P2N3 Programmer's Guide

Contents

Introduction

P2N3 (Privacy-preserving neural network of NTU) is a C++ open-source library which implements an optimized fully homomorphic encryption (FHE) scheme for privacy-preserving neural networks.

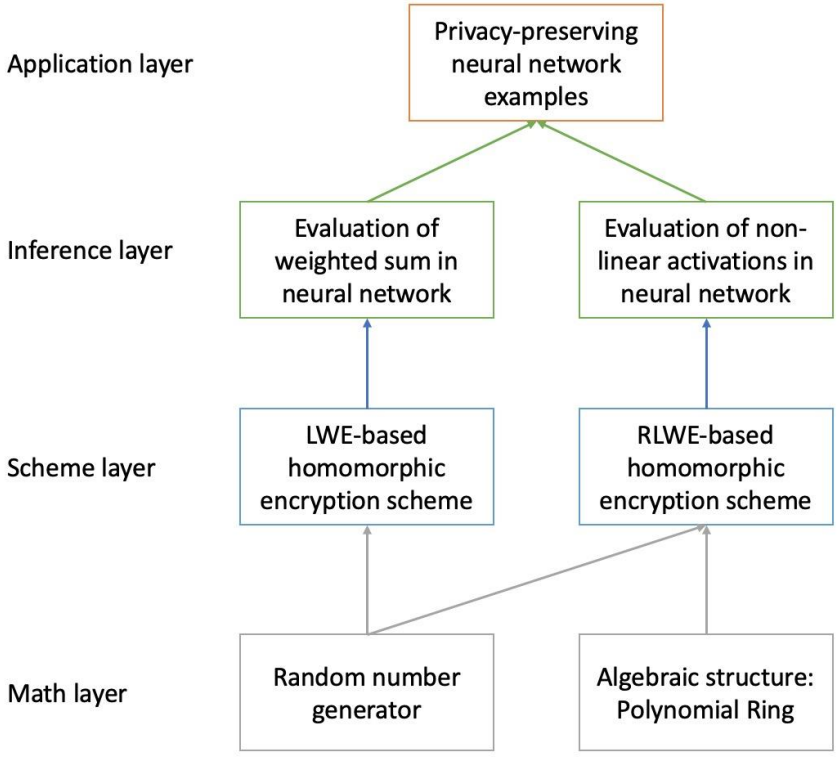
Our optimized FHE scheme enables us the ability to perform weighted sums and convolutions on the approximate LWE-based additive homomorphic encryption scheme, and to evaluate non-polynomial activations on FHEW ciphertexts.

our FHE scheme has the following properties:

- It can be applied to neural networks of arbitrary depth.
- It supports many kinds of widely used activations, such as ReLU and Sigmoid.
- When applied to inference of privacy-preserving neural networks, it is fast and accurate.

Library overview

The organization of P2N3 is shown in the following figure. It includes 4 layers. In math layer, we provide the base that our LWE-based scheme and RLWE-based scheme rely on. In scheme layer, we implement two schemes with homomorphic algorithm. In inference layer, we show how to apply our schemes to inference on encrypted data. Finally in application layer, we offer some examples in privacy-preserving neural network inference.



The library includes the following packages:

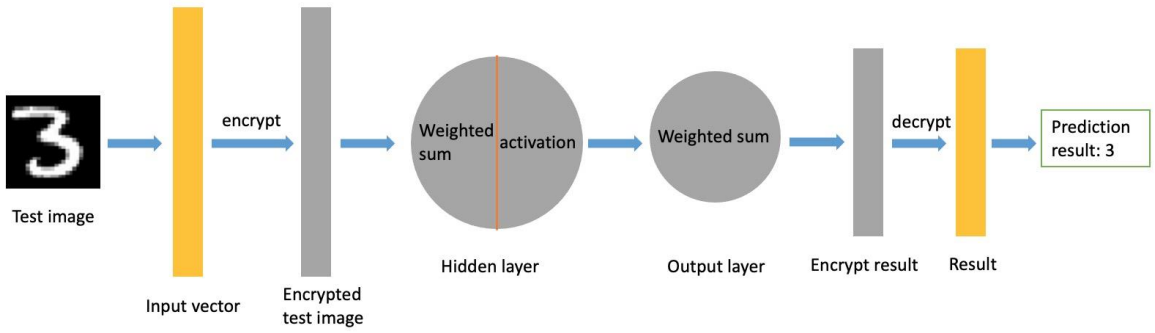
1. Random number generator: We offer a series of algorithms to generate samples from different distributions based on Openssl library (/index.html (openssl.org)), which are build

blocks for encryption schemes. The distributions include Bernoulli distribution, Ternary distribution and Uniform distribution.

2. Polynomial ring: We implement basic operations in a polynomial ring. It includes polynomial addition, polynomial multiplication, polynomial decomposition and so on.
3. LWE scheme: We implement an approximate additive homomorphic encryption scheme to support the evaluation of linear functions like weighted sum and convolutions. It includes key generation algorithm, encryption algorithm, decryption algorithm, homomorphic addition between ciphertexts, homomorphic addition between ciphertext and plaintext, homomorphic multiplication between ciphertext and plaintext and homomorphic rounding.
4. RLWE scheme: We implement a RLWE-based homomorphic encryption scheme to support the evaluation of non-linear activations. It includes key generation algorithm, encryption algorithm, decryption algorithm, extend RLWE encryption algorithm, RGSW ciphertext generation algorithm, extended RLWE ciphertext multiplication, external product, extract algorithm, key switch algorithm and look-up table algorithm.
5. Test files: We offer some sample tests to test the correctness of our implementations. It includes test on LWE scheme, test on RLWE scheme, test on polynomial ring and test on random number generator.
6. Privacy-preserving neural network examples: We offer some applications of our FHE scheme. It includes inference on MNIST dataset, facial recognition, speech recognition, text classification and object classification.
7. FasterNTT: FasterNTT(GitHub - nict-sfl/FasterNTT) is an open source library which offers a fast implementation of NTT algorithms. We use it to speed up our polynomial multiplication.

Examples

We take our inference in MNIST dataset(MNIST.cpp) as an example to show how to apply our FHE scheme to privacy-preserving neural network. We show the flow in the following figure. Test inputs and weights are stored in folder MNIST_data.



Network weights

At the beginning, we have to read the weights of network. As our scheme is based on integer, so we have to round the weights from floating numbers to integers. We have a scaler for each layer and can be set by users. In our MNIST example, we set the scaler of first layer is 16, and the scaler of second layer is 8. We multiply the weight by scaler and take its integer part. If the floating part > 0.5, then we add 1 to integer part.

```
1.  for (int i = 0; i < 784; i++)
2.   {
```

```
    for (int j = 0; j < nh; j++)
    {
      infile >> w1[i][j];
      double t = w1[i][j] * scaler1;
      ww1[i][j] = ((int)(w1[i][j] * scaler1));
      if (t - ww1[i][j] > 0.5 && t > 0) {
        w1[i][j]++;
      }
      else if (t < 0 && w1[i][j] - t>0.5) {
        w1[i][j]--;
      }
    }
  }
```

Security parameters

We have to fix our security parameters before we run encryption algorithms. We give a default setting with 80-bit security level in our project as following

Parameters for LWE scheme:

Vector length = 512, ciphertext modulus = 8192, variance of error = 0.125

Parameters for RLWE scheme:

Degree of polynomial ring = 2048, ciphertext modulus = 576460752213245953, variance of error = 0.125

In this open-source project, only the ciphertext modulus of RLWE scheme is fixed due to underlying polynomial multiplication algorithm, and other parameters can be set by users.

Other parameters

Besides of security parameters we showed above, some parameters which relate to evaluation of non-linear activations are needed. We give a default setting:

Delta = 1099511627776 and composition base of external product = 30 bits.

We recommend user to try our default setting first as we have tested that it works in many kinds of neural networks. For higher accuracy, we recommend user choose a smaller composition base, but it will result in longer inference time.

Key generation

The next step is to generate secret keys of LWE scheme, RLWE scheme and encrypted evaluation key.

Example of LWE secret key generation

```
int *x = new int [n1];
  x=LWE32_KeyGen(n1);
```

The generated key is stored in an array with length n1 (in default setting, n1=512). If you wish to apply our scheme in a client-server model, you can write it into a file and store in client's PC.

Example of RLWE secret key generation

```
polynomial s = RLWE64_KeyGen(n2);
  int *lwe_s = new int [n2];
  for (int i = (int)s[0] + 1; i >= 1; --i) {
    lwe_s[(int)s[0] + 1 - i]=s[i];
  }
```

The generated key is stored in an array with length n2 (in default setting, n2=2048). If you wish to apply our scheme in a client-server model, you can write it into a file and store in client's PC.

Example of encrypted evaluation key generation

```
vector<vector<polynomial>> ek0i= key_encrypt_1(n2, q2, k2, var2, x[2*i],x[2*i+1], s, b, logb);
vector<vector<polynomial>> ek1i = key_encrypt_2(n2, q2, k2,var2, x[2*i],x[2*i+1], s, b, logb);
vector<vector<polynomial>> ek2i = key_encrypt_3(n2, q2, k2,var2, x[2*i], x[2*i+1], s, b, logb);
```

The generated key is store in 3 vectors of polynomials. To speed up our evaluation of non-linear activations, we suggest to store it in NTT form, which means that we apply NTT transformation on each polynomial before we store it. User can insert the following code and the NTT form will be stored.

```
  //ntt ek0,ek1,ek2
   vector<vector<R>> _ek0,_ek1,_ek2;

  for (int i = 0; i < n1/2; ++i) {
    vector<R> tempek;
    vector<vector<polynomial>> ek0i= key_encrypt_1(n2, q2, k2, var2, x[2*i],x[2*i+1], s, b, logb);
    for(int j=0;j<2*(k/logb);++j){
      for(int k=0;k<2;++k){
        R tempp;
        for(int h=1;h<=n2;++h){
          if(ek0i[j][k][h]<0){
            ek0i[j][k][h]+=q2;
            tempp(0,h-1)=(uint64_t)ek0i[j][k][h];
          }
          else{
            tempp(0,h-1)=(uint64_t)ek0i[j][k][h];
          }
        }
        tempp.ntt();
        tempek.push_back(tempp);
      }
```

```
    }
    _ek0.push_back(tempek);
  }
```

Encryption

For every test input, we need to encrypt it and pass the ciphertext to the homomorphic evaluation of neural network. We use LWE-based scheme to encrypt the input. For example, in MNIST dataset, each input is a 28×28 grey-level image, which is represented by a vector with length 784 and stored in image[tt]. In our code, we encrypt each dimension of the vector, so we get 784 ciphertexts and each ciphertext is a vector of integers.

```
vector<vector<int>> ct_i;
  for (int j = 0; j < 784; ++j) {
    vector<int>ctj = LWE32_Enc(q1, n1,var1, 2*image[tt][j], x);
    ct_i.push_back(ctj);
  }
```

Homomorphic evaluation

1. Construct LUT function f. the LUT function f is decided by the activation function in neural network. In our default setting, the activation function is ReLU.

```
//lut, construct polynomial f
polynomial f = p_f(n2, delta, q1);
```

2. Evaluate the first weighted sum. The weights are stored in matrix ww1 and the biases are stored in ibias1. To calculate a weighted sum, we first homomorphically multiply the ciphertext and the corresponding weight. Then we add these 784 ciphertexts together. Finally we add the bias to the sum of ciphertext. The result is a LWE ciphertext.

```
for (register int i = 0; i < 784; ++i) {
    for (register int j = 0; j < nh; ++j) {
      vector<int> ct_iip0 = LWE32_Plain_Multi_ct(q1, n1, ct_i[i],
ww1[i][j]);
      if (i == 0) {
        ct_ip1.push_back(ct_iip0);
      }
      else {
        ct_ip1[j] = LWE32_Add_ct(q1, n1, ct_iip0, ct_ip1[j]);
      }
    }
  }

for (int i = 0; i < nh; ++i) {
  ct_ip1[i] = LWE32_Plain_Add_ct(q1, n1, ct_ip1[i], 2*ibias1[i]);
}
```

3. Evaluate the activation. We use homomorphic look-up-table (LUT) algorithm to evaluate the activations. The output of LUT is a RLWE ciphertext and an index to indicate the number of node(necessary in multi-thread case, in multi-thread case, we need to sort the output according to this index). Then we call Extract0 function to extract a LWE ciphertext from a RLWE ciphertext as the result.

```
for (int i = 0; i < nh; ++i) {
    vector<polynomial> relu=LUT_2048<2048,1,FNTT,AVX2>(i,q1,q2, n2, delta, k2, n1,ct_ip1[i], _ek0, _ek1,_ek2, f,b,logb);
    relu.pop_back();
    vector<int64_t> ans2 = Extract0(q2,delta, n2, relu);
    ct_lut.push_back(ans2);
  }
```

4. Evaluate the second weighted sum. The weights are stored in matrix ww2 and the biases are stored in ibias2. The algorithm is the same as evaluating the first weighted sum

```
  for (int i = 0; i < classes; ++i) {
     vector<int64_t> ct_iip0 = LWE64_Plain_Multi_ct(q2, n2, ct_lut[0], ww2[0][i]);
     for (int j = 1; j < nh; ++j) {
       vector<int64_t> ct_iipj = LWE64_Plain_Multi_ct(q2, n2, ct_lut[j], ww2[j][i]);
       ct_iip0 = LWE64_Add_ct(q2, n2, ct_iip0, ct_iipj);
     }
     ct_iip0 = LWE64_Plain_Add_ct(q2, n2, ct_iip0, ibias2[i]*delta);
     ct_ip2.push_back(ct_iip0);
   }
```

Decryption

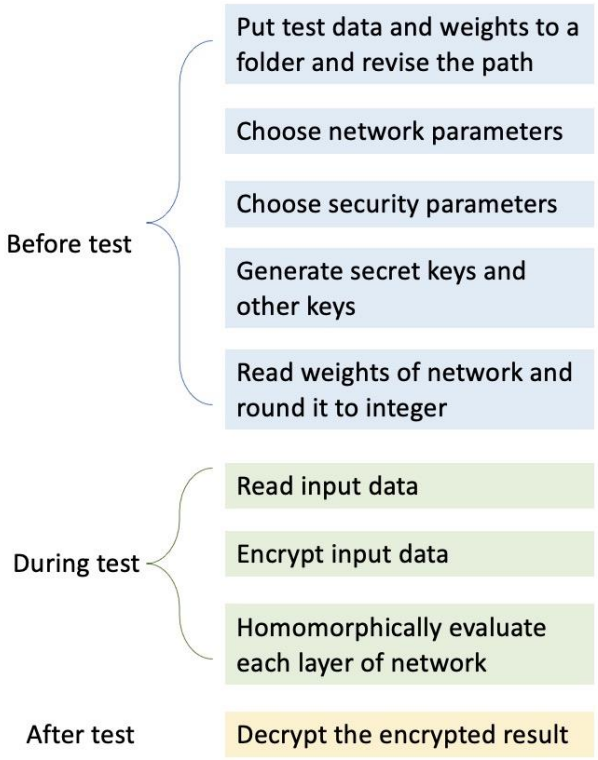
The last step is to decrypt the evaluation result and the corresponding index of the max value is the prediction result.

```
//decrypt
int64_t max = -1*mod;
int cindex = 0;

for (int i = 0; i < classes; ++i) {
    int64_t tempr = LWE64_Dec(q2, n2, ct_ip2[i], lwe_s);
    if (tempr > max) {
      max = tempr;
      cindex = i;
    }
  }
```

Applying P2N3 to your network

The flow of applying P2N3 FHE scheme to your network is shown as follows.



Putting test data and weights to a folder and revise the path

We recommend to make a new folder and name it "xxx_data" under our project. In your codes, please revise the path as "../xxx_data/file name" when you read the test data and weights. For example, in "MNIST.cpp", the path of test data is "../MNIST_data/t10k-images-idx3-ubyte" and the path of weights is "../MNIST_data/bp.txt".

Choosing network parameters

As our FHE scheme is based on integer, so we have to round the weights from floating numbers to integers. We have a scaler for each layer and can be set by users.

Regrading to the choice of scalers, it is decided by the size of the weights. If the weights are very small, then you need to a relatively large scaler to make the rounding results reasonable. For example, if most weights stay around 0.1-0.2, then the scaler should be larger than 10 to make the rounding results at least around 1-2.

Choosing security parameters

Security parameters are basically decided by the security level you want. For the modulus q_1 of LWE scheme, it should satisfy that $|output\ of\ each\ layer| < q_1/2$.

Although all parameters such as length of vector, modulus, degree of polynomial ring can be set by users, we recommend you try your default setting first, since we have verified that our parameters work well in many networks.

Generating secret keys and other keys

Generating LWE secret key is necessary for all networks. Generating RLWE secret key and evaluation keys are necessary when your network has an activation layer. If you wish to run our scheme in a client-server model, you can write such keys to client's PC and only pass the evaluation keys to the server. Check our examples for more details.

Reading weights of network and rounding it to integer

Before starting tests, the program needs to read the weights and round them to integer. We show how to read weights from a ".txt" files in our examples. If your weights are stored in other format, you need to write a function to read them.

The rounding is simple. We multiply the weight by scaler and take its integer part. If the floating part > 0.5, then we add 1 to integer part.

Reading input data

For each input, our program reads it and encrypts it by our LWE secret key. We show how to read weights from a ".txt" files in our examples. If your weights are stored in other format, you need to write a function to read them.

Encrypting the input data

For each dimension of input data, we encrypt it by our LWE encryption algorithm. For example, if the dimension of input is 784 (MNIST dataset), then we will generate 784 LWE ciphertexts. If you wish to run our scheme in a client-server model, you can store the ciphertexts in client's side and pass it to the server.

Homomorphically evaluating each layer of network

For linear evaluation layer, such as weighted sum and convolution, we call LWE-based homomorphic operations like homomorphic addition and homomorphic scaler multiplication. To calculate a weighted sum, we first homomorphically multiply the ciphertext and the corresponding weight. Then we add the ciphertexts together. Finally, we add the bias to the sum of ciphertext. The result is a LWE ciphertext. Check our examples for more details.

For non-linear evaluation layer, such as activation layer, we call homomorphic look-up table (LUT). The default activation function is ReLU. If you want to use other activation functions, you need to add it in "/include/RLWEscheme/LUT_64.hpp" first. If it is not the last non-linear layer in your network, key switching and rounding are needed to re-format the ciphertext. Check our test examples for details of key switching.

Decrypting the encrypted result

The last step is to decrypt the evaluation result and see whether it is correct. We call LWE decryption algorithm in this step. If you want to run our scheme in a client-server model, you can ask the server to return this encrypted result to the client. The client who has the secret key can decrypt it and see the result. The server without the secret key cannot decrypt it anymore.

Random number generator

In this section, we introduce the functions we provided to generate samples from different distributions.

Bernoulli generator

Generate a Bernoulli distribution, i.e., generate $\{0,1\}^n$ with $\Pr[x_i = 0] = 0.5$, and $\Pr[x_i = 1] = 0.5$.

- Function: gen_bernoulli
- Inputs:

  int *out: store the generated random numbers.

  unsigned int len_out: length of random number.

  unsigned char *seed: seed of random numbers.

- Output: An integer to indicate whether it is generated correctly.

Output integer	Error message
0	The algorithm runs correctly
1	The output array is empty
2	The input of len_out is not correct. (For simplicity, we require that 8|len_out)

Ternary generator

Generate a Ternary distribution, i.e., generate $\{0,1,-1\}^n$ with $\Pr[x_i = 0] = 0.5$ and $\Pr[x_i = 1] = \Pr[x_i = -1] = 0.25$.

- Function: gen_ternary
- Inputs:

  int *out: store the generated random numbers.

  unsigned int len_out: length of random number.

  unsigned char *seed: seed of random numbers.

- Output: An integer to indicate whether it is generated correctly.

Output integer	Error message
0	The algorithm runs correctly
1	The output array is empty
2	The input of len_out is not correct. (For simplicity, we require that 8|len_out)

Ternary generator with given variance

Generate a Ternary distribution with given variance $\sigma = 2^{-k}, k \geq 1$.

- Function: gen_ternary_var
- Inputs:

  int *out: store the generated random numbers.

  unsigned int len_out: length of random number.

  Unsigned int k: variance of distribution.

  unsigned char *seed: seed of random numbers.

- Output: An integer to indicate whether it is generated correctly.

Output integer	Error message
0	The algorithm runs correctly
1	The output array is empty
2	The input of len_out is not correct. (For simplicity, we require that 8|len_out)

Discrete uniform generator

Generate a uniform distribution of discrete values on Z_p. We offer two functions for different size of the integer p.

- Function: gen_uniform
- Inputs:

  int *out: store the generated random numbers.

  unsigned int len_out: length of random number.

  Int p: modolus.

  unsigned char *seed: seed of random numbers.

- Output: An integer to indicate whether it is generated correctly.

Output integer	Error message
0	The algorithm runs correctly
1	The output array is empty
2	The input of p is not correct. (For simplicity, we require that p=2^k)

- Function: gen_uniform_int64
- Inputs:

  int *out: store the generated random numbers.

  unsigned int len_out: length of random number.

  Int p: modolus.

  Unsigned int k: k=ceil(log(p)).

  unsigned char *seed: seed of random numbers.

- Output: An integer to indicate whether it is generated correctly.

Output integer	Error message
0	The algorithm runs correctly
1	The output array is empty

Polynomial ring

In this section, we introduce the operations we implemented in a polynomial ring $Z_q[x]/(x^n + 1)$.

In our implementation, a polynomial p is represented as an array of size (deg(p) + 2): $(\deg(p)$, $coef_0(p)$, $coef_1(p), coef_2(p) \dots, coef_{\deg(p)}(p))$. For example, polynomial $p(x) = 3x^3 + 2x^2 + x + 4$ is stored as $(3,4,1,2,3)$.

Polynomial addition

Calculate addition between two polynomials in a polynomial ring.

- Function: add_poly
- Inputs:

  Polynomial a: left input of addition and the result after addition.

  Polynomial b: right input of addition.

  Int64_t q: modulus of the polynomial ring.

- Output: Void.

Polynomial scaling

Scale a polynomial with scaler t, i.e., calculate $t * a$ in a polynomial ring, where a is a polynomial.

- Function: multi_scale _poly
- Inputs:

  Int64_t t: scaler.

  Polynomial a: left input of addition and the result after addition.

  Int64_t q: modulus of the polynomial ring.

- Output: Void.

Polynomial multiplication

Calculate multiplication between two polynomials in a polynomial ring.

- Function: multi_ poly_512
- Comment: polynomial multiplication in polynomial ring with degree 512.
- Inputs:

  Polynomial a: left input of multiplication.

  Polynomial b: right input of multiplication.

  Int n: degree of the polynomial ring.

  Int64_t q: modulus of the polynomial ring.

- Output: A polynomial to store the result after multiplication.

- Function: multi_ poly_1024
- Comment: polynomial multiplication in polynomial ring with degree 1024.
- Inputs:

  Polynomial a: left input of multiplication.

Polynomial b: right input of multiplication.

Int n: degree of the polynomial ring.

Int64_t q: modulus of the polynomial ring.

- Output: A polynomial to store the result after multiplication.

- Function: multi_ poly_2048
- Comment: polynomial multiplication in polynomial ring with degree 2048.
- Inputs:

  Polynomial a: left input of multiplication.

  Polynomial b: right input of multiplication.

  Int n: degree of the polynomial ring.

  Int64_t q: modulus of the polynomial ring.

- Output: A polynomial to store the result after multiplication.

- Function: multi_ poly_4096
- Comment: polynomial multiplication in polynomial ring with degree 4096.
- Inputs:

  Polynomial a: left input of multiplication.

  Polynomial b: right input of multiplication.

  Int n: degree of the polynomial ring.

  Int64_t q: modulus of the polynomial ring.

- Output: A polynomial to store the result after multiplication.

Polynomial decomposition

Decompose one polynomial to m polynomials according to decomposition base b. For example, if the modulus of polynomial ring is 2^{60} and the input decomposition base is 2^{30}, then the algorithm will decompose the input polynomial p into 2 polynomials whose coefficients are $coef(p)\ mod\ 2^{30}$ and $\lfloor coef(p)/2^{30} \rfloor$.

- Function: bit_poly
- Inputs:

  Int k: k=ceil(log(q)).

  Polynomial p: input polynomial.

  Int64_t q: modulus of the polynomial ring.

  Int64_t b: decomposition base.

  Int logb: logb=ceil(log(b)).

- Output: A vector of k/logb polynomials to store the result after decomposition.

Print a polynomial

Print a polynomial in the form: $coef_n(p)x^n + coef_{n-1}(p)x^{n-1} + \cdots + coef_1(p)x + coef_0(p)$.

- Function: print_polynomial
- Inputs: a polynomial
- Output: void

LWE Scheme

In this section, we introduce the algorithms of our approximate additive homomorphic encryption scheme based on LWE assumption, and homomorphic operations.

LWE_32 Scheme

LWE scheme with c++ type int range, i.e., 32 bits signed integer.

Key generation algorithm

- Function: extern int* LWE32_KeyGen(int n)
- Description: The LWE secret key generation algorithm that returns a key with dimension n.
- Input:

  n: type int, is the dimension of secret key.

- Output:

  A (type int) array with length n.

Encryption algorithm

- Function: extern vector<int> LWE32_Enc(int q, int n, int k, int m, const int* x)
- Description: The LWE encryption algorithm with specific parameters, and returns the ciphertext of plaintext m.
- Inputs:

  q: type int, the modulus of the LWE scheme.

  n: type int, the dimension of the LWE scheme.

  k: type int, the variance parameter of error used in the LWE scheme. The variance of error is 2^{-k}.

  m: type int, the plaintext.

  x: type const int*, the secret key of the LWE scheme.

- Output:

  The LWE ciphertext $(\vec{a}, b)$ of plaintext m. It is an int vector with length (n+1).

Decryption algorithm

- Function: extern int LWE32_Dec(int q, int n, const vector<int>& c, const int* x)

- Description: The LWE decryption algorithm with specific parameters, and returns the decryption of ciphertext c.
- Inputs:

  q: type int, the modulus of the LWE scheme.

  n: type int, the dimension of the LWE scheme.

  c: type const vector<int>&, a LWE ciphertext.

  x: type const int*, the secret key of the LWE scheme.

- Output:

  The decryption result, which has type int.

Homomorphic addition between two ciphertexts

- Function: extern vector<int> LWE32_Add_ct(int q, int n, const vector<int>& ct1, const vector<int>& ct2)
- Description: Homomorphically evaluate addition between two LWE ciphertexts. Suppose the two ciphertexts are ct1 and ct2, and they are the ciphertext of two integers m1 and m2, respectively. The homomorphically evaluation of addition should be a ciphertext ct, and the decryption of ct equals to m1+m2 (mod q).
- Inputs:

  q: type int, the modulus of the LWE scheme.

  n: type int, the dimension of the LWE scheme.

  ct1: type const vector<int>&, a LWE ciphertext of some integer m1.

  ct2: type const vector<int>&, a LWE ciphertext of some integer m2.

- Output:

  A LWE ciphertext of integer m1+m2 (mod q).

Homomorphic addition between plaintext and ciphertext

- Function: extern vector<int> LWE32_Plain_Add_ct(int q, int n, const vector<int>& ct1, int k)
- Description: Homomorphically evaluate addition between one LWE ciphertext and one plaintext integer. Suppose the LWE ciphertext is ct1, and is the ciphertext of integers m1. Suppose the plaintext integer is k. The homomorphically evaluation of addition should be a ciphertext ct, and the decryption of ct equals to m1+k (mod q).
- Inputs:

  q: type int, the modulus of the LWE scheme.

  n: type int, the dimension of the LWE scheme.

  ct1: type const vector<int>&, a LWE ciphertext of some integer m1.

  k: type int, a plaintext integer.

- Output:

  A LWE ciphertext of integer m1+k (mod q).

Homomorphic multiplication between plaintext and ciphertext

- Function: extern vector<int> LWE32_Plain_Multi_ct(int q, int n, const vector<int>& ct1, int k)
- Description: Homomorphically evaluate multiplication between one LWE ciphertext and one plaintext integer. Suppose the LWE ciphertext is ct1, and is the ciphertext of integers m1. Suppose the plaintext integer is k. The homomorphically evaluation of multiplication should be a ciphertext ct, and the decryption of ct equals to m1*k (mod q).
- Inputs:

  q: type int, the modulus of the LWE scheme.

  n: type int, the dimension of the LWE scheme.

  ct1: type const vector<int>&, a LWE ciphertext of some integer m1.

  k: type int, a plaintext integer.

- Output:

  A LWE ciphertext of integer m1*k (mod q).

Homomorphic rounding

- Function: vector<int> LWE32_Rounding(int q1, int q2, int n, const vector<int>& ct)
- Description: Switch the modulus of input LWE ciphertext to another one.
- Inputs:

  q1: type int, the old modulus of the input LWE ciphertext ct.

  q2: type int, the new modulus of the output LWE ciphertext, and we must have q2 | q1.

  n: type int, the dimension of the LWE scheme.

  ct: type const vector<int>&, a LWE ciphertext of some integer m with modulus q1.

- Output:

  A LWE ciphertext of integer m with modulus q2.

LWE_64 Scheme

LWE scheme with c++ type int64_t range, i.e., 64 bits signed integer.

Key generation algorithm

- Function: extern int* LWE64_KeyGen(int n)
- Description: The LWE secret key generation algorithm that returns a key with dimension n.
- Input:

  n: type int, is the dimension of secret key.

- Output:

A (type int) array with length n.

Encryption algorithm

- Function: extern vector<int64_t> LWE64_Enc(int64_t q, int n, int k, int64_t m, const int* x)
- Description: The LWE encryption algorithm with specific parameters, and returns the ciphertext of plaintext m.
- Inputs:

  q: type int64_t, the modulus of the LWE scheme.

  n: type int, the dimension of the LWE scheme.

  k: type int, the variance parameter of error used in the LWE scheme. The variance of error is 2^{-k}.

  m: type int64_t, the plaintext.

  x: type const int*, the secret key of the LWE scheme.

- Output:

  The LWE ciphertext $(\vec{a}, b)$ of plaintext m. It is an int64_t vector with length (n+1).

Decryption algorithm

- Function: extern int64_t LWE64_Dec(int64_t q, int n, const vector<int64_t>& c, const int* x)
- Description: The LWE decryption algorithm with specific parameters, and returns the decryption of ciphertext c.
- Inputs:

  q: type int64_t, the modulus of the LWE scheme.

  n: type int, the dimension of the LWE scheme.

  c: type const vector< int64_t >&, a LWE ciphertext.

  x: type const int*, the secret key of the LWE scheme.

- Output:

  The decryption result, which has type int64_t.

Homomorphic addition between two ciphertexts

- Function: vector<int64_t> LWE64_Add_ct(int64_t q, int n, const vector<int64_t>& ct1, const vector<int64_t>& ct2)
- Description: Homomorphically evaluate addition between two LWE ciphertexts. Suppose the two ciphertexts are ct1 and ct2, and they are the ciphertext of two integers m1 and m2, respectively. The homomorphically evaluation of addition should be a ciphertext ct, and the decryption of ct equals to m1+m2 (mod q).
- Inputs:

  q: type int64_t, the modulus of the LWE scheme.

n: type int, the dimension of the LWE scheme.

ct1: type const vector< int64_t>&, a LWE ciphertext of some integer m1.

ct2: type const vector< int64_t>&, a LWE ciphertext of some integer m2.

- Output:

  A LWE ciphertext of integer m1+m2 (mod q).

Homomorphic addition between plaintext and ciphertext

- Function: vector<int64_t> LWE64_Plain_Add_ct(int64_t q, int n, const vector<int64_t>& ct1, int64_t k)
- Description: Homomorphically evaluate addition between one LWE ciphertext and one plaintext integer. Suppose the LWE ciphertext is ct1, and is the ciphertext of integers m1. Suppose the plaintext integer is k. The homomorphically evaluation of addition should be a ciphertext ct, and the decryption of ct equals to m1+k (mod q).
- Inputs:

  q: type int64_t, the modulus of the LWE scheme.

  n: type int, the dimension of the LWE scheme.

  ct1: type const vector< int64_t>&, a LWE ciphertext of some integer m1.

  k: type int64_t, a plaintext integer.

- Output:

  A LWE ciphertext of integer m1+k (mod q).

Homomorphic multiplication between plaintext and ciphertext

- Function: vector<int64_t> LWE64_Plain_Multi_ct(int64_t q, int n, const vector<int64_t>& ct1, int64_t k)
- Description: Homomorphically evaluate multiplication between one LWE ciphertext and one plaintext integer. Suppose the LWE ciphertext is ct1, and is the ciphertext of integers m1. Suppose the plaintext integer is k. The homomorphically evaluation of multiplication should be a ciphertext ct, and the decryption of ct equals to m1*k (mod q).
- Inputs:

  q: type int64_t, the modulus of the LWE scheme.

  n: type int, the dimension of the LWE scheme.

  ct1: type const vector< int64_t >&, a LWE ciphertext of some integer m1.

  k: type int64_t, a plaintext integer.

- Output:

  A LWE ciphertext of integer m1*k (mod q).

Homomorphic rounding

- Function: vector<int64_t> LWE64_Rounding(int64_t q1, int64_t q2, int n, const vector<int64_t>& ct)
- Description: Switch the modulus of input LWE ciphertext to another one.
- Inputs:

  q1: type int64_t, the old modulus of the input LWE ciphertext ct.

  q2: type int64_t, the new modulus of the output LWE ciphertext, and we must have q2 | q1.

  n: type int, the dimension of the LWE scheme.

  ct: type const vector<int64_t>&, a LWE ciphertext of some integer m with modulus q1.

- Output:

  A LWE ciphertext of integer m with modulus q2.

RLWE Scheme

RLWE_64 Scheme

Ring LWE scheme with c++ type int64_t range, i.e., 64 bits signed integer. The scheme is based on polynomial ring with int64_t modulus.

typedef vector<int64_t> polynomial;

Description: We define polynomial to be an int64_t vector.

Key Generation algorithm

- Function: extern polynomial RLWE64_KeyGen(int n)
- Description: The RLWE secret key generation algorithm that returns a secret key, which is a polynomial with degree n-1.
- Input:

  n: type int, the degree parameter of RLWE scheme.

- Output:

  A degree-(n-1) polynomial whose coefficients are from Uniform Random Distribution on {-1,0,1}.

Encryption algorithm

- Function: extern vector<polynomial> RLWE64_Enc_2048(int n, int64_t q, int k, const polynomial & m, const polynomial & s)
- Description: The RLWE encryption algorithm with specific parameters, and returns the ciphertext of plaintext m. This function is used for degree-2048 polynomial ring.
- Inputs:

  n: type int, the degree parameter of RLWE scheme.

  q: type int64_t, the modulus of the polynomial ring.

k: type int, the variance parameter of error used in the RLWE scheme. The variance of error is 2^{-k}.

m: type const polynomial &, the plaintext to be encrypted.

s: type const polynomial &, the secret key of RLWE scheme.

- Output:

  The RLWE ciphertext (a, b) of plaintext m. a, b are two degree-(n-1) polynomials, with modulus q.

- Function: extern vector<polynomial> RLWE64_Enc_512(int n, int64_t q, int k, const polynomial & m, const polynomial & s)
- Description: The RLWE encryption algorithm with specific parameters, and returns the ciphertext of plaintext m. This function is used for degree-512 polynomial ring.
- Inputs:

  n: type int, the degree parameter of RLWE scheme.

  q: type int64_t, the modulus of the polynomial ring.

  k: type int, the variance parameter of error used in the RLWE scheme. The variance of error is 2^{-k}.

  m: type const polynomial &, the plaintext to be encrypted.

  s: type const polynomial &, the secret key of RLWE scheme.

- Output:

  The RLWE ciphertext (a, b) of plaintext m. a, b are two degree-(n-1) polynomials, with modulus q.

Decryption algorithm

- Function: extern polynomial RLWE64_Dec_2048(int n, int64_t q, const polynomial & s, const vector<polynomial> & ct)
- Description: The RLWE decryption algorithm with specific parameters, and returns the decryption of ciphertext ct. This function is used for degree-2048 polynomial ring.
- Inputs:

  n: type int, the degree parameter of RLWE scheme.

  q: type int64_t, the modulus of the polynomial ring.

  s: type const polynomial &, the secret key of RLWE scheme.

  ct: type const vector<polynomial> &, a RLWE ciphertext.

- Output:

  The decryption result, which has type polynomial.

- Function: extern polynomial RLWE64_Dec_512 (int n, int64_t q, const polynomial & s, const vector<polynomial> & ct)
- Description: The RLWE decryption algorithm with specific parameters, and returns the decryption of ciphertext ct. This function is used for degree-512 polynomial ring.
- Inputs:

  n: type int, the degree parameter of RLWE scheme.

  q: type int64_t, the modulus of the polynomial ring.

  s: type const polynomial &, the secret key of RLWE scheme.

  ct: type const vector<polynomial> &, a RLWE ciphertext.

- Output:

  The decryption result, which has type polynomial.

RGSW ciphertext generation

- Function: vector<vector<polynomial>> RGSWct_2048(int n, int64_t q, int k, int var, const polynomial & m, const polynomial & s, int64_t b, int logb)
- Description: The RGSW encryption algorithm with specific parameters, and returns the ciphertext of plaintext m. This function is used for degree-2047 polynomial ring.
- Inputs:

  n: type int, the degree parameter of RGSW scheme.

  q: type int64_t, the modulus of the polynomial ring.

  k: type int, k=log(q).

  var: type int, the variance parameter of error used in the RLWE scheme. The variance of error is 2^{-var}.

  m: type const polynomial &, the plaintext to be encrypted.

  s: type const polynomial &, the secret key.

  b: type int64_t, the gadget parameter of RGSW scheme.

  logb: type int, logb=log(b).

- Output:

  The RGSW ciphertext of polynomial m. The ciphertext has type vector<vector<polynomial>>.

Extended RLWE ciphertext multiplication

- Function: vector<polynomial> bit_then_multiply_2048(int n, int64_t q, int k, const polynomial & r, const vector<vector<polynomial>> & ct, int b, int logb)
- Description: Homomorphically evaluate polynomial multiplication between plaintext polynomial r, and an extended-RLWE ciphertext ct of some polynomial m. The evaluation result is a RLWE

ciphertext of polynomial r*m in polynomial ring $\mathbb{Z}_q/(x^n+1)$. This function is used for degree-2048 polynomial ring.

- Inputs:

  n: type int, the degree parameter of extended-RLWE scheme.

  q: type int64_t, the modulus of the polynomial ring.

  k: type int, k=log(q).

  r: type const polynomial &, the plaintext polynomial in the multiplication.

  ct: type const vector<vector<polynomial>>&, the extended-RLWE ciphertext of some polynomial m.

  b: type int64_t, the gadget parameter of extended-RLWE scheme.

  logb: type int, logb=log(b).

- Output:

  The RLWE ciphertext of polynomial r*m in polynomial ring $\mathbb{Z}_q/(x^n+1)$. The ciphertext has type vector<vector<polynomial>>.

- Function: vector<polynomial> bit_then_multiply_512(int n, int64_t q, int k, const polynomial & r, const vector<vector<polynomial>> & ct, int b, int logb)
- Description: Homomorphically evaluate polynomial multiplication between plaintext polynomial r, and an extended-RLWE ciphertext ct of some polynomial m. The evaluation result is a RLWE ciphertext of polynomial r*m in polynomial ring $\mathbb{Z}_q/(x^n+1)$. This function is used for degree-512 polynomial ring.
- Inputs:

  n: type int, the degree parameter of extended-RLWE scheme.

  q: type int64_t, the modulus of the polynomial ring.

  k: type int, k=log(q).

  r: type const polynomial &, the plaintext polynomial in the multiplication.

  ct: type const vector<vector<polynomial>>&, the extended-RLWE ciphertext of some polynomial m.

  b: type int64_t, the gadget parameter of extended-RLWE scheme.

  logb: type int, logb=log(b).

- Output:

  The RLWE ciphertext of polynomial r*m in polynomial ring $\mathbb{Z}_q/(x^n+1)$. The ciphertext has type vector<vector<polynomial>>.

External product

- Function: vector<polynomial> rlwe_multi_rgsw(int n, int64_t q, int k, const vector<polynomial> & rlwe_ct, const vector<vector<polynomial>> & rgsw_ct, int64_t b, int logb)

- Description: Homomorphically evaluate polynomial multiplication between RLWE ciphertext and RGSW ciphertext. Suppose the RLWE ciphertext is of plaintext polynomial m0, the RGSW ciphertext is of plaintext polynomial m. The evaluation result is a RLWE ciphertext of polynomial m0*m in polynomial ring $\mathbb{Z}_q/(x^n+1)$.
- Inputs:

  n: type int, the degree parameter of RLWE and RGSW scheme.

  q: type int64_t, the modulus of the polynomial ring.

  k: type int, k=log(q).

  rlwe_ct: type const vector<polynomial>&, a RLWE ciphertext of some polynomial m0.

  rgsw_ct: type const vector<vector<polynomial>>&, a RGSW ciphertext of some polynomial m.

  b: type int64_t, the gadget parameter of RGSW scheme.

  logb: type int, logb=log(b).

- Output:

  The RLWE ciphertext of polynomial m0*m in polynomial ring $\mathbb{Z}_q/(x^n+1)$. The ciphertext has type vector<vector<polynomial>>.

Extraction

- Function: extern vector<int64_t> Extract0(int64_t q, int64_t delta, int n, const vector<polynomial> & RLWE_ct)
- Description: Given input RLWE ciphertext of some polynomial m, the function returns the LWE ciphertext of the constant term of m.
- Inputs:

  q: type int64_t, the modulus of the polynomial ring.

  delta, type int64_t, the scale factor used in look-up algorithm.

  n: type int, the degree parameter of RLWE scheme.

  RLWE_ct: type const vector<polynomial>, RLWE ciphertext of some polynomial m.

- Output:

  The LWE ciphertext of the constant term of m, i.e., m(0). The output has type vector<int64_t>.

Key Switching Algorithms

Key switching key generation

- Function: extern vector<vector<vector<polynomial>>> LWE_ks_key(int n1, int n2, int64_t q, int k, int var, const int* x, const polynomial& s, int64_t b, int logb)
- Description: Generate a key-switching key from dimension n1 to dimension n2. The key-switching key some extended-RLWE ciphertexts.
- Inputs:

n1: type int, the old dimension of LWE ciphertext.

n2: type int, the new dimension of LWE ciphertext.

q: type int64_t, the modulus.

k: type int, k=log(q).

var: type int, the variance parameter of error used in the extended-RLWE scheme. The variance of error is 2^{-var}.

x: type const int*, the old secret key of LWE scheme.

s: type const polynomial &, the secret key of extended-RLWE scheme.

b: type int64_t, the gadget parameter of extended-RLWE scheme.

logb: type int, logb=log(b).

- Output:

  The key-switching key. It has type vector<vector<vector<polynomial>>>.

Key switching algorithm

- Function: extern vector<int64_t> LWE_ks(int n1, int n2, int64_t q, int k, const vector<int64_t>& lwe_ct, const vector<vector<vector<polynomial>>>& ks_key, int64_t b, int logb)
- Description: This function turns a LWE ciphertext lwe_ct with dimension n1 into another LWE ciphertext with dimension n2, and with a new LWE secret key.
- Inputs:

  n1: type int, the old dimension of LWE ciphertext.

  n2: type int, the new dimension of LWE ciphertext.

  q: type int64_t, the modulus.

  k: type int, k=log(q).

  lwe_ct: type const vector<int64_t>, the old LWE ciphertext of some plaintext m.

  ks_key: type vector<vector<vector<polynomial>>>, the key-switching key generated by LWE_ks_key().

  b: type int64_t, the gadget parameter of extended-RLWE scheme.

  logb: type int, logb=log(b).

- Output:

  A LWE ciphertext of plaintext m, with dimension n2.

Look-up Table Algorithm

template<int NN, int qNum, enum NTTSelector NTT, enum ImplSelector Impl>

- function: extern vector<polynomial> LUT_2048(int index, int q1, int64_t q, int n, int64_t delta, int k, int N, vector<int> const &ct_LWE, const vector<vector<Rq<NN,qNum,NTT,Impl>>> &_ek0,

const vector<vector<Rq<NN,qNum,NTT,Impl>>> &_ek1, const vector<vector<Rq<NN,qNum,NTT,Impl>>> &_ek2, polynomial const& f, int64_t b, int logb)

- Description: Evaluate the look-up table algorithms with input LWE ciphertext. This function is for LUT algorithm used in neural network nodes. This function is used for degree-2048 polynomial ring.
-
- Inputs:

  index: type int, the index of the node evaluating in the current layer of neural network.

  q1: type int, the input LWE ciphertext modulus.

  q: type int64_t, the output LWE ciphertext modulus.

  n: type int, the input LWE ciphertext dimension.

  delta: type int64_t, the scale parameter in LUT algorithm.

  k: type int, k=log(q).

  N: type int, the output ciphertext dimension.

  ct_LWE: type vector <int> const &, the input LWE ciphertext of some plaintext integer m.

  _ek0, _ek1, _ek2: type const vector<vector<Rq<NN,qNum,NTT,Impl>>> &, the LUT evaluation keys. They are RGSW ciphertext.

  f: type polynomial const&, the polynomial encoding of the evaluated function $F(m)$ in LUT algorithm.

  b: type int64_t, the gadget parameter of RGSW scheme.

  logb: type int, logb=log(b).

- Output:

  A RLWE ciphertext of f*$X^{F(m)}$. It has type vector<polynomial>.