Fully Homomorphic Encryption Tutorial

Iyad Alhasan

December 2024

1 Introduction

Fully Homomorphic Encryption (FHE) is a groundbreaking cryptographic technique that allows computations to be performed directly on encrypted data without needing to decrypt it. This ensures that the data remains secure and private throughout the computation process. In this tutorial, we will go over one of the most famous lattice based FHE schemes called CKKS.

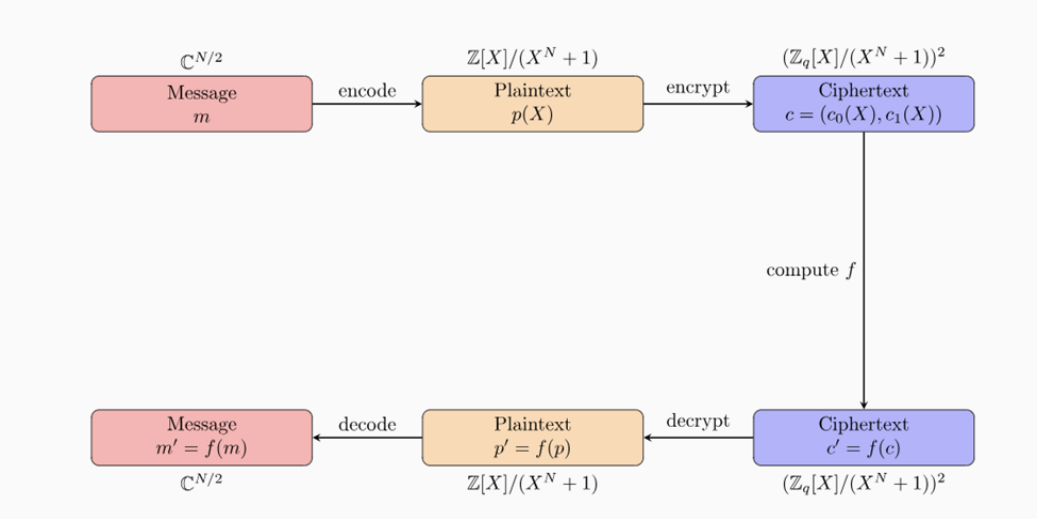

Figure 1: CKKS high-level Overview.

2 Overview

Figure 1 shows the complete flow of CKKS FHE scheme. This scheme enables encrypting complex values and encodes the messages into an integer polynomial before encrypting and operating on the values. First of all, the messages vector of size $N/2$ is encoded into a polynomial $p(x)$ modulo $x^N + 1$ (plain text polynomial). Then, the polynomial gets encrypted (cipher text polynomial) and sent to the other end (ideally, a server that will process the data). Once the data processing (compute of a specific function is carried on the cipher text polynomial), the results will be sent back to the client and then will decrypted and decoded respectively.

3 Encoding & Decoding

The first operation done on the message vector is "Encoding". In this setup, the encoding process converts a message vector into a polynomial that belongs to a specific set of values called "integer rings" where the highest power and the highest coefficient value possible for that polynomial are set (if at any given time after doing any operation the highest power of the coefficients exceed the highest level, the polynomial is divided by a special polynomial called the cyclotomic polynomial until remainder polynomial highest power is below the limit). Note that encoding only changes the representation of the data and does not make it secure. Also, encoding can change the data "shape" into lots of things but for this method (CKKS) its primary goal is to represent the data as polynomials to utilize properties of polynomials (polynomial rings). The way the encoding is derived is as follows:

1. We need to have a polynomial f(x) such that when we plug in every root of the cyclotomic polynomial (will explain this below) we get the message vector components (for every root plugged in f(x) the result is a message component or "slot").

2. Based on the description above, we have defined the decoding process (converting the messages from polynomial view back to the original vector view).

3. Now to find out how to carry the encoding process, we will use our definition of the decoding process to determine how to do the encoding.

4. If you have studied Linear Algebra, you probably know that one way to solve a system of equations is to model it as matrices in the following way:

$$2x + 3y = 8$$
$$5x - y = -2$$

becomes :

$$\begin{bmatrix} 2 & 3 \\ 5 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ -2 \end{bmatrix}$$

Then the Coefficient matrix (let's call it A) inverse is found and multiplied to find x & y values.

5. Now back to our case, our polynomial after encryption should look something like this:

$$f(x) = c_3x^3 + c_2x^2 + c_1x + c_0$$

This general form can encode 4 message components "slots". The reason why it can hold 4 messages is yet to be explained and will be discussed later. To retrieve these 4 messages, we will be given 4 special values (will denote them by$(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$ for x that are called the roots of unity of the cyclotomic polynomial (this special polynomial will be discussed later). Plugging in the equations yields the following:

$$m_1 = f(\zeta_1) = c_3(\zeta_1)^3 + c_2(\zeta_1)^2 + c_1(\zeta_1) + c_0$$
$$\vdots$$
$$m_1 = f(\zeta_4) = c_3(\zeta_4)^3 + c_2(\zeta_4)^2 + c_1(\zeta_4) + c_0$$

Now this is a system of linear equations that when written in matrix form looks like:

$$\begin{bmatrix} 1 & \zeta_1 & (\zeta_1)^2 & (\zeta_1)^3 \\ 1 & \zeta_2 & (\zeta_2)^2 & (\zeta_2)^3 \\ 1 & \zeta_3 & (\zeta_3)^2 & (\zeta_3)^3 \\ 1 & \zeta_4 & (\zeta_4)^2 & (\zeta_4)^3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \end{bmatrix}$$

```python
class CKKSEncoder:
    """Basic CKKS encoder to encode complex vectors into polynomials."""

    def __init__(self, M: int):
        """Initialization of the encoder for M a power of 2.

        xi, which is an M-th root of unity will, be used as a basis for our computations.
        """
        self.xi = np.exp(2 * np.pi * 1j / M)
        self.M = M

    @staticmethod
    def vandermonde(xi: np.complex128, M: int) -> np.array:
        """Computes the Vandermonde matrix from a m-th root of unity."""

        N = M //2
        matrix = []
        # We will generate each row of the matrix
        for i in range(N):
            # For each row we select a different root
            root = xi ** (2 * i + 1)
            row = []

            # Then we store its powers
            for j in range(N):
                row.append(root ** j)
            matrix.append(row)
        return matrix
```

Figure 2: Initialization of CKKS encoder class, where xi represents ζ and Vandermonde function implementation is shown below it.

> This System can now be solved to obtain the coefficients of the the polynomial we are trying to construct.

The step above underline the basic idea of encoding and decoding. The constructed coefficient matrix is a special matrix called the **Vandermonde matrix** (see construction in **figure 2**), this matrix has unique properties and has a special way of its inverse to solve the system. Second, The special keys we used are roots of unity of the cyclotomic polynomial which is used very often in cybersecurity when representing data in polynomial format. Different cyclotomic polynomials are used for different number of message slots that are required. An example cyclotomic polynomial is 8th cyclotomic

polynomial:

$$\Phi_8 = X^4 + 1$$

which has 8 roots of unity:

$$e^{-\pi i/4}, i = 0, 1, ...7$$

Note that for i is even, the result will be either real or imaginary, so we do not use the roots when i is even. This leaves 4 roots to be used which may lead us to think that it will allow us to encode 4 message slots, but in reality this is not the case. The definition of the encoding operation has one extra criterion, the coefficients of the polynomial we obtain from encoding should be real numbers (no imaginary part). Due to this and after lots of math, it was found that only half of these roots can uniquely encode the messages (the other half are conjugate to the first half and will yield the conjugate of the original message slot), hence dropping the number of messages we can encode in our example to only 2.

To understand this encoding and decoding in more details you can refer section 2.4 of On Architecting Fully Homomorphic Encryption-based Computing Systems and using the following blog CKKS Full Encoding and Decoding which includes a google colab notebook containing the code to run the encoding and decoding algorithm. You will notice that the blog post introduces an extra step which involves a projection of the message vector (project from Quaternion domain ($\mathbb{H}$) to $\sigma(\mathcal{R})$ because everything in ($\mathbb{H}$) is not in $\sigma(\mathcal{R})$). It is not easy to comprehend this part and the first reference ignores it in their example (yields a similar result in basic examples, but effect on large scale is unknown).

4 Encryption

Encryption in CKKS Scheme is based on the Ring Learning with Error Problem (RLWE), which is a variant of problem called Learning with errors (LWE). If you have taken any cybersecurity courses before, you probably know that encryption relies on creating values called the secret keys using mathematical problems that are hard to solve by brute force (trying every combination possible). It is worth noting that CKKS is a public key scheme, meaning that the client (holder of the message) creates two keys, a secret key and a public key. While the secret key will be held only by the user and

```
    def encode(self, z: np.array) -> Polynomial:
        """Encodes a vector by expanding it first to H,
        scale it, project it on the lattice of sigma(R), and performs
        sigma inverse.
        """
        pi_z = self.pi_inverse(z)
        scaled_pi_z = self.scale * pi_z
        rounded_scale_pi_zi = self.sigma_R_discretization(scaled_pi_z)
        p = self.sigma_inverse(rounded_scale_pi_zi)

        # We round it afterwards due to numerical imprecision
        coef = np.round(np.real(p.coef)).astype(int)
        p = Polynomial(coef)
        return p
```

Figure 3: Encoding operation (sub-operations not shown) as per CKKS Full Encoding and Decoding.

```
    def decode(self, p: Polynomial) -> np.array:
        """Decodes a polynomial by removing the scale,
        evaluating on the roots, and project it on C^(N/2)"""
        rescaled_p = p / self.scale
        z = self.sigma(rescaled_p)
        pi_z = self.pi(z)
        return pi_z
```

Figure 4: decoding operation (sub-operations not shown) as per CKKS Full Encoding and Decoding.

will be used to decrypt the data once the data sent to server is back, the public key will be shared with the data and everyone can obtain it. Public key function is to encrypt the data and is also used in internal operations at the server side to keep the cipher text decrypt-able by the same secret key (gets used in an operation called key switching that makes sure the data stays decrypt-able by the same secret key without decrypting or compromising the data lying within the cipher text). to be able to understand RLWE, lets talk a little bit about how LWE works. Imagine a uniformly spaced integer matrix of size nxn ($\mathbb{Z}_q^{nxn}$) and lets call it A. Now lets imagine a vector of size n ($\mathbb{Z}_q^n$)that is going to be our secret key s. If we multiply $A.s$ (which will result in a ($\mathbb{Z}_q^n$)) and finally add an error e (represented by a ($\mathbb{Z}_q^n$)) to that result, it will be very hard to recover the secret vector s due to the uncertainty produced by the error term even if the uniform sampling is known

```
z = np.array([3 +4j, 2 - 1j])
z
```

```
array([3.+4.j, 2.-1.j])
```

```
p = encoder.encode(z)
p
```

$$x \mapsto 160.0 + 90.0\,x + 160.0\,x^2 + 45.0\,x^3$$

Figure 5: Example of encoding a message (z is the message vector).

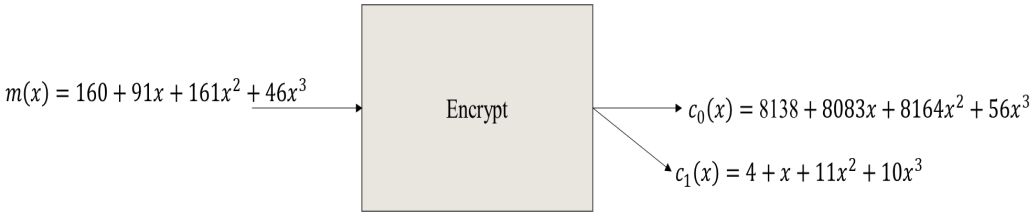

Figure 6: Encryption example

to public. This actually is the process to create the public key, which is the following pair of data:

$$p = (-A.s + e, A)$$

This problem is called lattice based problem because it can be demonstrated by a vectors projection and addition on uniformly sampled lattices. RLWE is a variant of LWE, the difference in RLWE is that it defines $A, s\&e$ as integer polynomials modulo $X^4 + 1$. Encryption of the message plain text m goes as follows:

$$ciphertext = (m, 0) + p = (m - A.s + e, A) = (c_0, c_1)$$

decryption of the cipher text using secret key s to obtain the plain text:

$$c_0 + c_1.s = m - A.s + e + A.s = m + e \approx m$$

Note that the two equations above hold true for both LWE ($A, s\&e$ are vectors) and RLWE (A , s & e are integer polynomials modulo $X^4 + 1$ and coefficients modulo q). Kindly refer Learning with errors: Encrypting with unsolvable equations to better imagine the LWE problem, and refer Python

```python
#Public & Secret Key generation

A = [4,1,11,10] #value based on M=8 (N=4)
s = [6,9,11,11] #value based on M=8 (N=4)
e =[0,-1,1,1] #value based on M=8 (N=4)

n=len(A)
q=8192

print (A,s,e)

xN_1 = [1] + [0] * (n-1) + [1] #value based on M=8 (N=4) so : X^4+1

print (xN_1)
A = np.floor(np.polydiv(A,xN_1)[1])

#constructing b (the public key)
b = np.polymul(-A,s)
b = np.floor(np.polydiv(b,xN_1)[1])
b= np.polyadd(b,e)

b = np.floor(np.polydiv(b,xN_1)[1]) #taking the remainder of division by cyclotomic polynomial x^4+1
```

Figure 7: Public and secret key generation hard coded example (in reality A is created by uniform sampling and e follows a distribution)

and Crypto: Learning With Errors and Ring Learning With Errors in which prof. Bill Buchanan provides an overview about both LWE and RLWE then carries on with an example.

```python
#Encryption

c0 = np.polyadd(p.coef, b)

c0 = np.polydiv(c0,xN_1)[1] #taking the remainder of division by cyclotomic polynomial x^4+1
c1=A
print(f"ciphertext polynomials = \nc0: {Polynomial(c0)}\nc1: {Polynomial(c1)}")
```

```
ciphertext polynomials =
c0: -54.0 - 109.0·x - 28.0·x² + 55.0·x³
c1: 4.0 + 1.0·x + 11.0·x² + 10.0·x³
```

Figure 8: Encryption of a plain text polynomial p.

```
#Decryption

messagepoly= np.polyadd(c0, np.polydiv(np.polymul(c1,s),xN_1)[1])
print (f"Plaintext polynomial:\n{Polynomial(messagepoly)}")
message = encoder.decode(Polynomial(messagepoly))
print (f"message: {message}")
```

```
Plaintext polynomial:
160.0 + 89.0·x + 161.0·x² + 46.0·x³
message: [2.97508737+4.00717837j 2.02491263-1.02407163j]
```

Figure 9: Decryption of the ciphertext polynomial then decoding to get the original message.

5 Optimizations used in CKKS

Although FHE enable very powerful and hard to break encryption (even for quantum computers), the compute time and resources it requires are massive. In the following subsections we will review three important compute optimizations used in CKKS that improve performance.

5.1 Handling of Modular Arithmetic overflow

Cybersecurity schemes rely on using modular math. Hence, the values will be mod a value Q. addition of two numbers mod Q can result in a result 1 bit larger than Q. Hence the "mod" needs to be calculated to represent it as number less than Q. But "mod" operation is computationally expensive due to its need for division. To avoid that, subtraction of Q from result happens until a value in range is acquired. for example: to solve $12 mod 5$ we first perform $12 - 5 = 7$ and perform another iteration $7 - 5 = 2$ which equals $12 mod 5 = 2$. This method is more efficient than carrying divisions. In case of multiplications, multiplying two values mod Q can result in a value twice the size of Q plus 1 (in bits) or $2 log Q + 1$. To perform multiplications and perform more efficient modular reduction, most people use one of two techniques called Barrett reduction, and Montgomery multiplication. Additionally, in cases of multiplying with a constant or known value, most people implement a technique called Shoup's technique, which pre computes an initial value before carrying the multiplication, hence performing the multiplication with the constant much faster. **Quick Note:** Despite having the coefficient modulus Q, coefficients should be represented from $(-Q/2, Q/2)$

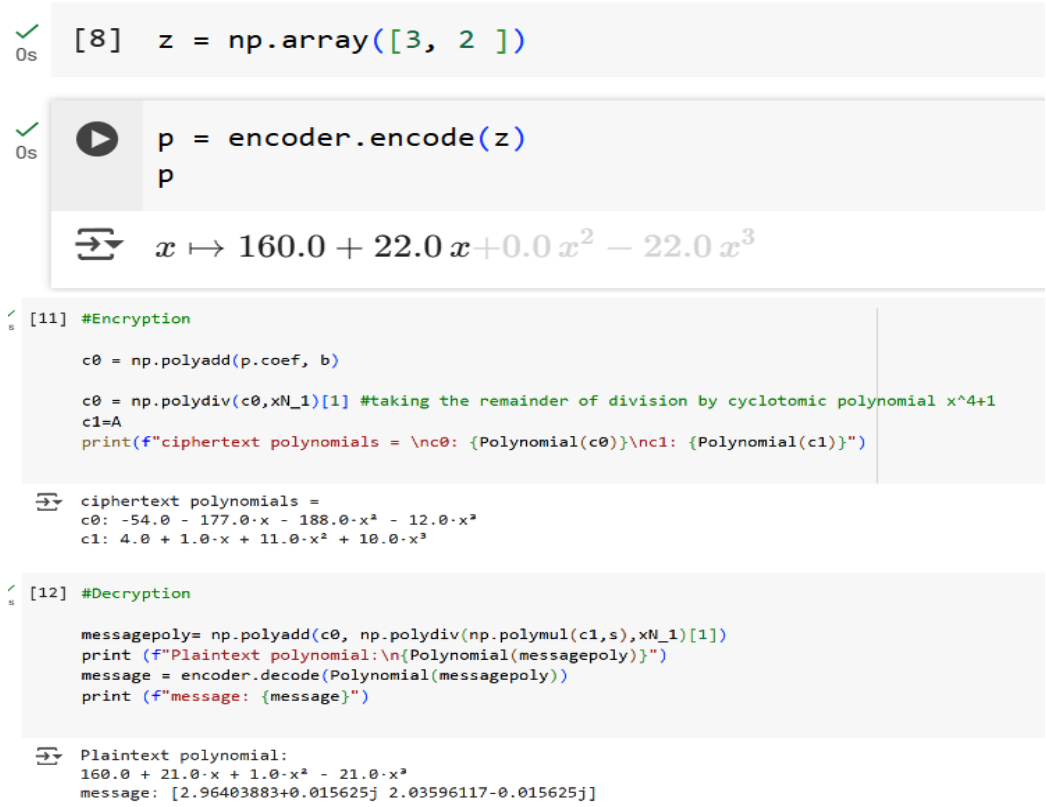

Figure 10: A full Encryption/Decryption example with an integer message vector (represented by z). A , s & e used are the same from previous example.

instead of $(0, Q)$.

5.2 Residue Number System (RNS)

Residue Number System (RNS) or "Chinese Remainder Theorem" is a way that enable use to represent a number as a vector of values that equal the original value mod a list of prime numbers (this is a loose definition but sufficient for meantime). For instance, if we have the basis list as [3,7], 14 can be represented as $14 mod 3 = 2$ & $14 mod 7 = 0$ so 14 is represented as [2,0] for basis [3,7]. In real world applications, the polynomial coefficients are extremely large and require more than 64 bits to represent which is the size of numbers current computers can handle. To overcome this issue, updated

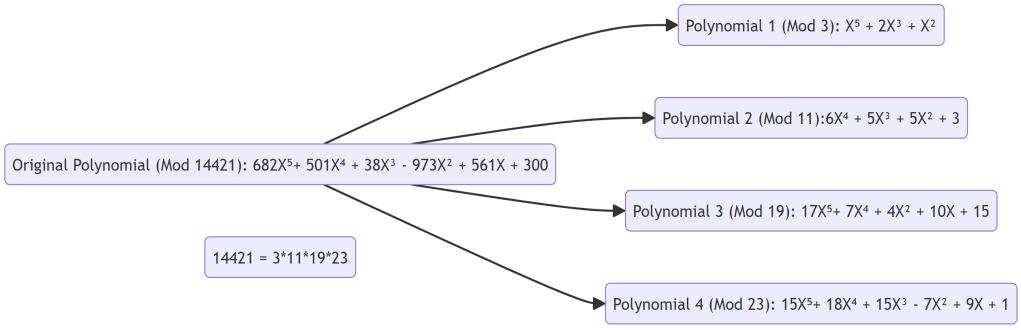

Figure 11: Example of converting a polynomial to its RNS view.

version of CKKS uses RNS to represent these large coefficients as 64 bit basis, thus each polynomial becomes a vector of polynomials as shown in **figure 11**. To learn how to go back from RNS representation to original representation kindly refer The Chinese Remainder Theorem (Solved Example 1) by Neso academy.

```
[ ]  def RNS_view(poly, RNS_Basis):
         matrix = []
         for y in RNS_Basis:
             row = [x % y for x in poly]
             matrix.append(row)
         return matrix
```

Figure 12: Code snippet showing how to convert a polynomial to RNS view.

5.3 Number Theoretic Transform

Number Theoretic transform (NTT) is a variant of Fast Fourier Transform (FFT). Difference between the two arise from the data they operate on. FFT operates on complex numbers $\mathbb{C}$ while NTT operates on integer rings $\mathcal{R}_q$ ($\mathcal{R}$ in this context means a ring, not the real numbers $\mathbb{R}$). In case you are not familiar with FFT, FFT is a computationally improved way of the Discrete Fourier Transform (DFT) so it is a method to compute DFT and not a unique transform itself. But why do we care about FFT? The answer lies in the fact that convolutions in time domain can be solved by element wise multiplication in Frequency domain. Long story short, when operating on the polynomials to compute functions, we will come across cases where two polynomials need to be multiplied and in reality, multiplying two polynomials is carried like a convolution (each term in the polynomial gets multiplied by all terms in the other polynomial similarly to convolution). So here NTT comes into the picture, polynomials get converted to NTT representation or what is called "evaluation representation" by most resources. Then an element wise multiplication of occurs to calculate the polynomial multiplication. This reduces the complexity of the multiplications from $O(N^2)$ to $O(NlogN)$ Still, the change of data representation from polynomial coefficients representation to evaluation is compute expensive, hence most resources report that data will be kept in evaluation representation unless a specific operation requires using polynomial coefficients representation.

```
!pip install galois
import galois

def NTT_transform(poly,mod):
    """ Perform the Number Theoretic Transform (NTT) on the input polynomial. """
    return galois.ntt(poly,modulus=mod)

def iNTT_transform(poly,mod):
    """ Perform the inverse Number Theoretic Transform (iNTT) on the input polynomial. """
    return galois.intt(poly,modulus=mod)
```

Figure 13: Code snippet showing how to convert to an NTT representation and its inverse.

6 Key Switching

In On Architecting Fully Homomorphic Encryption-based Computing Systems, the authors describe the CKKS as a group of building blocks (routines) that build the functionality of CKKS scheme. It is important to note that the authors describe a later version of CKKS that uses a hybrid key switching method described in Better Bootstrapping for Approximate Homomorphic Encryption. This method was later implemented in CKKS libraries but there are still libraries, codes, tutorials that are based on the original CKKS implementation so be careful. In this section, I will review the hybrid key switching method as a concept.

The need for key switch arise from the fact that some operations performed on the ciphertexts change the key that can decrypt the resultant ciphertext. For instance, When multiplying two ciphertexts, the resultant ciphertext can only be decrypted by the square of the secret key s^2. Because keeping track of how the secret key should change is difficult especially for circuits with lots of operations, key switching algorithm was found to restore ciphertext decrypt-ability under the original secret key s. The hybrid key switching method is a n approach that aims to reduce the calculations complexity while performing the key switching operations.

In the original scheme, to be able to perform key switching with low noise (error added to ciphertext value), a number P larger than the coefficients modulus Q is added as extra basis along with modulus Q basis for RNS view. Instead of returning the RNS view of data (vector of polynomials like **figure 11**) to regular view then evaluating RNS again for a list of basis of Q, P, A method called the fast base conversion is used to find the original data representation for these new basis with converting back and forth. In On Architecting Fully Homomorphic Encryption-based Computing Systems, the fast base conversion is done in ***mod up***. Note that this operation happens before the math operation that will require the key switching (multiplication for example).

In the hybrid key switching method. The authors introduce a new idea that allows P to be smaller by decomposing the vector of polynomials that is the RNS representation of the data into what they call **"digits"**. The decompose operation will "kind of" split every certain number of limbs (controlled by a parameter they call **"dnum"** which controls the number of digits formed, for $dnum = 1$ key switching is same as original method). Afterwards, a temporary moduli P that is only larger than the value of largest digit cre-

ated is found (which is way smaller than the full modulus Q). Now moving to perform ***mod up*** operation on the digits, **each digit is extended to have all Q and P basis**. once that is done, each digit is multiplied by similar digit of evaluation key (evaluation keys are precomputed and I will not discuss them here), then results of all multiplications are summed to form **one** vector of polynomials that contains RNS polynomials of both Q and P basis. Finally, an operation called ***mod down*** is used to remove the polynomials that correspond to the temporary moduli P basis. **Figure 14** summarizes Hybrid key switching.

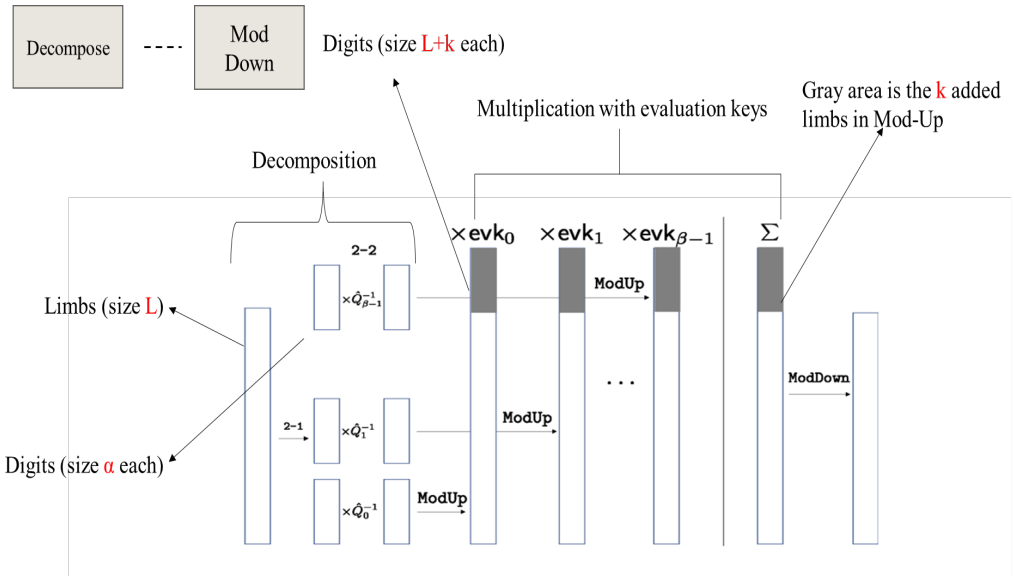

Figure 14: Hybrid Key Switching high-level Overview. $\boldsymbol{L}$: number of Q basis, $\boldsymbol{k}$: number of P basis, α: number of limbs in one digit,β: highest digit number.

7 Useful Resources

1. CKKS explained series: This is the first thing I advice you start with as it provides a good overview over the fundamentals.

2. On Architecting Fully Homomorphic Encryption-based Computing Systems: This book delivers more in depth understanding of the CKKS scheme, I advice you go through this after finishing the blog post mentioned above. Note: some of the equations mentioned for key switching in the book have typos and I recommend using the original paper Better Bootstrapping for Approximate Homomorphic Encryption to understand the hybrid key switching while having the book as second helping reference)

3. CKKS tutorial by gausslab: This resource was shared recent to the time I wrote this tutorial at. It provides both a detailed description of CKKS Encoding, Decoding, Rotation and how to implement SIMD encoding/decoding.

4. Asecurity: Asecurity website by Prof. Bill Buchanan has great content regarding everything that relates to cybersecurity, page I referenced in here is not the only page with content on FHE, note that CKKS in this page is abbreviated as HEAAN which is the original name for CKKS. Other pages include content on using SEAL and OPENFHE libraries; feel free to discover it.

5. GitHub: My GitHub repository containing two Jupyter Notebooks I used to test out and implement my understanding of CKKS scheme. None of these codes is a finalized or completely working code.