RLWE Homomorphic Encryption Scheme

Aidan (@CodeByAidan)

June 6, 2024

1 Introduction

This document provides a mathematical explanation of the RLWE (Ring Learning With Errors) homomorphic encryption scheme. The provided Python code demonstrates key generation, encryption, and various operations on encrypted data.

See more:

- "Somewhat Practical Fully Homomorphic",
- "Homomorphic Encryption: a Toy Implementation in Python"

2 RLWE Encryption Parameters

The RLWE encryption scheme operates in a polynomial ring modulo a polynomial modulus *poly_mod*. The ciphertext modulus is denoted as q, and the plaintext modulus is denoted as t. The encryption scheme uses a base T for polynomial switching and a modulus p for modulus switching.

2.1 Parameters

The following parameters are defined:

- n: Polynomial modulus degree
- q: Ciphertext modulus
- t: Plaintext modulus
- T: Base for polynomial switching
- p: Modulus for modulus switching

3 Key Generation

The key generation process involves generating a public key (a, b) and a secret key s. The public key is used for encryption, and the secret key is used for decryption.

3.1 Generating Secret and Public Keys

The secret key s is a binary polynomial. The public key is generated as follows:

- s: Binary secret key polynomial
- a: Uniformly random polynomial
- e: Error polynomial sampled from a normal distribution
- b: $a \cdot s - e \mod q$

4 Encryption

To encrypt a plaintext message m using the public key (a, b), the following steps are performed:

$$\begin{aligned}
m &\equiv m \mod t \\
\text{delta} &= \frac{q}{t} \\
e1 &= \text{Error polynomial} \\
\text{scaled_m} &= \delta \cdot m \mod q \\
\text{new_ct} &\equiv \text{new_ct_0} \\
\text{new_ct_0} &= a \cdot u + e1 + \text{scaled_m} \mod q
\end{aligned}$$

5 Homomorphic Operations

The RLWE encryption scheme supports various homomorphic operations, such as addition and multiplication, on encrypted data.

5.1 Addition of Ciphertext and Plaintext

To add a ciphertext ct and a plaintext pt, the following operation is performed:

$$
\begin{aligned}
m &\equiv \text{pt} \mod t \\
\text{delta} &= \frac{q}{t} \\
e1 &= \text{Error polynomial} \\
\text{scaled\_m} &= \delta \cdot m \mod q \\
\text{new\_ct} &\equiv \text{new\_ct\_0} \\
\text{new\_ct\_0} &= ct_0 + \text{scaled\_m} \mod q
\end{aligned}
$$

5.2 Addition of Ciphertexts

To add two ciphertexts $ct1$ and $ct2$, the following operation is performed:

$$
\begin{aligned}
\text{new\_ct} &\equiv \text{new\_ct\_0} \\
\text{new\_ct\_0} &= ct1_0 + ct2_0 \mod q
\end{aligned}
$$

5.3 Multiplication of Ciphertext and Plaintext

To multiply a ciphertext ct by a plaintext pt, the following operation is performed:

$$
\begin{aligned}
\text{scaled\_m} &= \delta \cdot m \mod q \\
\text{new\_ct} &\equiv \text{new\_ct\_0} \\
\text{new\_ct\_0} &= ct_0 \cdot \text{scaled\_m} \mod q
\end{aligned}
$$

5.4 Multiplication of Ciphertexts

The multiplication of ciphertexts involves more complex steps and relinearization keys:

$$
\begin{aligned}
\text{new\_ct} &\equiv \text{new\_ct\_0} \\
\text{new\_ct\_0} &= ct1_0 \cdot ct2_0 + ct1_1 \cdot ct2_1 \mod q \\
\text{new\_ct\_1} &= ct1_0 \cdot ct2_1 + ct1_1 \cdot ct2_0 \mod q
\end{aligned}
$$

6 Decryption

To decrypt a ciphertext and obtain the original plaintext message, the following steps are performed:

$$\begin{aligned}
\text{scaled_pt} &= ct_0 \cdot sk + ct_0 \mod q \\
\text{decrypted_poly} &= \left\lfloor \frac{t \cdot \text{scaled_pt}}{q} \right\rceil \mod t \\
\text{Decrypted Plaintext} &= \text{decrypted_poly}
\end{aligned}$$

The RLWE homomorphic encryption scheme provides a powerful tool for performing computations on encrypted data while preserving data privacy. The scheme involves key generation, encryption, and various homomorphic operations, making it suitable for secure computations in various applications.

7 Python Code

Thank you to the authors of the original code:

- Mădălina Bolboceanu (https://github.com/mbolboceanu)
- Miruna Roșca (https://github.com/MirunaRosca)
- Radu Țițiu (https://github.com/rtitiu)

This code is an updated version of the original code, which can be found here:

- https://github.com/bit-ml/he-scheme

```python
"""
Thank you to the author of the original code:
    - Madalina Bolboceanu (@mbolboceanu)

This code is an updated version of the original code, which can be found here:
    - https://github.com/bit-ml/he-scheme
"""

import numpy as np
from numpy.polynomial import polynomial as poly

#------Functions for polynomial evaluations mod poly_mod only------
def polymul_wm(x, y, poly_mod):
    """Multiply two polynomials
    Args:
        x, y: two polynomials to be multiplied.
        poly_mod: polynomial modulus.
    Returns:
        A polynomial in Z[X]/(poly_mod).
    """
    return poly.polydiv(poly.polymul(x, y), poly_mod)[1]

def polyadd_wm(x, y, poly_mod):
    """Add two polynomials
        Args:
            x, y: two polynomials to be added.
            poly_mod: polynomial modulus.
        Returns:
            A polynomial in Z[X]/(poly_mod).
        """
    return poly.polydiv(poly.polyadd(x, y), poly_mod)[1]
#==============================================================

# ------Functions for polynomial evaluations both mod poly_mod and mod q------
def polymul(x, y, modulus, poly_mod):
    """Multiply two polynomials
    Args:
        x, y: two polynomials to be multiplied.
        modulus: coefficient modulus.
        poly_mod: polynomial modulus.
    Returns:
        A polynomial in Z_modulus[X]/(poly_mod).
    """
    return np.int64(
        np.round(poly.polydiv(poly.polymul(x, y) %
                              modulus, poly_mod)[1] % modulus)
```

```
    )


def polyadd(x, y, modulus, poly_mod):
    """Add two polynomials
    Args:
        x, y: two polynoms to be added.
        modulus: coefficient modulus.
        poly_mod: polynomial modulus.
    Returns:
        A polynomial in Z_modulus[X]/(poly_mod).
    """
    return np.int64(
        np.round(poly.polydiv(poly.polyadd(x, y) %
                              modulus, poly_mod)[1] % modulus)
    )
#==============================================================================

# ---------Functions for random polynomial generation---------
def gen_binary_poly(size):
    """Generates a polynomial with coeffecients in [0, 1]
    Args:
        size: number of coeffcients, size-1 being the degree of the
            polynomial.
    Returns:
        array of coefficients with the coeff[i] being
        the coeff of x ^ i.
    """
    return np.random.randint(0, 2, size, dtype=np.int64)


def gen_uniform_poly(size, modulus):
    """Generates a polynomial with coeffecients being integers in Z_modulus
    Args:
        size: number of coeffcients, size-1 being the degree of the
            polynomial.
    Returns:
        array of coefficients with the coeff[i] being
        the coeff of x ^ i.
    """
    return np.random.randint(0, modulus, size, dtype=np.int64)


def gen_normal_poly(size, mean, std):
    """Generates a polynomial with coeffecients in a normal distribution
    of mean mean and a standard deviation std, then discretize it.
```

```python
    Args:
        size: number of coeffcients, size-1 being the degree of the
            polynomial.
    Returns:
        array of coefficients with the coeff[i] being
        the coeff of x ^ i.
    """
    return np.int64(np.random.normal(mean, std, size=size))
#==============================================================================

# ------ Function for returning n's coefficients in base b ( lsb is on the lef
def int2base(n, b):
    """Generates the base decomposition of an integer n.
    Args:
        n: integer to be decomposed.
        b: base.
    Returns:
        array of coefficients from the base decomposition of n
        with the coeff[i] being the coeff of b ^ i.
    """
    return [n] if n < b else [n % b] + int2base(n // b, b)


# ----- Functions for keygen, encryption and decryption -----
def keygen(size, modulus, poly_mod, std1):
    """Generate a public and secret keys
    Args:
        size: size of the polynoms for the public and secret keys.
        modulus: coefficient modulus.
        poly_mod: polynomial modulus.
        std1: standard deviation of the error.
    Returns:
        Public and secret key.
    """
    s = gen_binary_poly(size)
    a = gen_uniform_poly(size, modulus)
    e = gen_normal_poly(size, 0, std1)
    b = polyadd(polymul(-a, s, modulus, poly_mod), -e, modulus, poly_mod)
    return (b, a), s


def evaluate_keygen_v1(sk, size, modulus, T, poly_mod, std2):
    """Generate a relinearization key using version 1.
        Args:
            sk: secret key.
            size: size of the polynomials.
```

```python
            modulus: coefficient modulus.
            T: base.
            poly_mod: polynomial modulus.
            std2: standard deviation for the error distribution.
        Returns:
            rlk0, rlk1: relinearization key.
        """
    n = len(poly_mod) - 1
    l = np.int64(np.log(modulus) / np.log(T))
    rlk0 = np.zeros((l + 1, n), dtype=np.int64)
    rlk1 = np.zeros((l + 1, n), dtype=np.int64)
    for i in range(l + 1):
        a = gen_uniform_poly(size, modulus)
        e = gen_normal_poly(size, 0, std2)
        secret_part = T ** i * poly.polymul(sk, sk)
        b = np.int64(polyadd(
        polymul_wm(-a, sk, poly_mod),
        polyadd_wm(-e, secret_part, poly_mod), modulus, poly_mod))

        b = np.int64(np.concatenate( (b, [0] * (n - len(b)) ) )) # pad b
        a = np.int64(np.concatenate( (a, [0] * (n - len(a)) ) )) # pad a

        rlk0[i] = b
        rlk1[i] = a
    return rlk0, rlk1

def evaluate_keygen_v2(sk, size, modulus, poly_mod, extra_modulus, std2):
    """Generate a relinearization key using version 2.
        Args:
            sk: secret key.
            size: size of the polynomials.
            modulus: coefficient modulus.
            poly_mod: polynomial modulus.
            extra_modulus: the "p" modulus for modulus switching.
            st2: standard deviation for the error distribution.
        Returns:
            rlk0, rlk1: relinearization key.
        """
    new_modulus = modulus * extra_modulus
    a = gen_uniform_poly(size, new_modulus)
    e = gen_normal_poly(size, 0, std2)
    secret_part = extra_modulus * poly.polymul(sk, sk)

    b = np.int64(polyadd_wm(
        polymul_wm(-a, sk, poly_mod),
        polyadd_wm(-e, secret_part, poly_mod), poly_mod)) % new_modulus
```

```python
    return b, a

def encrypt(pk, size, q, t, poly_mod, m, std1):
    """Encrypt an integer.
    Args:
        pk: public-key.
        size: size of polynomials.
        q: ciphertext modulus.
        t: plaintext modulus.
        poly_mod: polynomial modulus.
        m: plaintext message, as an integer vector (of length <= size) with entr
    Returns:
        Tuple representing a ciphertext.
    """
    m = np.array(m + [0] * (size - len(m)), dtype=np.int64) % t
    delta = q // t
    scaled_m = delta * m
    e1 = gen_normal_poly(size, 0, std1)
    e2 = gen_normal_poly(size, 0, std1)
    u = gen_binary_poly(size)
    ct0 = polyadd(
        polyadd(
            polymul(pk[0], u, q, poly_mod),
            e1, q, poly_mod),
        scaled_m, q, poly_mod
    )
    ct1 = polyadd(
        polymul(pk[1], u, q, poly_mod),
        e2, q, poly_mod
    )
    return (ct0, ct1)


def decrypt(sk, size, q, t, poly_mod, ct):
    """Decrypt a ciphertext.
    Args:
        sk: secret-key.
        size: size of polynomials.
        q: ciphertext modulus.
        t: plaintext modulus.
        poly_mod: polynomial modulus.
        ct: ciphertext.
    Returns:
        Integer vector representing the plaintext.
    """
    scaled_pt = polyadd(
```

```python
        polymul(ct[1], sk, q, poly_mod),
        ct[0], q, poly_mod
    )
    decrypted_poly = np.round(t * scaled_pt / q) % t
    decrypted_poly_1 = list(decrypted_poly)
    bound = len(decrypted_poly_1)
    if bound<size:
        number_of_zeros_to_pad = size-bound
        list_to_append = [0] * number_of_zeros_to_pad
        decrypted_poly_to_return = np.append(decrypted_poly, list_to_append) #pa
    else:
        decrypted_poly_to_return = decrypted_poly
    return np.int64(decrypted_poly_to_return)

#==================================================================


# ---------Function for adding and multiplying encrypted values---------
def add_plain(ct, pt, q, t, poly_mod):
    """Add a ciphertext and a plaintext.
    Args:
        ct: ciphertext.
        pt: integer to add.
        q: ciphertext modulus.
        t: plaintext modulus.
        poly_mod: polynomial modulus.
    Returns:
        Tuple representing a ciphertext.
    """
    size = len(poly_mod) - 1
    # encode the integer into a plaintext polynomial
    m = np.array(pt + [0] * (size - len(pt)), dtype=np.int64) % t
    delta = q // t
    scaled_m = delta * m
    new_ct0 = polyadd(ct[0], scaled_m, q, poly_mod)
    return (new_ct0, ct[1])


def add_cipher(ct1, ct2, q, poly_mod):
    """Add a ciphertext and a ciphertext.
    Args:
        ct1, ct2: ciphertexts.
        q: ciphertext modulus.
        poly_mod: polynomial modulus.
    Returns:
        Tuple representing a ciphertext.
```

```python
    """
    new_ct0 = polyadd(ct1[0], ct2[0], q, poly_mod)
    new_ct1 = polyadd(ct1[1], ct2[1], q, poly_mod)
    return (new_ct0, new_ct1)


def mul_plain(ct, pt, q, t, poly_mod):
    """Multiply a ciphertext and a plaintext.
    Args:
        ct: ciphertext.
        pt: integer polynomial to multiply.
        q: ciphertext modulus.
        t: plaintext modulus.
        poly_mod: polynomial modulus.
    Returns:
        Tuple representing a ciphertext.
    """
    size = len(poly_mod) - 1
    # encode the integer polynomial into a plaintext vector of size=size
    m = np.array(pt + [0] * (size - len(pt)), dtype=np.int64) % t
    new_c0 = polymul(ct[0], m, q, poly_mod)
    new_c1 = polymul(ct[1], m, q, poly_mod)
    return (new_c0, new_c1)

def multiplication_coeffs(ct1, ct2, q, t, poly_mod):
    """Multiply two ciphertexts.
        Args:
            ct1: first ciphertext.
            ct2: second ciphertext
            q: ciphertext modulus.
            t: plaintext modulus.
            poly_mod: polynomial modulus.
        Returns:
            Triplet (c0,c1,c2) encoding the multiplied ciphertexts.
        """

    c_0 = np.int64(np.round(polymul_wm(ct1[0], ct2[0], poly_mod) * t / q)) % q
    c_1 = np.int64(np.round(polyadd_wm(polymul_wm(ct1[0], ct2[1], poly_mod), poly
    c_2 = np.int64(np.round(polymul_wm(ct1[1], ct2[1], poly_mod) * t / q)) % q
    return c_0, c_1, c_2


def mul_cipher_v1(ct1, ct2, q, t, T, poly_mod , rlk0, rlk1):
    """Multiply two ciphertexts.
    Args:
        ct1: first ciphertext.
```

```
        ct2: second ciphertext
        q: ciphertext modulus.
        t: plaintext modulus.
        T: base
        poly_mod: polynomial modulus.
        rlk0, rlk1: output of the EvaluateKeygen_v1 function.
    Returns:
        Tuple representing a ciphertext.
    """
    n = len(poly_mod) - 1
    l = np.int64(np.log(q) / np.log(T))  #l = log_T(q)

    c_0, c_1, c_2 = multiplication_coeffs(ct1, ct2, q, t, poly_mod)
    c_2 = np.int64(np.concatenate( (c_2, [0] * (n - len(c_2))) )) #pad

    #Next, we decompose c_2 in base T:
    #more precisely, each coefficient of c_2 is decomposed in base T such that c
    Reps = np.zeros((n, l + 1), dtype = np.int64)
    for i in range(n):
        rep = int2base(c_2[i], T)
        rep2 = rep + [0] * (l + 1 - len(rep)) #pad with 0
        Reps[i] = np.array(rep2, dtype=np.int64)
    # Each row Reps[i] is the base T representation of the i-th coefficient c_2[
    # The polynomials c_2(j) are given by the columns Reps[:,j].

    c_20 = np.zeros(shape=n)
    c_21 = np.zeros(shape=n)
    # Here we compute the sums: rlk[j][0] * c_2(j) and rlk[j][1] * c_2(j)
    for j in range(l + 1):
        c_20 = polyadd_wm(c_20, polymul_wm(rlk0[j], Reps[:,j], poly_mod), poly_m
        c_21 = polyadd_wm(c_21, polymul_wm(rlk1[j], Reps[:,j], poly_mod), poly_m

    c_20 = np.int64(np.round(c_20)) % q
    c_21 = np.int64(np.round(c_21)) % q

    new_c0 = np.int64(polyadd_wm(c_0, c_20, poly_mod)) % q
    new_c1 = np.int64(polyadd_wm(c_1, c_21, poly_mod)) % q

    return (new_c0, new_c1)

def mul_cipher_v2(ct1, ct2, q, t, p, poly_mod, rlk0, rlk1):
    """Multiply two ciphertexts.
    Args:
        ct1: first ciphertext.
        ct2: second ciphertext.
        q: ciphertext modulus.
```

```
        t: plaintext modulus.
        p: modulus-swithcing modulus.
        poly_mod: polynomial modulus.
        rlk0, rlk1: output of the EvaluateKeygen_v2 function.
    Returns:
        Tuple representing a ciphertext.
    """
    c_0, c_1, c_2 = multiplication_coeffs(ct1, ct2, q, t, poly_mod)

    c_20 = np.int64(np.round(polymul_wm(c_2, rlk0, poly_mod) / p)) % q
    c_21 = np.int64(np.round(polymul_wm(c_2, rlk1, poly_mod) / p)) % q

    new_c0 = np.int64(polyadd_wm(c_0, c_20, poly_mod)) % q
    new_c1 = np.int64(polyadd_wm(c_1, c_21, poly_mod)) % q
    return (new_c0, new_c1)
#==============================================================
```