Final Project Report for FA20 CSE208

Xiaohan Fu
UC San Diego
x5fu@ucsd.edu

1. Introduction

In this project, my major accomplishments are:

- Exploration on multiple FHE libraries, including SEAL, Lattigo, and Palisade;
- Better understanding of `CMake` and 3rd party libraries for `C++`;
- Implementation of an anonymous rider-driver pairing application with batching and serialization (code available here).

My initial plan was to use Lattigo as `golang` is more widely used in distributed systems. However after installed Lattigo and played with the samples a little bit, I found that the documentation is not that good and I felt a little bit confused with their APIs. Also, I felt I hadn't written `C++` for a while so it might be a good idea to take a look at the `C++` libraries. I studied Palisade, SEAL, and HElib and found their APIs to be generally similar. When building and installing Palisade, I met some problems that I couldn't resolve so I moved on to Microsoft SEAL. It has the best documentation (inline comments in source code) from my perspective as well. The installation process was non-trivial and interesting as I have never systematically used CMake and 3rd party libraries for C++ in the past. But fortunately everything worked well.

2. Implementation

The application simulates the situation where an anonymous rider needs to find the closest driver around to pick her up, hinted by the sample code of Lattigo [1]. On top of it, batching and serialization is ultized. The locations of the rider and drivers are represented as points in a rectangular grid. Server knows that the rider and some drivers are located within this area (this grid), but should not know their exact coordinates.

I strongly recommend to read the well-commented source code and run the application to see the output report for more details.

2.1. Workflow

The application is simulating the sense of server-client communications of real rider-driver pairing applications with two shared streams among users (drivers, rider) and servers for data (ciphertext, public key) and parameter settings.

The workflow in general is:

1. The server publishes and sets the parameters of the FHE context;
2. The rider follows the parameters and generated a pair of `pk` and `sk`, then she uses her `sk` to symmetrically encrypt her coordinates and send `pk` and the ciphertext to the server;
3. The server broadcasts her pk to all the drivers in this region;
4. Drivers encrypt their own coordinates with the rider's `pk` and send them to the server;
5. The server evaluates the squared distance between the rider and each of the drivers homomorphically with relinearization after each operation and then send back `result` to the rider;
6. The rider decrypts `result` with her `sk`, and identifies the closest driver.

Note. Coordinates of the rider and drivers are randomly generated for each execution.
The security parameters chosen in Step 1 is:

polynomial modulus degree	ciphertext coefficient modulus	plaintext modulus
$N = 2^{13} = 8192$	218 (suggested by the helper function)	$T = 65929217$

A relatively small polynomial modulus was used here for small ciphertext size and faster execution. To support batching, the plaintext modulus T was chosen to be a prime number congruent to 1 modulo 2*poly_modulus_degree ($T = 65929217$ specifically here).

2.2. Batching

As hinted by the documentation of SEAL [2], I used batching for full utilization of the the plaintext polynomial. The total size of the batching vector is exactly the polynomial modulus degree N.

In Step 2, the plaintext that the rider is encrypting is not a single pair of her coordinates (rider_x, rider_y), but in the format of

$$[rider_x, rider_y, rider_x, rider_y, ..., rider_x, rider_y].$$

Accordingly, in step 4, each driver is encrypting its coordinates in the format where for i-th driver, all coordinates are zero except $(2i)$-th and $(2i+1)$-th coordinates. For example, for driver[1], it looks like:

$$[0, 0, driver[1]_x, driver[1]_y, ..., 0, 0].$$

Under this setup, in Step 5, the server just need to evaluate the summation of all drivers' ciphertexts and substract the rider's ciphertext from it, which gives a ciphertext which can be decrypted to:

$$[driver[0]_x - rider_x, driver[0]_y - rider_y, driver[1]_x - rider_x, driver[1]_y - rider_y, ...]$$

, and then apply square inplace to obtain the evaluation ciphertext `result`.

Therefore, computing `result'[2i]+result'[2i+1]`, the distance between the rider and i-th driver can be obtained immediately after the decryption (gives `result'`). The rider then can find the closest driver easily.

Note. Because of the limitation of batching slots available, We can have at most $N/2$ drivers.

2.3. Serialization and Communication Channel

I didn't write a pair of client-server applications, but used two shared streams to simulate the communication between client and server. It can be transformed to two client-side and server-side applications with minimum efforts by replacing the dummy streams used here with a real data transfer stream for `TCP/UDP`.

- `parm_stream:` used by the server to pass the parameters setting to clients.
- `data_stream:` used by servers and clients to exchange public keys, linearization keys, and ciphertexts.

To optimize the communication package size transferred, I took use of serialization of the SEAL objects. Public keys, relinearization keys, symmetric encrypted ciphertexts in SEAL can all be serialized to have better space efficiency. However, public-key encrypted ciphertexts cannot be serialized. Therefore it is preferred to use the symmetric encryption if possible. That's why in Step 3, the rider is using her `sk` rather than `pk` to do the encryption.

2.4. Some Notable Observations

Here are some interesting findings that I observed when playing with my application:

1. The documentation of SEAL claims that the library built with `clang++` can run significantly faster than the one built with `g++`. But my program linked with them respectively does not show much difference given a relatively long execution time ~15s.
2. The noise budget remaining in the result ciphertext does not change much from maximum number of drivers (96 bits) i.e. $N/2$ drivers to 3 drivers (101 bits).
3. The noise budget remaining in the result ciphertext does not change at all with or without relinearization after the additions and square operation. I can't explain this well.
4. A smaller plaintext modulus gives more remaining noise budget but has no impact on the overall running time.
5. A smaller polynomial modulus degree i.e. $N = 4096$ with maximum drivers configured gives a much faster running speed from 15s to 2.5s, while at the cost of a much smaller remaining noise budget from 96 bits to 8 bits.

3. Conclusion

In this project, I experienced how to use a FHE library to implement an application with FHE in practice. I better understood the knowledge in lectures and learned how to play with 3rd party libraries in C++. I implemented an application for oblivious riding with batching and serialization.

References

[1] “Lattigo v2.1.0.” Online: http://github.com/ldsec/lattigo. Dec. 2020. EPFL-LDS.

[2] “Microsoft SEAL (release 3.6).” https://github.com/Microsoft/SEAL. Nov. 2020. Microsoft Research, Redmond, WA.