline{m}_n$ as inputs and outputs a ciphertext $\overline{m}_\varphi$ which decrypts to m_φ, the result of applying the function φ on the set of plaintexts.

Here, arbitrary functions φ are allowed to be evaluated in the $Eval$ algorithm of pure FHE scheme.

A leveled FHE (LFHE) scheme can, however, evaluate L-level arithmetic circuits according to a predetermined L in contrast of pure FHE due to the fact that arithmetic circuit evaluations on the ciphertext incur noises and will thus compromise the encryption (i.e. no longer decrypts correctly) when the preset level is exceeded. Generally, multiplications incur significantly more noises than additions. Therefore, in this paperm L refers primarily to the multiplicative depth of the arithmetic circuit that can be evaluated on.

To achieve FHE from LFHE, a different operation named bootstrapping is necessary to reset the noise and thus refresh the ciphertext through re-encryption. As a result, the original evaluation circuit must be augmented to be able to evaluate the scheme's own decryption circuit. The operation is hence considerably costly. To avoid the need for bootstrapping, minimizing L is crucial in our implementation to achieve efficiency.

2.2 Private Database Query processing

In the proposed PDQ construction [1], a database $D = (\alpha_1, ..., \alpha_n)$ consists of n tuples, where each tuple α_i is an ordered list of τ attributes $(\alpha_{i,1}, ..., \alpha_{i,\tau})$. Each attribute $\alpha_{i,j}$ is an integer encoded as a finite field element in the plaintext space P, i.e. a polynomial according to its binary representation. For example, 18 in binary is 10010, which becomes $x^4 + 0x^3 + 0x^2 + x + 0$ or just $x^4 + x$. Initially, the client will encrypt the database $\{\alpha_{i,j}\}$ into $\{\overline{\alpha_{i,j}}\}$ and send them over to the server for storage.

Subsequently, the client can send queries to the server and the server will send back the results. Three types of queries are supported:

- Conjunction query: e.g. attribute 1 = 10 AND attribute 7 = 7332

- Disjunction query: e.g. attribute 1 = 10 OR attribute 7 = 7332

- Threshold query: e.g. at least 3 conditions in the following list are satisfied: attribute 1 = 10, attribute 2 = 282, attribute 4 = 837, attribute 7 = 55, attribute 10 = 70

The advantage of using HE with the proposed PDQ construction is that the type of query, as well as the value in the equality conditions are encrypted. Thus the server does not know anything about the database, or the nature of the query it is processing, thereby preserving the privacy of the data.

The detailed explanation of this PDQ construction can be found in the cited paper. Here, we summarize the computations that the client and server need to carry out.

Query Type	b_j	c_j	d_i
Conjunction	0	1	0
Disjunction	1	1	1
Threshold	1	$1+t$	0

Table 1: Values of constants for different query types.

2.2.1 Client side

- Compute $\{b_j\}$, $\{c_j\}$, $\{d_i\}$ where $i = 1, ..., n$, $j = 1, ..., \tau$ depending on the query type according to Table 1.
- Compute polynomial $g(x)$. For conjunction and disjunction query, $g(x) = x$. For threshold query with threshold T, $g(x)$ is a polynomial such that $g(t^\kappa) = \begin{cases} 1 & \text{if } T \leq \kappa < \tau \\ 0 & \text{if } 0 \leq \tau < T \end{cases}$.
- Encrypt $\{a_j\}$, $\{b_j\}$, $\{c_j\}$, $\{d_i\}$ and $g(x)$ and send them to the server. (a_j is the value in the equality condition, i.e. attribute j = a_j).

2.2.2 Server side

- Compute $\overline{\beta}_{ij} = b_j + EQTest(\overline{\alpha}_{ij}, \overline{a}_j) \cdot \overline{c}_j$ for $i = 1, ..., n$ and $j = 1, ..., \tau$.
- Compute $\overline{\zeta}_i = \overline{d}_i + \prod_{j \in [\tau]} \overline{\beta}_{ij}$.
- Compute $\overline{\gamma}_{ij} = \overline{g}(\overline{\zeta}_i) \cdot \overline{\alpha}_{ij}$ for all $i \in [n]$ and j refers to the attribute to be returned.
- Send $\{\overline{\gamma}_1, ..., \overline{\gamma}_n\}$ to the client.

3 Our work

3.1 Implement client-server model

We created a library called HomomorphicDB which provides the functionalities of a simple private database. An application can use the classes `Server` and `Client` to access these functionalities. In addition, we also created several executables to provide a command line interface.

- Start the server: `./server-cli <port> [ optional parameters ]`

  `optional parameters` have the form `attr1=val1 attr2=val2 ...`

  `pubKey` [default=pubKey.txt] refers to the file where the public key is saved.

  `database` [default=encDB.txt] refers to the file where the encrypted database is saved. If the files exist, the server will load the public key and encrypted database from them. Otherwise, the server's public key and encrypted database is not initialized, and they will be saved to the specified location once the server receive them from the client.

  E.g. `./server-cli 7777 pubKey=pubKey.txt database=encDB.txt`

- Start the client: `./client-cli <host> <port> [ optional parameters ]`

  `optional parameters` have the form `attr1=val1 attr2=val2 ...`

  `m` is a specific cyclotomic ring [default=14491]

  `L` is the number of levels supported [default=17]

  `secKey` is the location of the secret key [default=secKey.txt] (If the file exists, all information will be loaded from the file and the other optional parameters will be ignored. If the file does not exist, a new secKey will be automatically generated and saved to this file)

- Use the client:

  - Send the public key to the server: `init`
  - Add data to the database: `add <path-to-database-file>`
    An example `database-file`:

    ```
    1 7 22 12
    2 1 45 42
    3 1 38 56
    4 1 38 12
    5 7 29 31
    6 1 48 52
    7 1 38 33
    8 1 22 8
    9 7 47 41
    10 1 48 6
    ```

  - Turn on/off sorting: `sort [ on/off ]`
  - Exit: `exit`
  - Make a query: `query <query-type> [threshold-level] <attr1=val1> [attr2=val2 ...] return <return_attr1> [return_attr2 ...] [limit=<limit>] [offset=<offset>]`
    Note: `limit=0` means return all results
    E.g.
    `query conjunction 1=14 2=5 3=7 return 0 4 limit=10 offset=0`
    `query threshold 2 1=14 2=5 3=7 return 0 4 limit=10 offset=0`

3.2 Parallelising calculation of beta, β, and gZeta $g(\zeta)$

The computation of β and $g(\zeta)$ can be easily parallelized using multi-threading as there is no race condition among the computations, and we use low-level POXIS Threads in our implementation.

Due to the setting of our encryption scheme, we encrypt data in packed ciphertexts which is the smallest encrypted data unit in our implementation. A packed ciphertext can be simply understood as a ciphertext with multiple slots, and each of the slots can accommodate one encrypted data value, i.e. one attribute value.

Database Size	With Parallelization/sec	No Parallelization/sec
20×5	3.5	13
256×12	5	37
1200×12	22	142

Table 2: Time taken to compute β.

Database Size	With Parallelization/sec	No Parallelization/sec
20×5	3	3
256×12	6	7.5
1200×12	8	28

Table 3: Time taken to compute $g(\zeta)$.

For computation of β, the computation is done for every attribute for all database entries. As one column of the database will encrypt a particular attribute for all database entries, and one slot inside a packed ciphertext will encrypt one particular attribute value, if the number of packed ciphertexts is larger than or equal to the number of CPUs, we split one thread to compute on one packed ciphertext for one particular attribute in each round, and repeat the process for number-of-attributes times. Otherwise, one thread will compute on more than one packed ciphertext for the attribute to fully utilize the hardware.

For computation of $g(\zeta)$, the computation is only done for every entry instead. Therefore, we simply split number-of-CPUs threads to compute this many number of $g(\zeta)$s in one round, and repeat the process for all database entries.

With $m = 14491$, and $L = 19$, the difference in time taken (in seconds) to compute β and $g(\zeta)$ with and without parallelization are shown in Table 2 and Table 3. We test the implementation with different database size, $a \times b$, where a refers to the number database entries, and b refers to the number of attributes.

3.3 Implement adding database entries

We implement one single addNewEntries() function to set up the initial database at server and to add subsequent new entries to server. As we encrypt data in packed ciphertexts (Section 3.2), it is important that we fully utilize the capacity of each ciphertext, i.e. to use all its slots. Note that leaving empty slots will increase the size of the database and scatter the valid data values, which will undermine the efficiency of query. Therefore, we try to bind the new data to the last not-fully-utilized packed ciphertext which we keep track of, and only create new ciphertexts if it is necessary.

For example. assume that we encrypt data in packed ciphertexts, each with 300 slots. Further assume that the last ciphertext of the database has 100 unused slots available, and we wish to add 700 new attribute values to the database. Instead of having to create three new packed ciphertexts and waste 100 unused slots in the last ciphertext, we simply encrypt 100 attribute values in these unused slots, and create only two new ciphertexts to encrypt the remaining 600 attribute values. As such, the utilization of ciphertext space is maximized.

In order to realize this idea, the server will always update the client with the last ciphertext whenever new entries are being added to the database. Mean-

while, as there is no easy and efficient way to determine the number of unused slots in a packed ciphertext, we simply decrypt the last ciphertext, concantenate it with the new data in plaintext, encrypt the combined data, and send the new ciphertexts to the server.

3.4 Limit, offset, 0-1 sorting of query results

Due to the fact that the server is processing a query on encrypted data, it cannot selectively choose only data entries that satisfy the query. Instead, the server will just mask entries not selected with zeros, and then send the whole column back to the client, which subsequently decrypts and filters out the zero entries. As a result, the size of the data being transmitted for each query is proportional to the size of the database, which is large, causing significant overhead in network transmission time.

One way to tackle this problem is to include 2 more parameters in each query, namely limit (the number of selected entries to be returned) and offset (the number of selected entries before those to be returned). This would reduce the transmission size to be linear to limit. However, in order to do this, the server need to be able to group all the selected entries together. This prompts us to look into an algorithm to sort encrypted 0-1 arrays, since the server can compute a `selectionVector` of encrypted 0's and 1's which determines whether an entry is selected.

Because the server is working on encrypted data, the sorting algorithm need to be data-independent. Hence, we looked at existing parallel sorting network algorithm in the literature, namely odd even transposition sort, odd even merge sort, bitonic sort. Odd even transposition sort is the most simple algorithm to implement, however, its complexity of $O(\log n)$ is very high. While odd even merge sort and bitonic sort both have running time complexity of $O(n \log^2 n)$, the later is more complicated as it involves 2 kinds of compare and swap operations: one to keep the bigger and one to keep the smaller element. It should be noted that in our homomorphic encryption scheme, we want to minimize the number of multiplicative levels in our evaluation circuit, and supporting bitonic sort will require a circuit of higher multiplicative depth. Hence, we decided to adapt odd even merge sort for our application.

Since odd even merge sort is a well known sorting algorithm, we omit its details here and only describe our optimization of the compare and swap operation, which is based on the fact that we are only sorting a 0 or 1 vector. Specifically, `compareAndSwap(A, B, i, j)` will swap A_i with A_j and B_i with B_j if $A_i = 0$. This can be done with the following formula (note that addition and multiplication is done in base 2):

$$
\begin{aligned}
A_i &= A_i \times A_i + (A_i + 1) \times A_j \\
A_j &= (A_i + 1) \times A_i + A_i \times A_j \\
B_i &= A_i \times B_i + (A_i + 1) \times B_j \\
B_j &= (A_i + 1) \times B_i + A_i \times B_j
\end{aligned}
$$

Using this version of `compareAndSwap`, we manage build a sorting network that needs $O(\log^2 n)$ levels of multiplication. We use it to sort the query results of the PDQ we constructed earlier. With 338 entries, we need to set the number

of levels L = 110 for both processing of query and sorting results. With these parameters, sorting takes an average of 430 seconds, which is not yet practical. This can be attributed to the high number of levels ($O(\log^2 n)$) required by our sorting techniques which makes L increases significantly from 17 to 110. One possible future work would be to derive an algorithm that require fewer multiplicative levels.

4 Conclusion

The use of HE in private database query has certainly a very promising prospect given its better security and thus the possibility of outsourcing data to foreign servers.

On the other hand, the efficiency of HE schemes on handling complex queries on big databases or sorting of encrypted data must still be improved significantly to attain true utility in the market as demonstrated by the test statistics shown in the report.

References

[1] Myungsun Kim, Hyung Tae Lee, San Ling, Shu Qin Ren, Benjamin Hong Meng Tan, and Huaxiong Wang, *Better Security for Queries on Encrypted Databases*, submitted to IEEE Transactions On Information Forensics And Security, May 9, 2016.

Halevi,S., Shoup, V. (2013). *Design and Implementation of a Homomorphic-Encryption Library.*

Halevi,S., Shoup, V. (2014). *Algorithms in HElib. IACR Cryptology ePrint Archive 2014: 106.*