TAMPERE UNIVERSITY – DEPARTMENT OF COMPUTING SCIENCES

Security Protocols: Helping Alice and Bob to Share Secrets (COMP.SEC.220)

Tutorial 10: Homomorphic Encryption

Antonis Michalas
antonios.michalas@tuni.fi

Mindaugas Budzys
mindaugas.budzys@tuni.fi

Hossein Abdinasibfar
hossein.abdinasibfar@tuni.fi

August 19, 2024

SUBMISSION DEADLINE: 02.12.2024 AT 23:00

Tutorial Description

In Lecture 10, Homomorphic Encryption (HE) was introduced to you. In this tutorial, you will delve deeper into the HE concept and explore its various applications by implementing a basic homomorphic encryption scheme. In this exercise, you will compute encrypted data to understand HE better.

INTRODUCTION TO HOMOMORPHIC ENCRYPTION

Homomorphic Encryption (HE) is a powerful cryptographic primitive that enables performing computations on encrypted data without decrypting the ciphertext. Overall, an HE scheme consists of four different algorithms as follows:

1. **Key Generation**: The key generation algorithm takes as input the security parameter λ and outputs a public key pk, evaluation key evk, and a secret key sk.

   Algorithm 1: HE Key Generation

   Input: 1^λ
   Output: pk, sk, evk
   $(\mathsf{pk}, \mathsf{sk}, \mathsf{evk}) \leftarrow HE.KeyGen(1^\lambda)$

2. **Encryption**: The encryption algorithm takes as input the public key pk and a plaintext (a vector) m and outputs a ciphertext (a vector) c.

   Algorithm 2: HE Encryption

   Input: $m = [x_1, x_2, \ldots, x_n]$
   Output: $c = [c_1, c_2, \ldots, c_n]$
   $c \leftarrow HE.Enc(\mathsf{pk}, m)$

3. **Evaluation**: The evaluation algorithm takes as input the evaluation key evk, a ciphertext c and a function f (a specific function to compute over encrypted data, e.g., addition), and outputs a ciphertext c_f.

Algorithm 3: HE Evaluation

Input: $c = [c_1, c_2, \ldots, c_n]$, f
Output: c_f
$c_f \leftarrow HE.Eval(\mathsf{evk}, f, c)$

4. **Decryption**: The decryption algorithm takes as input the secret key sk and the ciphertext resulted after doing evaluation c_f, and outputs the final result plaintext m_f, which should be the result of running f over plaintext data.

Algorithm 4: HE Decryption

Input: c_f
Output: m_f
$m_f \leftarrow HE.Dec(\mathsf{sk}, c_f)$

Task 1 – Drawing the Protocol

For this task, let us consider a scenario where Alice wishes to store some data with a server (Charlie). In the assumed scenario, Alice is the data owner, and her data contains many integers. Although Alice wishes to store her data with Charlie, she does not want Charlie to know anything about the data. Additionally, she wants to be able to perform some computations on the encrypted data stored with Charlie without revealing the contents of her data. To do this, Alice uses the HE scheme described above. Using the HE scheme, Alice encrypts her data and transfers the encrypted data to Charlie. Subsequently, she requests that Charlie perform some computation on the encrypted data, which is transferred back to her. Finally, Alice decrypts the result with her secret key.

Task Goal: Your first task is to draw a simple protocol sequence diagram for the scenario described above. Your protocol should include the four steps of HE (Key Generation, Encryption, Evaluation, and Decryption) and the two entities (Alice and Charlie).

Task 2 – Implementing the Protocol

For this task, we will be using the OpenFHE library, which is a standard Homomorphic Encryption (HE) library, to implement the protocol you designed in the previous task. You can also choose any operating system you prefer, but Ubuntu Linux is recommended. We will first install and build the library by following the instructions below. If you wish to use another operating system, you can check this link to install and build on your chosen system .

2.1 Install Pre-requisites and Build the Library

For this step we are using Ubuntu Linux 20.04.2 LTS, and we assume that you have already installed it on your machine!

1. Install these pre-requisites (if not already installed) and set the default compiler.

```
$ sudo apt update
$ sudo apt install git
$ sudo apt-get install build-essential #this already includes g++
$ sudo apt-get install cmake
# in case that there was a problem with g++ version, you can install clang
$ sudo apt-get install clang
$ sudo apt-get install libomp5
$ sudo apt-get install libomp-dev
$ export CC=/usr/bin/clang-11
$ export CXX=/usr/bin/clang++-11
$ sudo install autoconf
```

2. Create a Folder for your implementation and clone the OpenFHE library.

```
$ mkdir ~/Implementation
$ cd ~/Implementation
$ git clone https://github.com/openfheorg/openfhe-development.git
$ cd openfhe-development
$ git submodule update --init --recursive
# if there was an error in the build process, you can checkout to the dev branch
$ git checkout dev
```

3. Build the library.

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ make install # optional
```

4. Run the examples.

```
$ cd ~/Implementation/openfhe-development/build/bin/examples/pke/
$ ./simple-integers
# by running the above command, you should see three plaintext (vector of integers)
    and the results of homomorphic computations (addition and multiplication) over
    them.
```

5. Take a screenshot of the output when you run the examples script in step 4 and add it to your submission.

2.2 Simple HE Protocol

Once the library has been built and installed, we are now ready to write a simple code to implement the designed protocol. We will be implementing a simple addition ($sum = x_1 + x_2$) over two or more encrypted vectors of integers.

1. Create a file "mycode.cpp" in "openfhe-development/src/pke/examples" path.
2. Open the mycode.cpp file and write the following code:

```
#include "openfhe.h"

using namespace lbcrypto;

int main() {
   // Step 1: Set CryptoContext
   CCParams<CryptoContextBFVRNS> parameters;
   parameters.SetPlaintextModulus(65537);
   parameters.SetMultiplicativeDepth(2);
   CryptoContext<DCRTPoly> cryptoContext = GenCryptoContext(parameters);
   // Enable features that you wish to use
   cryptoContext->Enable(PKE);
   cryptoContext->Enable(KEYSWITCH);
   cryptoContext->Enable(LEVELEDSHE);
   // Step 2: Key Generation
   // Initialize Public Key Containers
   KeyPair<DCRTPoly> keyPair;
   // Generate a public/private key pair
   keyPair = cryptoContext->KeyGen();
   // Generate the relinearization key
   cryptoContext->EvalMultKeyGen(keyPair.secretKey);
```

```
    // Step 3: Encryption
    // First plaintext vector is encoded
    std::vector<int64_t> vectorOfInts1 = {1, 2, 3, 4, 5};
    Plaintext plaintext1 = cryptoContext->MakePackedPlaintext(vectorOfInts1);
    // Second plaintext vector is encoded
    std::vector<int64_t> vectorOfInts2 = {3, 2, 1, 4, 5};
    Plaintext plaintext2 = cryptoContext->MakePackedPlaintext(vectorOfInts2);

    // The encoded vectors are encrypted
    auto ciphertext1 = cryptoContext->Encrypt(keyPair.publicKey, plaintext1);
    auto ciphertext2 = cryptoContext->Encrypt(keyPair.publicKey, plaintext2);

    // Step 4: Evaluation
    // Homomorphic additions
    auto ciphertextAddResult = cryptoContext->EvalAdd(ciphertext1, ciphertext2);

    // Step 5: Decryption
    // Decrypt the result of additions
    Plaintext plaintextAddResult;
    cryptoContext->Decrypt(keyPair.secretKey, ciphertextAddResult,
        &plaintextAddResult);

    std::cout << "Plaintext #1: " << plaintext1 << std::endl;
    std::cout << "Plaintext #2: " << plaintext2 << std::endl;

    // Output results
    std::cout << "\nResults of homomorphic computations" << std::endl;
    std::cout << "#1 + #2: " << plaintextAddResult << std::endl;
    return 0;
}
```

3. Save the file and run the following commands to build and execute your code:

```
$ cd ~/Implementation/openfhe-development/build
$ make
$ cd ~/Implementation/openfhe-development/build/bin/examples/pke/
$ ./mycode
```

If everything works well, you should see the following output:

```
Plaintext #1: (1, 2, 3, 4, 5 ...)
Plaintext #2: (3, 2, 1, 4, 5 ...)

Results of homomorphic computations
#1 + #2: (4, 4, 4, 8, 10 ...)
```

Take a screenshot of your results and add it to your submission pdf.

Additional Task Goal: Expand the code to execute a simple aggregation for 10 vector inputs (ciphertexts) instead of 2 as implemented above. (You should generate and use 5 random integers between 1 and 10, for each vector input).

TASK 3 – ANALYZING THE IMPLEMENTATION

The code you wrote in Task 2 should be analyzed to measure some metrics, including execution time and storage costs. The first step in this task is to save each key (public, secret, and evaluation keys) in a separate file and measure their sizes. Afterward, you should measure how long it takes to execute key generation, encryption, evaluation (addition), and decryption. To conclude, you should prepare a report summarizing your findings. Make sure your final report includes statistics (for key sizes and execution times).

FINAL SUBMISSION

Your final pdf submission should include the following:

- **Task 1**: The sequence diagram for the HE protocol and the description for each step.
- **Task 2**: A short description of what you have done, your code, and the screenshots of code execution.
- **Task 3**: A short description of the analysis findings and a measurement results table.