Primer Guide

This tutorial will guide you through development of your very own applications which can take advantage of homomorphic encryption. We will demonstrate the ability to encrypt and decrypt given phrases. These phrases can be simple such as a word or it can be operations such as multiplication or addition. This guide assumes you have gained an understanding of the basic Palisade Library, build your own projects using cmake, and how to implement some basic programs. If you need some assistance, please refer to the Palisade Documentation for further information. Follow these links for more information:

Official Documentation: https://palisade-crypto.org/documentation/

GitLab Wiki: https://gitlab.com/palisade/palisade-release/-/wikis/home

Primer Program #1: Homomorphic Addition and Multiplication

To create a program that can perform encrypted operations, we need to first create a cryptocontext to be able to call the EvalAdd and EvalMult methods to add and multiply ciphertexts. These methods enable encrypted plaintexts called ciphertexts to be used in mathematical operations. The steps to create the program is to start with creating a cryptocontext. To do this simply establish the parameters first. In this cryptocontext, we will be using DCRTPoly for the underlying lattice layer element.

We will use a simple plaintext modulus parameter. This signifies the maximum size of the data type to perform a modulus operation, in this case it is an unsigned 32 bit integer:

```
int plaintextModulus = 65537;
```

Next we will establish sigma which will be as follows:

```
double sigma = 3.2;
```

Then we determine the security level which can be 128, 192, or 256 bit security. In this case it is 128 bit. This is shown below:

```
SecurityLevel seclvl = HEStd_128_classic;
```

Next we determine the maximum key-switching depth for the computation using an unsigned integer that is 32 bits long and defined as follows:

```
uint32_t depth = 2;
```

Finally we can create the cyrptocontext using our chosen lattice layer, DCRTPoly, and the parameters we have established. We also have to enable encryption and SHE which is somewhat homomorphic encryption which allows for limited computation on encrypted data.

```
CryptoContext<DCRTPoly> cryptoContext =
CryptoContextFactory<DCRTPoly>::genCryptoContextBFVrns(plaintextModulus, seclvl, sigma, 0,
depth, 0, OPTIMIZED);

cryptoContext->Enable(ENCRYPTION);

cryptoContext->Enable(SHE);
```

We then establish our keypair so that we can encrypt our plaintext into ciphertext. To do this we need to use the KeyGen method provided by the cryptocontext.

```
LPKeyPair<DCRTPoly> keyPair;
```

```
keyPair = cryptoContext->KeyGen();
```

Then we use the EvalMultKeyGen method to enable the EvalAdd and EvalMult operations.

```
cryptoContext->EvalMultKeyGen(keyPair.secretKey);
```

Now we can declare our plaintext. In this case, we are using an integer list using a vector array. We will declare three separate plaintexts to perform operations on. We will call the method MakePackedPlaintext to create an efficient encoder packing multiple integers into a single plaintext polynomial.

```
std::vector<int64_t> intList1 = {1,2,3,4,5,6,7,8,9,10,11,12};

Plaintext plaintext1 = cryptoContext->MakePackedPlaintext(intList1);

std::vector<int64_t> intList2 = {12,11,10,1,2,3,9,8,7,4,5,6};

Plaintext plaintext2 = cryptoContext->MakePackedPlaintext(intList2);

std::vector<int64_t> intList3 = {12,11,10,9,8,7,6,5,4,3,2,1};

Plaintext plaintext3 = cryptoContext->MakePackedPlaintext(intList3);
```

Next we create the ciphertext by using the Encrypt method and the keypair public key we generated earlier along with the plaintext we want to encrypt as parameters.

```
auto ciphertext1 = cryptoContext->Encrypt(keyPair.publicKey, plaintext1);

auto ciphertext2 = cryptoContext->Encrypt(keyPair.publicKey, plaintext2);

auto ciphertext3 = cryptoContext->Encrypt(keyPair.publicKey, plaintext3);
```

We can finally perform the operations on the ciphertext using the EvalAdd and EvalMult methods. The parameters are the ciphertexts we generated or a result of a previous EvalAdd or EvalMult result.

```
auto add12 = cryptoContext->EvalAdd(ciphertext1,ciphertext2);

auto add123 = cryptoContext->EvalAdd(add12, ciphertext3);

auto mult12 = cryptoContext->EvalMult(ciphertext1, ciphertext2);

auto mult123 = cryptoContext->EvalMult(mult12, ciphertext3);
```

We can decrypt the results of the previous step using the Decrypt method along with the keypair secret key, the ciphertext addition or multiplication result, and the plaintext variable to save the decrypted result into as parameters.

```
Plaintext plaintextAddResult;

cryptoContext->Decrypt(keyPair.secretKey, add123, &plaintextAddResult);

Plaintext plaintextMultResult;

cryptoContext->Decrypt(keyPair.secretKey, mult123, &plaintextMultResult);
```

Finally we display the results along with the original plaintexts.

```
cout << "Plaintext #1: " << plaintext1 << std::endl;

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

cout << "Plaintext #3: " << plaintext3 << std::endl;
```

```
cout << "plaintexts 1 2 and 3 added together = " << plaintextAddResult << std::endl;

cout << "plaintexts 1 2 and 3 multiplied together = " << plaintextMultResult << std::endl;

return 0;
```

Example of output:

```
Plaintext #1: ( 1 2 3 4 5 6 7 8 9 10 11 12 ... )
Plaintext #2: ( 12 11 10 1 2 3 9 8 7 4 5 6 ... )
Plaintext #3: ( 12 11 10 9 8 7 6 5 4 3 2 1 ... )
plaintexts 1 2 and 3 added together = ( 25 24 23 14 15 16 22 21 20 17 18 19 ... )
plaintexts 1 2 and 3 multiplied together = ( 144 242 300 36 80 126 378 320 252 120 110 72 ... )
```

Primer Program #2: Input Phrase Encryption and Decryption

This program is very similar to the previous primer we went over. This program is designed to take input from the user such as a phrase and store it in a variable called phrase. The program then proceed to call the method ***MakeStringPlainText(string)*** to convert the phrase into plaintext:

```
Plaintext plaintext1 = cryptoContext->MakeStringPlaintext(phrase1);

Plaintext plaintext2 = cryptoContext->MakeStringPlaintext(phrase2);

Plaintext plaintext3 = cryptoContext->MakeStringPlaintext(phrase3);
```

This is then followed by calling the ***Encrypt*** method which takes our generated key and the phrase that would like to be encrypted.

```
auto ciphertext1 = cryptoContext->Encrypt(keyPair.publicKey, plaintext1);
```

```
   auto ciphertext2 = cryptoContext->Encrypt(keyPair.publicKey, plaintext2);

   auto ciphertext3 = cryptoContext->Encrypt(keyPair.publicKey, plaintext3);
```

We have now successfully encrypted the phrases that were inputted. Now the program calls the ***Decrypt*** method which takes in our generated key, encrypted text, and the reference of the variable where we are storing the decrypted text.

```
   DecryptResult result =
cryptoContext->Decrypt(keyPair.secretKey,ciphertext1,&plaintextDecrypted1);
```

Lastly, the program does 2 checks. First we see if the decryption was successful. If it is not then we let the user know.

```
   if (!result.isValid) {

       cout << "Decryption failed" << endl;

       return 1;

   }
```

Secondly, we check if the decrypted plaintext is the same as what we started with. If it's not the same we display the results of the decrypted text to the user for further inspection.

```
if( plaintext1 != plaintextDecrypted1 ) {

       cout << "First decrypted plaintext does not match original plaintext!" << endl;
```

```
    cout << "Original Plaintext: " << plaintext1 << endl;

    cout << "Decrypted Plaintext: " << plaintextDecrypted1 << endl;

    return 1;

}
```

If both checks have passed, this means that we have successfully encrypted the inputted phrase.

This process is repeated for each phrase that is inputted.

Example of expected output:

```
Please enter Phrase #1 to encrypt: sad
Please enter Phrase #2 to encrypt: happy
Please enter Phrase #3 to encrypt: mad
Successfully encrypted!
Now running decryption!
Decryption successful! Phrase #1:sad
Decryption successful! Phrase #2:happy
Decryption successful! Phrase #3:mad
```

Distribution of Primer Programs

The primer programs can be found on GitHud at the following link:

Program #1's source code file is Examples.cpp. Program #2's source code file is Example2.cpp.

Users should follow the steps to build and compile the programs.

Step 1. Build and install PALISADE using "make install". This will copy the PALISADE library files and header files to the directory chosen for installation.

Step 2.Create the folder for the project on the system.

Step 3.Copy CMakeLists. Use.txt form the root directory of the git repo to the folder for the project.

Step 4.Rename CMakeLists.User.txt to CMakeLists.txt.

Step 5.Update CMakeLists.txt to specify the name of the executable and the source code files. For example, include the following line

```
add_executable( fhe-demo demo-simple-Examples.cpp )
```

If using MinGW/Windows(skip this step for other platforms), copy PreLoad.cmke

Form the root directory of the git repo to the folder for the project.

Step 6. We create the build directory and cd to it

```
mkdir build

cd build
```

Step 7. Run

```
cmake -DPALISADE_DIR=`<path_to_palisade>` ..
```

Step 8. Run “make” to build the executable.

After building the executable, if users are using the linux environment and want to run the compiled program then type the command: ./<your_program_name>.

The distribution of the primer guide is in a pdf available on the GitHub link.