Project Report

On

Computation on Homomorphically Encrypted Data

Prepared in fulfilment of

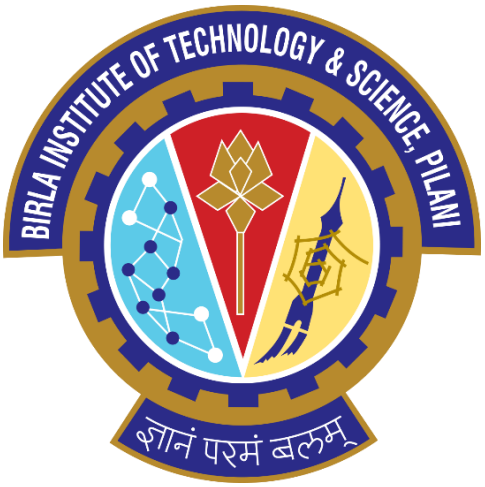
Cryptography (BITS-F463)

Submitted By:

Ishan Rai - 2017B4A20596P

Under the Guidance of

Dr. Ashutosh Bhatia



BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

18th December 2020

Table of Contents

Acknowledgement

I want to express my sincere gratitude to Dr. Ashutosh Bhatia for giving me the opportunity to work on this project and for being a constant source of support. His approach to teaching cryptography and emphasizing the modern cryptography applications motivated me to explore this field of privacy-preserving computation on data. Throughout this project, I have learned a lot about Ring theory and homomorphisms in Algebra, Homomorphic Encryption, and its implementations from scratch in Python.

I also want to extend my gratitude to the Department of Mathematics, BITS Pilani, and Department of Computer Science, BITS Pilani for letting me take up this course and pursue this insightful project.

Introduction

In today's modern era, machine learning algorithms have found a place in almost every industry and significantly impacted our lives. These algorithms are data-centric, i.e., they rely on the data that we provide as input. With the emergence of machine learning as a service, data acquisition is turning out to be one of the biggest challenges for companies providing model inference as a service. Domains that deal with sensitive data like the health and finance sector cannot be given access to sensitive information due to regulatory constraints in many countries. Stringent data privacy laws and the emergence of machine learning have led to recent computation developments on encrypted data. When extended to machine learning algorithms, these computations have come to be known as Privacy-preserving Machine Learning.

Homomorphic Encryption allows computation on encrypted data and is considered as one of the three pillars of privacy-preserving machine learning, with the others being Federated Learning and Differential Privacy. This project aims at implementing a Homomorphic Encryption scheme using the basic NumPy library in Python.

Homomorphic Encryption: The History

Homomorphic Encryption is a mode of encryption that allows one to perform computations on encrypted data without decrypting it. The output of the computation is in an encrypted form. After decryption, the output is the same as if the operations had been performed on unencrypted data. Homomorphic encryption schemes have been developed using various approaches. Fully homomorphic encryption schemes are usually classified into generations corresponding to the underlying methodology used to formulate the Encryption scheme.

- **Pre-FHE:**
  In 1978, within a year of the RSA scheme being published, the problem of constructing a fully homomorphic encryption scheme was first proposed. Ron Rivest, Len Adleman, and Michael Dertouzos published a report called "On Data Banks and Privacy Homomorphisms." The paper detailed how a loan company, for example, could use a cloud provider (then known as a commercial time-sharing service) to store and compute encrypted data. This influential paper led to the term "homomorphic encryption." Several schemes were proposed for Homomorphic Encryption, but none of them could achieve both addition and multiplication, and hence for more than 30 years, it was unclear whether a solution existed.

- **First generation FHE:**
  The first plausible construction for a fully homomorphic encryption scheme was described by Craig Gentry, using lattice-based cryptography. Both addition and multiplication operations on ciphertexts are supported by Gentry's scheme, from which it is possible to construct circuits for performing an arbitrary computation. The construction of this scheme starts from a somewhat-homomorphic encryption scheme. It is to be noted that this has a limitation of being able to evaluate low-degree polynomials over encrypted data only. This limitation exists because each ciphertext is noisy in some sense. This noise grows as one successively adds and multiplies ciphertexts until ultimately, the noise makes the resulting ciphertext indecipherable. Gentry also introduced the idea of bootstrapping, i.e., the capability of the scheme evaluating its decryption circuit.
  In 2010, a second fully homomorphic encryption scheme was presented by Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. It uses many of the tools of Gentry's construction but showed that the somewhat homomorphic component of Gentry's ideal lattice-based scheme could be replaced with a straightforward, somewhat homomorphic scheme that uses integers.

- **Second generation FHE:**
  The homomorphic cryptosystems in current use are derivations of methodologies that were developed starting in 2011-2012 by Craig Gentry, Zvika Brakerski, Vinod Vaikuntanathan, and others. Much more efficient fully and somewhat homomorphic cryptosystems were developed due to these innovations. A distinguishing characteristic of the second-generation cryptosystems in contrast to the first generation is that they all feature a much slower growth rate of the noise during homomorphic computations. Additional optimizations by Craig Gentry, Shai Halevi, and Nigel Smart resulted in cryptosystems with nearly optimal asymptotic complexity. Another distinguishing feature of second-generation schemes is that they are efficient enough for many applications, even without bootstrapping.

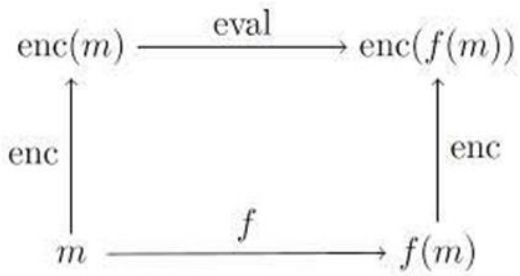
- **Third generation FHE:**
  A new technique for building FHE schemes was proposed by Gentry, Sahai, and Waters (GSW) in 2013 that avoids a computationally expensive "re-linearization" step in homomorphic multiplication. Zvika Brakerski and Vinod Vaikuntanathan observed that the GSW cryptosystem features an even slower growth rate of noise for certain circuits, hence better efficiency and more robust security.

- **Fourth generation FHE:**
  CKKS scheme supports efficient rounding operations in an encrypted state. The rounding operation controls noise increase in encrypted multiplication, which reduces the number of bootstrappings done in a circuit. In Crypto2018, CKKS was focused on as a solution for encrypted machine learning. This is due to a characteristic of the CKKS scheme that encrypts approximate values rather than exact values. When computers store real-valued data, they remember approximate values with long significant bits, not real values exactly. CKKS scheme is constructed to deal efficiently with the errors arising from the approximations. The scheme is familiar to machine learning, which has inherent noises in its structure.

Homomorphic Encryption: The Mathematics

As mentioned before, there have been multiple schemes that have been formulated to implement Homomorphic Encryption. Till now, the most efficient homomorphic encryption scheme when performing the same operations on numerous ciphertexts at once is the Brakerski-Gentry-Vaikuntanathan (BGV) scheme. The Brakerski/Fan-Vercauteren (BFV) and the Cheon-Kim-Kim-Song (CKKS) schemes share second place for efficiency. BGV is, however, more challenging to use. At least for now these schemes are quantum-safe as these are all lattice-based cryptographic schemes which are dependent on the hardness of the Ring Learning with Errors (RLWE) problem. The implementation presented in this project is similar to BFV, where ring learning with error will play a significant role.

Homomorphic Encryption can be thought of as composite functions of Encryption and computation commuting with each other where our input is sensitive/private data. In the figure, the function f signifies the computation we are executing, and the function enc signifies the Encryption of the data. Input data, i.e., the plaintext, is indicated by m and our ciphertext on computed data is $enc(f(m))$.



Before we go into the implementation algorithm, it is crucial to understand why a ring structure is used. The answer lies in the fact that any encryption would be implemented through logic gates in real life.

A	B	$-$	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9	Y_{10}	Y_{11}	Y_{12}	Y_{13}	Y_{14}	Y_{15}
0	0		0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1		0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0		0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

$$Y_0 = A \cdot \bar{A}$$
$$Y_1 = (\bar{A} \cdot \bar{B})$$
$$Y_2 = (\bar{A} \cdot B)$$
$$Y_3 = (\bar{A} \cdot B) + (\bar{A} \cdot \bar{B})$$
$$Y_4 = (A \cdot \bar{B})$$
$$Y_5 = (A \cdot \bar{B}) + (\bar{A} \cdot \bar{B})$$
$$Y_6 = (A \cdot \bar{B}) + (\bar{A} \cdot B)$$
$$Y_7 = (A \cdot \bar{B}) + (\bar{A} \cdot B) + (\bar{A} \cdot \bar{B})$$

$$Y_8 = (A \cdot B)$$
$$Y_9 = (A \cdot B) + (\bar{A} \cdot \bar{B})$$
$$Y_{10} = (A \cdot B) + (\bar{A} \cdot B)$$
$$Y_{11} = (A \cdot B) + (\bar{A} \cdot B) + (\bar{A} \cdot \bar{B})$$
$$Y_{12} = (A \cdot B) + (A \cdot \bar{B})$$
$$Y_{13} = (A \cdot B) + (A \cdot \bar{B}) + (\bar{A} \cdot \bar{B})$$
$$Y_{14} = (A \cdot B) + (A \cdot \bar{B}) + (\bar{A} \cdot B)$$
$$Y_{15} = (A \cdot B) + (A \cdot \bar{B}) + (\bar{A} \cdot B) + (\bar{A} \cdot \bar{B})$$

As seen in the figure, any Boolean function can be obtained by successive combinations of XOR and AND. As XOR and AND have the same characteristics as binary addition and multiplication, we need a scheme that is homomorphic under addition and multiplication to achieve fully homomorphic Encryption. The requirement of 2 operations suggests the use of a ring as the underlying algebraic structure. In layman's terms, a set of objects, like integers, is considered a ring if we can perform an "addition-like" and a "multiplication-like" operation on it and if it admits neutral elements for both operations, respectively 0 and 1.

We denote by Z_q the set of integers $(-q/2, q/2]$ where $q > 1$ is an integer and all integer operations will be performed $mod\ q$ if not stated otherwise. We will use $[\cdot]_m$ to specify that we are applying modulo m, and $\lfloor \cdot \rceil$ for rounding to the nearest integer. We denote by $\langle a, b \rangle$ the inner product of two elements $a, b \in Z_q^n$ and is defined as follows:

$$\langle a, b \rangle = \sum_i^n a_i \cdot b_i \ (mod\ q)$$

In 2009, Learning with Error (LWE) was introduced by Regev and can be defined as follows: For integers $n \geq 1$ and $q \geq 2$, let's consider the following equations:

$$\langle s, a_1 \rangle + e_1 = b_1 (mod\ q)$$

$$\langle s, a_2 \rangle + e_2 = b_2 (mod\ q)$$

...

$$\langle s, a_m \rangle + e_m = b_m (mod\ q)$$

where s and a_i are chosen independently and uniformly from Z_q^n, and e_i are independently chosen by sampling from a probability distribution over Z_q, and $b_i \in Z_q$. The LWE problem states that given the pairs (a_i, b_i) it is hard to recover s, and it is on such hardness that the security of the scheme relies upon. For the post-quantum cryptography standardization, on the list of candidate algorithms some are based on LWE.

Ring-LWE being a variant of LWE, is still based on the hardness of recovering s from the pairs (a_i, b_i) and the equations are consequentially the same. However, our domain shifts from the world of integers (Z_q^n) to the world of polynomial quotient rings ($\frac{Z_q[x]}{\langle x^n+1 \rangle}$). This means that we will be majorly dealing with polynomials with coefficients in Z_q, and the polynomial operations are done mod some polynomial that is called the polynomial modulus (in our case: $\langle x^n + 1 \rangle$), so all polynomials should be of degree $d < n$, and $x^n \equiv -1\ (mod \langle x^n + 1 \rangle)$.

We now state a more formal definition: Let n be a power of 2 and let q be a prime modulus satisfying $q = 1\ mod\ 2n$. Define R_q as the ring $\frac{Z_q[x]}{\langle x^n+1 \rangle}$ containing all polynomials over the field Z_q in which x^n is identified with index -1. In ring-LWE, we have multiple samples of the form $(a, b = a \cdot s + e) \in R_q \times R_q$ where $a \in R_q$ is chosen uniformly, $s \in R_q$ is a fixed secret and e is an error term independently chosen from some error distribution over R_q.

Implementation: The code

This implementation is not meant to be secure. Rather, it is designed to show the underlying mathematical concepts and how they help in developing a homomorphic encryption scheme.

- **Polynomial operators**: We start by importing the excellent Numpy library, and then we define two helper functions (add_poly and mul_poly) for adding and multiplying polynomials over a ring $R_q = \frac{Z_q[x]}{\langle x^n+1 \rangle}$.

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

def mul_poly(x, y, mod_coeff, mod_poly):
    """
    Takes:
        x, y: the two polynoms to be added.
        mod_coeff: coefficient modulus.
        mod_poly: polynomial modulus.
    Returns:
        A polynomial in Z_modulus[X]/(poly_mod).

    Multiplies the two polynoms x and y.
    """
    # It is important to explicitly typecast the sum returned as 64-bit
    return np.int64(
        np.round(poly.polydiv(poly.polymul(x, y) % mod_coeff, mod_poly)[1] % mod_coeff)
    )


def add_poly(x, y, mod_coeff, mod_poly):
    """
    Takes:
        x, y: the two polynoms to be multiplied.
        mod_coeff: coefficient modulus.
        mod_poly: polynomial modulus.
    Returns:
        A polynomial in Z_modulus[X]/(poly_mod).

    Adds the two polynoms x and y.
    """
    return np.int64(
        np.round(poly.polydiv(poly.polyadd(x, y) % mod_coeff, mod_poly)[1] % mod_coeff)
    )
```

- **Key generation**: We generate a random secret-key sk from a probability distribution. We will use the uniform distribution over R_2, hence sk will be a polynomial having coefficients as 0 or 1. For generating the public-key, we first sample a polynomial a uniformly over R_q and again a small error polynomial e from a discrete normal distribution over R_q. We then set the public-key as $pk = ([-(a \cdot sk + e)]_q, a)$. We implement the generation of polynomials from different probability distributions as shown.

```python
def poly_binary_gen(size):
    """
    Takes:
        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.
    Generates a polynomial with coeffecients in [0, 1]
    """
    return np.random.randint(0, 2, size, dtype=np.int64)


def poly_uniform_gen(size, modulus):
    """
    Takes:
        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.
    Generates a polynomial with coeffecients being integers in Z_modulus
    """
    return np.random.randint(0, modulus, size, dtype=np.int64)


def poly_normal_gen(size):
    """
    Takes:
        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.
    Generates a polynomial with coeffecients in a normal distribution
    of 0 mean and a standard deviation of 2, then discretize it by converting to integer.
    """
    return np.int64(np.random.normal(0, 2, size=size))
```

- **Encryption**: The scheme will support encryption of polynomials in the ring $\frac{Z_q[x]}{\langle x^n+1 \rangle}$ where t is called the plaintext modulus. As we want to encrypt integers in Z_t, so we need to encode this integer into the plaintext domain R_t. We will encode an integer pt (plaintext) as the constant polynomial $m(x) = pt$. The encryption algorithm takes a public-key $pk \in R_q \times R_q$ and a plaintext polynomial $m \in R_t$ and outputs a ciphertext $ct \in R_q \times R_q$, which is a tuple of two polynomials ct_0 and ct_1 and they are computed as follows:

$$ct_0 = [pk_0 \cdot u + e_1 + \delta \cdot m]_q$$
$$ct_1 = [pk_1 \cdot u + e_2]_q$$

where u is sampled from the uniform distribution over R_2 (same as the secret-key), e_1 and e_2 are sampled from a normal distribution discretized over R_q (same as the error term in key-generation), and δ is the integer division of q over t.

```python
def encrypt(pk, size, q, t, mod_poly, pt):
    """
    Takes:
        pk: public-key.
        size: size of polynomials.
        q: ciphertext modulus.
        t: plaintext modulus.
        mod_poly: polynomial modulus.
        pt: integer to be encrypted.
    Returns:
        Tuple representing a ciphertext.
    Encrypts a plaintext(integer in this case)
    """
    # encode the integer into a plaintext polynomial
    m = np.array([pt] + [0] * (size - 1), dtype=np.int64) % t
    delta = q // t
    scaled_m = delta * m  % q
    e1 = poly_normal_gen(size)
    e2 = poly_normal_gen(size)
    u = poly_binary_gen(size)
    ct0 = add_poly(
            add_poly(
                mul_poly(pk[0], u, q, mod_poly),
                e1, q, mod_poly),
            scaled_m, q, mod_poly
        )
    ct1 = add_poly(
            mul_poly(pk[1], u, q, mod_poly),
            e2, q, mod_poly
        )
    return (ct0, ct1)
```

- **Decryption**: We first know intuitively that $pk_1 \cdot sk \approx -pk_0$ which means that they sum up to a very small polynomial. We look at how $[ct_0 + ct_1 \cdot sk]_q$ is computed:

$$[ct_0 + ct_1 \cdot sk]_q = [pk_0 \cdot u + e_1 + \delta \cdot m + (pk_1 \cdot u + e_2) \cdot sk]_q$$
$$ct_0 + ct_1 \cdot sk]_q = [pk_0 \cdot u + e_1 + \delta \cdot m + pk_1 \cdot sk \cdot u + e_2 \cdot sk]_q$$

Our public-key terms can be expanded as:

$$[ct_0 + ct_1 \cdot sk]_q = [-(a \cdot sk + e) \cdot u + e_1 + \delta \cdot m + a \cdot sk \cdot u + e_2 \cdot sk]_q$$
$$[ct_0 + ct_1 \cdot sk]_q = [-a \cdot sk \cdot u - e \cdot u + e_1 + \delta \cdot m + a \cdot sk \cdot u + e_2 \cdot sk]_q$$
$$[ct_0 + ct_1 \cdot sk]_q = [\delta \cdot m - e \cdot u + e_1 + e_2 \cdot sk]_q$$

We ended up with a scaled message and a few error terms. Now let us multiply by $\frac{1}{\delta}$

$$\frac{1}{\delta} \cdot [ct_0 + ct_1 \cdot sk]_q = [m + \frac{1}{\delta} \cdot errors]_q$$

Rounding off to the nearest integer and then going back to R_t space we get

$$[\lfloor \frac{1}{\delta} \cdot [ct_0 + ct_1 \cdot sk]_q \rceil]_t = [\lfloor [m + \frac{1}{\delta} \cdot errors]_q \rceil]_t$$

We will decrypt to the correct value m if the rounding to the nearest integer does not get impacted by the error terms, which means that the error terms must be bounded by $\frac{1}{2}$ and so we have:

$$\frac{1}{\delta} \cdot errors \leq \frac{1}{2} \Leftrightarrow errors \leq \frac{q}{2t}$$

So, all these error terms must be bounded by $\frac{q}{2t}$ for correct decryption. Clearly, the parameters q and t impact the correctness of the decryption. However, they are not the only factors. These error terms are constructed from probability distributions, so we must also choose these distributions carefully.

```
148  def decrypt(sk, size, q, t, mod_poly, ct):
149      """
150      Takes:
151          sk: secret-key.
152          size: size of polynomials.
153          q: ciphertext modulus.
154          t: plaintext modulus.
155          mod_poly: polynomial modulus.
156          ct: ciphertext.
157      Returns:
158          Integer representing the plaintext.
159      Decrypts a ciphertext
160      """
161      scaled_pt = add_poly(
162              mul_poly(ct[1], sk, q, mod_poly),
163              ct[0], q, mod_poly
164          )
165      decrypted_poly = np.round(scaled_pt * t / q) % t
166      return int(decrypted_poly[0])
```

Conclusion and Future Work

This implementation shows how homomorphic encryption works and the key role it will be playing in privacy preserving machine learning. It was demonstrated how a scheme can be made from simple libraries to implement the same. It is to be noted that nowhere in this implementation, the security of the algorithm was considered. And hence, the scope for future work lies in this domain. Libraries that are currently being used like SEAL and Syft have security issues addressed also and the future goal is to incorporate the same.

References

[1] Marten ban Dijk, Craig Gentry, Shai Halevi and Vinod Vaikuntanathan: FHE over the Integers, International Association for Cryptologic Research (2009)

[2] Oded Regev: On Lattices, Learning with Errors, Random Linear Codes, and Cryptography, Journal of the ACM (2009)