CMPSC 174A: Fall 2015

Final Project

Due Date: Demo on December 7, 2015 (Monday) – No late submissions!

This is the final project of the course and is a team project (maximum size of team: 2). The objective of this project is to implement and evaluate a secure database service prototype that stores data encrypted in the cloud and allows clients to run aggregate SQL queries over it. To ensure data confidentiality, it is required to use homomorphic encryption schemes, which allow performing addition or multiplication operations over encrypted ciphertexts without the need for decryption while ensuring strong security. In a nutshell, you will write an aggregation User-Defined-Function (UDF) for MySQL that implements the SUM operation over encrypted values. The MySQL server will run on an Amazon AWS instance and your client should connect remotely to issue queries: INSERT, simple SELECT, and SELECT SUM().

You can test your implementation locally before moving to the cloud, which will require to install a MySQL v5.7 server, if you haven't already for the previous project: `https://dev.mysql.com/downloads/`. UDFs in MySQL can only be written in the C language so also make sure that your development environment has a C compiler installed (gcc).

You will demonstrate your solution on Monday, December 7, 2015. We will ask you to reserve a 15-minute timeslot soon. You are still required to submit your solution through Gauchospace as with the previous projects. The deadline for this submission will be on the same day with the demo, **at 9am**. Late submissions will not be accepted. Your submission should be a set of files in a zip archive named *TeamName*_final_project.zip.

Project description: Security and privacy concerns of outsourced databases remain roadblocks to the widespread adoption of the cloud for the data management. Although encrypting data in the public cloud is an obvious solution for data confidentiality, making use of the encrypted data in the cloud is still a difficult task. Recently, homomorphic encryption has gained increasing attention due to its capabilities to allow computing arbitrary functions over encrypted data. Fully homomorphic encryption is used to solve general database queries such as selection and range.

For this project you are required to:

1. Read the literature about homomorphic encryption and understand its basic concept. There is a nice Wikipedia article about Homomorphic Encryption here: `https://en.wikipedia.org/wiki/Homomorphic_encryption`. Additionally, you can find more information about the specific scheme you are asked to implement (Pailler) here: `https://en.wikipedia.org/wiki/Paillier_cryptosystem` and in the original publication:

http://link.springer.com/chapter/10.1007%2F3-540-48910-X_16 (free access if you visit the url from a campus IP).

2. Implement a specific partially homomorphic encryption scheme called **Paillier** as a C library. The implementation of the Pailler algorithm must be in C so that you can use part of the code in your UDF which has to be C. While you are welcome to implement the library from scratch, *feel free to use existing libraries available online*. Regardless of what you choose to do, your implementation should be able to create **128bit encrypted values**. Your library needs to support the following operations:

   - Create a pair of public and private keys.
   - Encrypt a numeric value using a public key.
   - Decrypt an encrypted value using the private and public keys.
   - Homomorphic sum of encrypted values using the public key.

3. Initialize a MySQL **v5.7** database on AWS that is remotely accessible. Every team will be allocated with an AWS account so that you can create an instance and work on it. You can either create a clean instance or use a provided Ubuntu image that has most of the necessary development tools and MySQL server already installed. After instantiating the database server you need to create a single database called *project*. Within this database, you need to create a single table named *Employees* with three fields: A numeric *id* (primary key), another numeric field named *age*, and a *NOT NULL* field named *salary*. For the purposes of this project, we want the salary of every inserted employee to be encrypted so that even the database admin cannot see these values. Therefore, the field type of the salary must either be TEXT or BLOB so that you can store encrypted values.

4. Implement a MySQL user defined function (UDF) named *SUM_HE* that performs homomorphic summation over encrypted values. Documentation on UDFs can be found here: http://dev.mysql.com/doc/refman/5.7/en/adding-udf.html. A useful tutorial on writing an aggregation function can also be found here: http://www.codeproject.com/Articles/15643/MySQL-User-Defined-Functions. Your code needs to be compiled as a shared library and then placed in a directory where MySQL can find it and dynamically load it. Note that if you have a bug, the MySQL server will crash and debugging can be tricky. Make sure that your encryption/decryption/homomorphism implementation works well before testing it as a UDF. We also suggest that you first implement a simple SUM before the encrypted SUM. Remember to drop and reload your UDF every time you make a change. Also make sure that your UDF

code is technically correct: you release any allocated resources, you don't let the UDF crash if a value is NULL or arithmetic, etc.

5. Implement a client interface that connects remotely to the MySQL server and supports encrypted data insertion and select queries. You can use any language you want but keep in mind that you will also need to encrypt and decrypt values therefore you will have to use your Peiller library which will be in C. The client should have a **command line interface** that supports and displays the results of the following commands:

   (a) INSERT *emp_id emp_age emp_salary*: Insert a single entry for an employee which consists of a numeric id, a numeric age, and a numeric salary. The salary needs to be encrypted before sending the insert command to the MySQL server. If the id already exists in the table, the insert should fail with a message.

   (b) SELECT *emp_id*: Select and display all the information about a specific employee with id *emp_id*. Display the employee id (if the employee exists), his or her age, and the decrypted salary. If the employee does not exist show a failure message.

   (c) SELECT *: Select and display all the information about *every employee*. If there are no employees in the table display an appropriate message stating this fact.

   (d) SELECT SUM [WHERE *conditions*] [GROUP BY age] [HAVING *conditions*]: Calculate the sum of the salaries of specific employees or groups of employees. Note that the sum calculation has to be performed **on the server side** using your SUM_HE UDF and then decrypt the result on the client side. If there are no employees, the displayed sum should be NULL. The SELECT command should also support the **optional** clauses WHERE, GROUP BY, and HAVING:

      - WHERE *id_conditions*: Select the employees that satisfy the provided conditions on the *id* or *age* columns. You can treat the whole substring between the WHERE keyword and the end of the command (or till the GROUP BY or HAVING clauses) as a correct WHERE statement that you can directly plug to a MySQL query; there is no need to parse it.
      - GROUP BY age: Group the employees by age. **You must also display the corresponding ages for every group if a GROUP BY is specified.** The GROUP BY clause will only be called with the age attribute. Any other GROUP BY should be ignored.
      - HAVING *conditions*: Keep only the groups that satisfy some specific conditions. Again, there is no need to parse the *conditions* string, just plug it in the MySQL query.

      Examples of valid commands and the corresponding MySQL command you should produce:

- INSERT 12 45 95000

  ```
  INSERT INTO Employees VALUES(12, 45, c2dc865...d6034ba1f58c563af);
  ```

- SELECT 25

  ```
  SELECT id, age, salary FROM Employees WHERE id = 25;
  ```

- SELECT *

  ```
  SELECT id, age, salary FROM Employees;
  ```

- SELECT SUM WHERE id > 5

  ```
  SELECT SUM_HE(salary) FROM Employees WHERE id > 5;
  ```

- SELECT SUM WHERE id < 4 OR id >= 8

  ```
  SELECT SUM_HE(salary) FROM Employees WHERE id < 4 OR id >= 8;
  ```

- SELECT SUM WHERE 2 <> id

  ```
  SELECT SUM_HE(salary) FROM Employees WHERE 2 <> id;
  ```

- SELECT SUM WHERE id <> 5 AND 2 < id AND (age >= 15 OR id = 0)

  ```
  SELECT SUM_HE(salary) FROM Employees
  WHERE id <> 5 AND 2 < id AND (age >= 15 OR id = 0);
  ```

- SELECT SUM WHERE id > 13 GROUP BY age

  ```
  SELECT age, SUM_HE(salary) FROM Employees WHERE id > 13 GROUP BY age;
  ```

- SELECT SUM GROUP BY age

  ```
  SELECT age, SUM_HE(salary) FROM Employees GROUP BY age;
  ```

- SELECT SUM WHERE (NOT id > 103) OR age <= 28 GROUP BY age HAVING COUNT(*) > 2

  ```
  SELECT age, SUM_HE(salary) FROM Employees
  WHERE (NOT id > 103) OR age <= 28
  GROUP BY age HAVING COUNT(*) > 2;
  ```

- SELECT SUM GROUP BY age HAVING age > 35 OR age < 30

  ```
  SELECT age, SUM_HE(salary) FROM Employees
  GROUP BY age HAVING age > 35 OR age < 30;
  ```

If a select query produces an error when executed on MySQL your client *must not crash*. Catch the error and display a failure message.

(e) SELECT AVG [WHERE *id_conditions*] [GROUP BY age] [HAVING *conditions*]: Same as with the sum, but now we want to get the average salary. Since your UDF will only support a simple SUM, you will need to issue two queries: One to obtain the encrypted SUM for the employees or groups that satisfy the conditions and one to obtain the row count of these employees. You can then decrypt the sum and combine it with the count to obtain the average salary. If there are no employees the displayed avg should be NULL.

(f) exit: Closes the connection to the database server and terminates the client.

Submit on Gauchospace all the necessary code files (pailler library, UDF, client, etc.) and a single *.txt* file with your team name and instructions on how to compile and install your code on your AWS instance. For the demo you will need to have a running AWS instance at the time of the demonstration with a clean Employees table (no rows inserted yet). During the demo you will be asked to issue some insert commands on your client that runs on a local machine, then show us that the data is correctly inserted in the remote database on AWS, and finally to issue a few select sum/avg queries.

If you have any questions/concerns, you can post it on Piazza or email us.