Homomorphic Database

CSC Project

2020/2021

Introduction

The goal of this project is training the students in the use of public key cryptography and homomorphic encryption.
With that purpose the students should develop an encrypted database system using i) public key cryptography for client authentication; ii) homomorphic encryption for confidentiality of stored data.

System Overview

The homomorphic database system comprises the following entities:

1. The administrator that sets everything;
2. The clients;
3. The database server;
4. The encrypted database.

The system architecture is depicted in Figure 1.

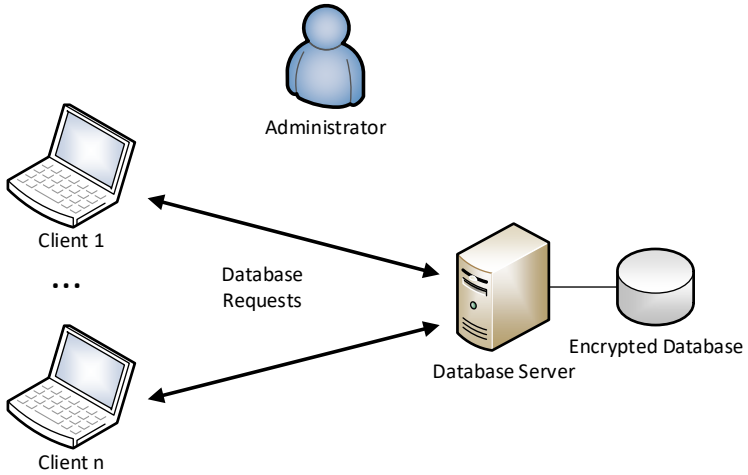

Figure 1: Homomorphic Database system architecture.

The server receives requests from clients to create, change, query and drop database tables. The database is shared between the clients, which means that a client can query or change tables created by another client. In a real system, the clients would authenticate before the server using Transport Layer Security (TLS). The server must not have access to the stored data belonging to the clients. In order to achieve this goal, the clients will encrypt the data that they intend to store in the database with the homomorphic encryption algorithm implemented in the Microsoft Simple Encrypted Arithmetic Library (SEAL). In this way, the

server can process the queries from clients (including data filters) with no need to decrypt the data.
The students must ensure that the communication between the clients and server are protected in terms of Confidentiality and Integrity.

Administrator

The **administrator** is not a process or any kind of service it is just the user running a set of tools to set things up. His goals are:

1) Generate a root CA certificate and private key;
2) Install the root certificate in the clients and server;
3) Generate a certificate for every client (e.g. with OpenSSL);
4) Generate a certificate for the server (e.g. with OpenSSL);
5) Generate the database key - a special homomorphic key pair (e.g. using Microsoft SEAL library, see below);
6) Install on each client app:
    a. The root CA certificate;
    b. The client private key and certificate;
    c. The server certificate;
    d. The database key.

Database Server

The **server** offers an SQL-like API that can be used by the clients to create, change, query and drop database tables. The syntax of this API is specified in Table 1. The API commands/responses are encrypted so that they can only be issued/interpreted by the respective client.

Table 1: Database API.

API Commands	**Syntax**	**Obs.**
Create Table	CREATE TABLE *tablename* (*col1name*, *col2name*, … , *colNname*)	
Insert Row into Table	INSERT INTO tablename (col1name, … , colNname) VALUES (value1, .., valueN)	The values are encoded according to the database entry format (see below).
Delete Row from Table	DELETE FROM tablename WHERE col1name =|<|> value1 AND|OR col2name =|<|> value2	=, < and > are alternative operators and may differ between columns. The values are encoded according to the database entry format (see below).
Query Table	SELECT col1name, .., colNname FROM tablename WHERE col1name =|<|>	=, < and > are alternative operators and may differ between columns.

	value1 AND|OR col2name =|<|> value2	The values are encoded according to the database entry format (see below).
Sum Column Elements from Table	SELECT SUM(colname) FROM tablename WHERE col1name =|<|> value AND|OR col2name =|<|> value	=, < and > are alternative operators and may differ between columns. The values are encoded according to the database entry format (see below).
Multiply Column Elements from Table	SELECT MULT(colname) FROM tablename WHERE col1name =|<|> value AND|OR col2name =|<|> value	=, < and > are alternative operators and may differ between columns. The values are encoded according to the database entry format (see below).

The server also keeps the encrypted database. The server is unable to decrypt the data values issued in client commands or kept in the database. Consequently, in order to compare and find matches between the database entries and the requested values, the server will resort to the implementation of a digital comparator using homomorphic operations. A digital comparator is a circuit, which , given two binary values A and B, is able to evaluate the following functions, returning values 1 or 0 for each: $f_{A=B}$, $f_{A>B}$ and $f_{A<B}$. Because of homomorphic encryption, when performing comparisons between command values and database entries, the server will not have access to the actual result of these operations. However, the homomorphic digital comparator will allow the result of a query to be calculated and sent (homomorphically encrypted) to the client. The following example illustrates how this can be done.
Consider the database table depicted in Table 2, where $H()$ is the homomorphic encryption operator.

Table 2: Example database table.

Row	Age	Height
1	$H(23)$	$H(172)$
2	$H(45)$	$H(171)$
3	$H(34)$	$H(167)$
4	$H(23)$	$H(180)$

The client issues the following command, requesting the sum of the heights of all the individuals whose age is 23:

SELECT SUM(Height) FROM example_table WHERE Age = $H(23)$

Although the server is unable to read or to directly compare the Age table entries with the target value 23, it can still send the correct answer to the client, calculating it as follows:

$$H(answer) = \sum_{i=1}^{4} f_{A=B}(H(23), Age_i) \cdot Height_i$$

This is made possible by the special way in which data entries will be encoded, which is defined in the section about the encrypted database.

Encrypted Database

The **encrypted database** is structured into independent tables. For each table, the database keeps the following info:

1) Table name;
2) ID of the client that owns the table;
3) Column names (plaintext);
4) Data.

The data is structured in one or more rows. For each row, there is one data entry for each column defined for the table. The encoding of each data entry is the secret behind the query logic of the server. In order to make it possible to implement the digital comparator as well as to perform addition and multiplication operations on the data, each data entry will contain the data item x encoded twice as:

1) $H(x)$
2) $< H(x_0), H(x_1), \dots, H(x_{n-1}) >$,

where x_i is bit number i of x. Version 1) of the data item allows straightforward homomorphic addition and multiplication, while version 2) allows comparison between data values using the digital comparator operations. The inputs and outputs of a digital comparator of one bit are depicted in Figure 2. This 1-bit comparator can be cascaded in order to be able to compare numbers of more than 1-bit. As such, besides the A and B bits to be compared, it received inputs $I_{A>B}$, $I_{A=B}$ and $I_{A<B}$, which correspond to the output bits of the bit immediately to its left (i.e., more significant[1]): $O_{A>B}^{i+1}$, $O_{A=B}^{i+1}$ and $O_{A<B}^{i+1}$. When the output of $Compare(A_{i+1}, B_{i+1})$ is $O_{A>B}^{i+1} = 1$, $O_{A=B}^{i+1} = 0$ and $O_{A<B}^{i+1} = 0$, the output of $Compare(A_i, B_i)$ $O_{A>B}^{i} = 1$, $O_{A=B}^{i} = 0$ and $O_{A<B}^{i} = 0$ independently of the values of A_i and B_i. Mutatis mutandis, it is similar when $O_{A>B}^{i+1} = 0$, $O_{A=B}^{i+1} = 0$ and $O_{A<B}^{i+1} = 1$. Otherwise, the result will be the comparison of A_i and B_i. If the bits that you are comparing are the most significant of the respective numbers, just consider $I_{A>B}^{i} = 0$, $I_{A=B}^{i} = 0$ and $I_{A<B}^{i} = 0$.

In order to build the logical functions that implement logical functions $O_{A>B}$, $O_{A=B}$ and $O_{A<B}$, you must first build the respective truth tables. Then, you already know that any logical function can be built using AND and NOT gates (alternatively NAND gates)[2]. The trick now is to build AND and NOT functions based on arithmetic operations where the only valid numbers are 1 and 0.

---

[1] In fact, we can also implement comparators that receive input from the less significant bits, and it is equally simple. It's just a matter of adapting the logical function inside the comparator.

[2] You can also implement any logical function using only OR and NOT gates, or only NOR gates, but AND and NOT will be more useful in your case. =)

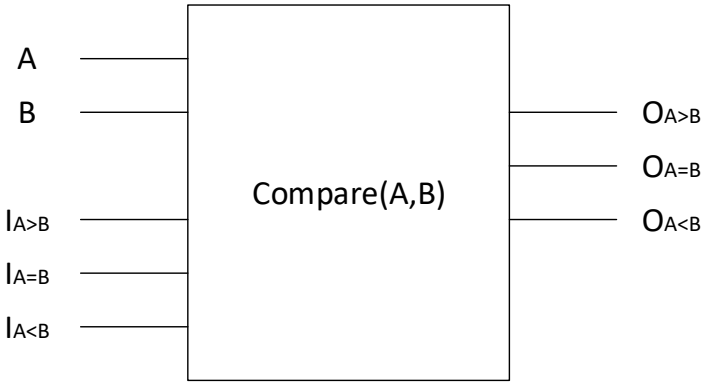

Figure 2: Inputs and outputs of a 1-bit digital comparator.

Client

The **client** issues commands to the database server. The client runs a command line application that:

1) Reads commands from the command line (query values are still plaintext);
2) For each command, it formats the query values according to the database entry format, making use of the database key and Microsoft SEAL library;
3) Signs each command using the client's private key and the libcrypto library (see below);
4) Encrypts the commands using the server's public key and the libcrypto library (see below);
5) Sends the commands to the database server;
6) Receives the responses from the database server;
7) Decrypts and displays the results of the commands.

Libraries and references

For homomorphic encryption use the Microsoft SEAL library available here:
https://github.com/Microsoft/SEAL

For public key encryption in C, use, for instance, the libcrypto available here:
https://github.com/openssl/openssl

You may find example code to sign and verify documents with libcrypto here:
https://wiki.openssl.org/index.php/EVP_Signing_and_Verifying