

---

# TFHE

October 7, 2024

## 1 Introduction

TFHE is a Fully Homomorphic Encryption (FHE) scheme. It is an encryption scheme that allows you to perform computations over encrypted data.

### 1.1 Integer ring modulo $\mathbb{Z}_q$

An integer ring modulo  $q$  (also called integers modulo a prime number), denoted by  $\mathbb{Z}_q$ , is a mathematical structure known as a cyclic group or finite field in abstract algebra. Specifically, it is the set of residue classes of integers modulo  $q$ . This means it's the set of all possible remainders when integers are divided by  $q$ . For example, if  $q$  equals 2, then  $\mathbb{Z}_q$  (which is  $\mathbb{Z}_2$  in this case) includes only 0 and 1 because any integer either leaves a remainder of 0 or 1 when divided by 2. In the field of cryptography, we typically work with finite fields. The exciting thing about working in a finite field is that the polynomial coefficients “wrap around” when multiplied. So, regardless of how much polynomial arithmetic we perform, the polynomial coefficients can still be bounded in a fixed range, which is very attractive from an implementation perspective. If  $q$  equals 17, then  $\mathbb{Z}_q$  (which is  $\mathbb{Z}_{17}$  in this case) includes only 0 to 16 because any integer leaves a remainder of 0 to 16 when divided by 17. Hence the vector multiplication for vectors  $g = [1, 2, 3, 4]$ , and  $h = [5, 6, 7, 8]$  with modulo a prime number  $q = 17$  is  $[5, 16, 0, 9, 10, 1, 15]$

### 1.2 Congruence

The symbol  $\equiv$  denotes congruence and is different from equality. While equality means that items are the same, congruence means items have the same remainder when divided by mod. In a ring  $\mathbb{Z}_{7681}$  and  $n = 4$ , the 4-th root of unity, which satisfy the condition  $\omega^4 \equiv 1 \pmod{7681}$  are 3383, 4298, 7680.

## 2 TFHE Ciphertexts

TFHE mainly uses three ciphertext types: LWE, RLWE, and RGSW. All of them have different properties which will be useful in the homomorphic operations

## 3 GLWE

General LWE, or GLWE includes both of LWE, RLWE

### 3.1 LWE

LWE encryption supports encrypting small-bit-width integers by placing those bits in the most significant bits of a machine word—for simplicity, say it's a 4-bit integer in the top-most bits of a

---

32-bit integer with all the other bits initialized to zero, and call that whole encoded plaintext. The secret key is a random binary vector of some fixed length chosen to achieve a specific security. Then, sample a random length vector of 32-bit integers to encrypt, take a dot product with the secret key, add the message, and add some random noise. The encrypted value is both the random samples chosen and the noise-masked result. LWE decryption then reverses this process: re-compute the dot product and subtract it from the output of the encryption. But at the end, you must apply a rounding step to remove the noise added to the ciphertext during encryption. Because the message is in the highest-order bits of the message (say, bits 28-31) and because the noise added was not very large, rounding to the nearest multiple removes it.

### 3.2 RLWE

In RLWE, the scalar multiplications and additions from LWE are upgraded to polynomial multiplications and additions. We pick a polynomial degree as the maximum degree (say 1024), the coefficients are always integers modulo some chosen modulus  $q$ , and finally, we pick a polynomial, usually  $x^n + 1$ , and represent the result of every operation as a remainder when divided by that polynomial. One or more small integer messages are encoded into a polynomial to encrypt. The secret key is a list of random polynomials with binary coefficients, and the samples are random polynomials with uniformly random mod  $q$  coefficients. Then, you take a dot product, add the message, and add a similar “noise polynomial” to mask the result. The main advantage of using RLWE over LWE is that you can pack many messages into a single polynomial and the homomorphic operations you apply to all the messages.

### 3.3 LWE and RLWE from GLWE

When we instantiate GLWE with  $k = n$  and  $N = 1$ , we get LWE. Observe that  $R_q$  is actually  $Z_q$  when  $N = 1$ . We use small letters for (modular) integers (i.e.,  $b$ ,  $m$ ,  $e$  ...).

$$b = \sum_{i=0}^{k-1} a_i \cdot s_i + \Delta m + e \in \mathbb{Z}_q$$

When we instantiate GLWE with  $k = 1$ , we get RLWE. Here, we use capital letters for polynomials.

$$B = A \cdot S + \Delta M + E \in R_q$$

### 3.4 Secret key

To generate any ciphertext, we first need a secret key. With GLWE ciphertexts, the secret key is a list of random polynomials from  $R$ :

$$\vec{S} = (S_0, S_1, \dots, S_{k-1}) \in R^k$$

---

The coefficients of the elements can be sampled from a uniform binary distribution, a uniform ternary distribution, a Gaussian distribution, or a uniform distribution. Please note that we can find parameters to achieve the desired security level for these secret keys.

### 3.4.1 Example

Let's choose  $N$  (degree or dimension) = 4 and  $k=2$ . Let's sample the secret key with a uniform binary distribution of a degree  $N - 1$  polynomial. In this example, the secret keys are  $[0,1,1,0]$  and  $[1,0,1,1]$ .

$$\begin{aligned}\vec{S} &= ([0, 1, 1, 0], [1, 0, 1, 1]) \in R^2 \\ \vec{S} &= (0 + x + x^2 + 0x^3, 1 + 0x + x^2 + x^3) \in R^2 \\ \vec{S} &= (x + x^2, 1 + x^2 + x^3) \in R^2\end{aligned}$$

### 3.4.2 Example TFHEpp lvlparam

For TFHEpp lvlparam, the value for  $N = 1024$ ,  $k = 1$ . The key value maximum is 1, and the minimum is  $-1$ . For example:

$$\begin{aligned}\vec{S} &= ([0, 1, 0, -1, -1, \dots, 1]) \in R^1 \\ \vec{S} &= (x - x^3 - x^5 - x^6 \dots + x^{1023}) \in R\end{aligned}$$

## 3.5 Message

The message is a polynomial of degree smaller than  $N$  with coefficients whose maximum value depends on the value  $p$ .

$$M \in R^p$$

### 3.5.1 Example

Let's choose  $N$  (degree) = 4 and  $p$  (plain modulus) = 4. The coefficient values of the message are stored in 2 bits ( $p=4 = 2^2$ ). The possible coefficient value in binary format is 11, 10, 00, and 01. The possible coefficient value in a signed integer is -2, -1, 0 and 1. The possible coefficient value in an unsigned integer is 3, 2, 0 and 1. Let Message be  $[-2, 1, 0, -1]$  in this example.

### 3.5.2 TFHEpp Example

In the TFHEpp library, each message is binary (i.e., 1 or 0 only). For 16-bit integers, we will have 16 messages, one for each bit.

---

### 3.6 Mask

To encrypt the message, we must sample a uniformly random mask with coefficients whose maximum value depends on  $q$  (modulus).

$$\vec{A} = (A_0, A_1, \dots, A_{k-1}) \in R_q^k$$

#### 3.6.1 GLWE Example

Let's choose  $N$  (degree) = 4,  $k = 2$  and  $q = 64$  (modulus). The coefficient values of the mask are stored in 6 bits ( $q=64 = 2^6$ ). The possible coefficient value in binary format is 111111, 111110 ..., 100000, 000000, 000001, ... 011111. The possible coefficient value in a signed integer is -31, -30, ..., -1, 0, 1, .. 32. The possible coefficient value in an unsigned integer is 64, 63, ... 33, 0, 1, ..32. Let Mask be [17, -2, -24, 9], [-14, 0, -1, 21] in this example.

$$\begin{aligned}\vec{A} &= ([17, -2, -24, 9], [-14, 0, -1, 21]) \in R_{64}^2 \\ \vec{A} &= (17 - 2x - 24x^2 + 9x^3, -14 - x^2 + 21x^3) \in R_{64}^2\end{aligned}$$

#### 3.6.2 RLWE Example

Let's choose  $N$  (degree) = 4,  $k = 1$  (for RLWE) and  $q = 64$  (modulus). The coefficient values of the mask are stored in 6 bits ( $q=64 = 2^6$ ). The possible coefficient value in binary format is 111111, 111110 ..., 100000, 000000, 000001, ... 011111. The possible coefficient value in a signed integer is -31, -30, ..., -1, 0, 1, .. 32. The possible coefficient value in an unsigned integer is 64, 63, ... 33, 0, 1, ..32. Let Mask be [17, -2, -24, 9] in this example.

$$\begin{aligned}\vec{A} &= ([17, -2, -24, 9]) \in R_{64}^1 \\ \vec{A} &= (17 - 2x - 24x^2 + 9x^3) \in R_{64}\end{aligned}$$

#### 3.6.3 TFHEpp code

In file `tlwe.hpp` function `tlweSymEncrypt()`, it adds mask using `std::uniform_int_distribution()` C++ utility.

### 3.7 Error

We must add a discrete Gaussian Error (small coefficients) to encrypt the message.  $\chi_{\mu, \sigma}$  is a Gaussian probability distribution with mean  $\mu$  and standard deviation  $\sigma$

$$E \in R_q$$

---

---

### 3.7.1 Example

Let's add  $[-1, 1, 0, 1]$  error.

$$\begin{aligned} E &= ([-1, 1, 0, 1]) \in R_q \\ E &= (x + x^2, 1 + x^3) \in R_q \end{aligned}$$

### 3.7.2 TFHEpp code

In TFHEpp, the noise is added for each message using `ModularGaussian()` in the `utils.hpp` file. i.e. Noise standard deviation for `lvlparam` is  $2^{-25}$ . i.e. if the  $\Delta$  message is 536870912, add some noise and make it 536870870.

### 3.8 Body

The body of an encrypted message is:

$$B = \sum_{i=0}^{k-1} A_i \cdot S_i + \Delta M + E \in R_q$$

where  $\Delta = q/p$ .

#### 3.8.1 Example

Let's continue with previous examples for  $N$  (degree) = 4,  $p$  (plain modulus) = 4,  $k = 2$ ,  $q = 64$  (modulo) and  $\Delta = q/p = 16$ .

$$\begin{aligned} B &= \sum_{i=0}^{k-1} A_i \cdot S_i + \Delta M + E \in R_q \\ &= \sum_{i=0}^2 A_i \cdot S_i + 16M + E \in R_q \\ &= A_0 \cdot S_0 + A_1 \cdot S_1 + 16M + E \in R_q \end{aligned}$$

When we compute  $R_q$ , we do polynomial operations modulo  $x^N + 1$  and modulo  $q$ . To reduce modulo  $x^N + 1$ , you can observe that:

$$x^N = x^N \equiv -1 \pmod{x^N + 1}$$

---

So

$$\begin{aligned}
A_0.S_0 &= (17 - 2x - 24x^2 + 9x^3).(x + x^2) \\
&= 17x + (17 - 2)x^2 + (-2 - 24)x^3 + (-24 + 9)x^4 + 9x^5 \\
&= 17x + 15x^2 - 26x^3 - 15x^4 + 9x^5 \\
&= 17x + 15x^2 - 26x^3 + (-15 + 9x)x^4 \\
&= 17x + 15x^2 - 26x^3 + (-15 + 9x)(-1) \\
&= 17x + 15x^2 - 26x^3 + 15 - 9x \\
&= 15 + 8x + 15x^2 - 26x^3 \in R_q
\end{aligned}$$

In the same way:

$$\begin{aligned}
A_1.S_1 &= -13 - 20x + 28x + 7x^3 \in R_q \\
\Delta M &= -32 + 16x - 16x^3
\end{aligned}$$

Then:

$$\begin{aligned}
B &= A_0.S_0 + A_1.S_1 + \Delta M + E \in R_q \\
&= -31 + 5x - 21x^2 + 30x^3 \in R_q
\end{aligned}$$

### 3.9 Encryption

A GLWE ciphertext encrypting the message  $M$  under the secret key  $\vec{S}$  is a tuple:

$$GLWE_{\vec{S}, \sigma}(\Delta M) = (A_0, A_1, \dots, A_{k-1}, B) \subseteq R_q^{k+1}$$

#### 3.9.1 Example

Let's continue with previous examples for  $N$  (degree) = 4,  $p$  (plain modulus) = 4,  $k = 2$ ,  $q = 64$  (modulo) and  $\Delta = q/p = 16$ .

$$\begin{aligned}
GLWE_{\vec{S}, \sigma}(\Delta M) &= (A_0, A_1, B) \subseteq R_{64}^3 \\
&= (17 - 2x - 24x^2 + 9x^3, -14 - x^2 + 21x^3, -31 + 5x - 21x^2 + 30x^3) \subseteq R_{64}^3
\end{aligned}$$


---

---

### 3.9.2 TFHEpp Example

In the TFHEpp library, each message is binary (i.e., 1 or 0). The following formula transforms  $\Delta M$  the message in the function `bootsSymEncrypt()` tthe `hpp` file.

$$message = message? \mu : -\mu;$$

For `lvlparam`,  $\mu = 1 \ll 29 = 536870912$  (0x20000000). So for `message = 1`, it is transformed to 536870912 else it is transformed to -536870912 (0xE0000000).

Here are the complete steps for encryption:

1. If the message is 1.
2. Transform the message to 536870912 (0x20000000) by shifting the bits of number 1 to the left 29 places
3. Add a small noise to the transformed message, i.e. make it make it 536870870 (0x1FFFFFF6)
4. Now add the mask to the message.  $message + \sum_{i=0}^{1023} A_i \cdot S_i$ , where  $A_i$  is mask coefficients, and  $S_i$  is secret key coefficients. Let the masked message is 1128353980 (0x434150b). It will be the last element of the cypher text. The first 1024 elements of the cypher text are mask coefficients.

### 3.10 Decryption

We can decrypt the ciphertext using the following equation:

$$\begin{aligned} B - \sum_{i=0}^{k-1} A_i \cdot S_i &= \Delta M + E \\ (\Delta M + E) / \Delta &= M \end{aligned}$$

Observe that the message  $M$  is in the MSB part (thanks to the multiplication by  $\Delta$ ) while  $E$  is in the LSB part. If  $|E| < \Delta/2$  (so if every coefficient of  $|e_i| < \Delta/2$ ), then the second step of the decryption  $M$  returns as expected.

#### 3.10.1 Example

Let's continue with previous examples.

---


$$\begin{aligned}
B &= -31 + 5x - 21x^2 + 30x^3 \\
A_0.S_0 &= 15 + 8x + 15x^2 - 26x^3 \\
A_1.S_1 &= -13 - 20x + 28x^2 + 7x^3 \\
\Delta M + E &= B - \sum_{i=0}^1 A_i.S_i \\
\Delta M + E &= B - A_0.S_0 - A_1.S_1 \\
\Delta M + E &= -33 + 17x + 49x^2 \\
\Delta M + E &= -33 + 17x + (64 - 15)x^2 \\
\Delta M + E &= -33 + 17x - 15x^2 \\
M &= (-33 + 17x - 15x^3)/\Delta \\
M &= (-33 + 17x - 15x^3)/16 \\
&= -2 + x - x^2
\end{aligned}$$

### 3.10.2 TFHEpp code

tlweSymPhase() function in tlwe.hpp removes the mask from the encrypted message. tlweSymDecrypt() checks if MSB bit is set or not. If the MSB bit is not set, it is 1, or else it is 0.

Here are the complete steps for decryption:

1. The masked message is 1128353980 (0x434150b) in our previous example
2. Now remove the mask from the masked message. message -  $\sum_{i=0}^{1023} A_i.S_i$ , where  $A_i$  is mask coefficients, and  $S_i$  is secret key coefficients. The message will be 536870870 (0x1FFFFFFF6). tlweSymPhase()
3. Function tlweSymDecrypt() checks if the MSB bit is set. For 0x1FFFFFFF6 the MSB bit is not set. From it, we deduce that the message is 1.

## 4 GLew

GLew is an intermediate ciphertext type between GLWE and GGSW ciphertexts, which can be very useful for better understanding GGSW ciphertexts. GLew can be seen as a generalization of the well-known Powers of two encryptions used in BGV. A GLew ciphertext contains redundancy: a list of GLWE ciphertexts encrypting the same message with different and exact scaling factors  $\Delta$ . Two parameters are necessary to define these special  $\Delta$ 's: a base  $\beta$  and many levels  $l \in \mathbb{Z}$ .  $\beta$  and  $q$  are the power of 2.

$$\Delta^j = \frac{q}{\beta^j}$$



---

The secret key is the same as for GLWE ciphertexts. To decrypt it is sufficient to decrypt one of the GLWE ciphertexts with the corresponding scaling factor. The set of GLev encryptions of the same message, under the secret key  $\vec{S}$ , with Gaussian noise with standard deviation, with base  $\beta$  and level  $l$ , will be noted  $GLev_{\vec{S},\sigma}^{\beta,l}(M)$

$$(GLWE_{\vec{S},\sigma}(\frac{q}{\beta^1}M) \times \dots \times GLWE_{\vec{S},\sigma}(\frac{q}{\beta^l}M)) = GLev_{\vec{S},\sigma}^{\beta,l}(M) \subseteq R_q^{l(k+1)}$$

#### 4.1 Body

The body of an encrypted GLev message is:

$$B^j = \sum_{i=0}^{k-1} A_i^j \cdot S_i + \frac{q}{\beta^{j+1}} M + E^j \in R_q$$

$$j = 0, 1, \dots, l-1$$

#### 4.2 Lev and RLev from GLev

In the same way that we saw that GLWE was a generalization for both LWE and RLWE, we can observe that GLev can be specialized into Lev and RLev by following the same rules. When we instantiate GLev with  $k = n$  and  $N = 1$ , we get Lev. When we instantiate GLev with  $k = 1$ , we get RLev.

### 5 GGSW

Let's put it this way:

- A GLWE ciphertext is a vector of elements from a 1-dimensional matrix from  $R_q$ ,
- A GLev ciphertext is a vector of GLWE ciphertexts (2-dimensional matrix from  $R_q$ )
- A GGSW ciphertext is a vector of GLev ciphertexts (3-dimensional matrix from  $R_q$ )

With GGSW ciphertexts, we again add some redundancy thanks to a 3rd dimension in the structure. In particular, in a GGSW, each GLev ciphertext encrypts the product between  $M$  and one of the polynomials of the secret key  $-S_i$ . The last GLev in the list just encrypts the message  $M$ :

$$(GLev_{\vec{S},\sigma}^{\beta,l}(-S_0M) \times \dots \times GLev_{\vec{S},\sigma}^{\beta,l}(-S_{k-1}M)) = GGSW_{\vec{S},\sigma}^{\beta,l}(M) \subseteq R_q^{(k+1)l(k+1)}$$

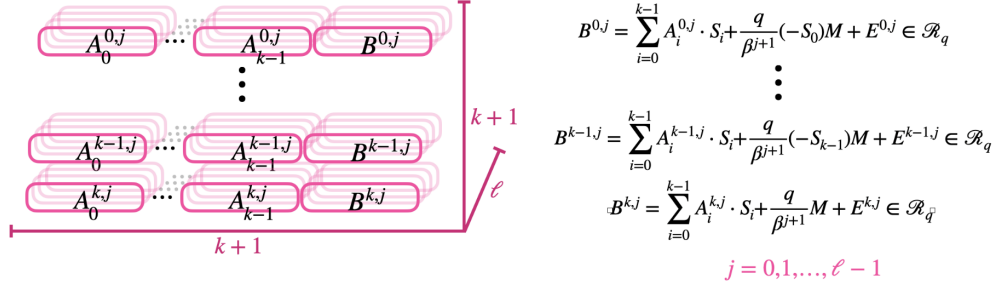


Figure 1: GGSW

### 5.1 GSW and RGSW from GGSW

In the same way that we saw that GLWE was a generalization for both LWE and RLWE, we can observe that GGSW can be specialized into GSW and RGSW by following the same rules. When we instantiate GGSW with  $k = n$  and  $N = 1$ , we get GSW. When we instantiate GGSW with  $k = 1$ , we get RGSW.

### 5.2 GLWE homomorphic addition

A GLWE ciphertext encrypting the message  $M$  under the secret key  $\vec{S}$  is a tuple:

$$C = \text{GLWE}_{\vec{S}, \sigma}(\Delta M) = (A_0, A_1, \dots, A_{k-1}, B) \subseteq R_q^{k+1}$$

Now, let's consider another GLWE ciphertext encrypting the message  $M^i$  under the same secret key

$$C^i = \text{GLWE}_{\vec{S}, \sigma}(\Delta M^i) = (A_0^i, A_1^i, \dots, A_{k-1}^i, B^i) \subseteq R_q^{k+1}$$

Then, we can add every component of the two ciphertexts and the result will be a new GLWE ciphertext encrypting the sum  $M + M^i$  under the same secret key with noise that grew a little bit.

$$C^+ = C + C^i = \text{GLWE}_{\vec{S}, \sigma}(\Delta M + M^i) = (A_0 + A_0^i, A_1 + A_1^i, \dots, A_{k-1} + A_{k-1}^i, B + B^i) \subseteq R_q^{k+1}$$

### 5.3 GLWE homomorphic multiplication by a constant

Let's now consider a small constant polynomial:

---


$$\Lambda = \sum_{i=0}^{N-1} \Lambda_i X^i \in R$$

Then, we can multiply the polynomial  $\Lambda$  to every component of the ciphertext and the result will be a new GLWE ciphertext encrypting the product  $\Lambda \cdot M$  under the same secret key with noise that grew a little bit

$$C^{(\cdot)} = C \cdot \Lambda = \text{GLWE}_{\vec{s}, \sigma}(\Delta(\Lambda \cdot M)) = (\Lambda \cdot A_0, \Lambda \cdot A_1, \dots, \Lambda \cdot A_{k+1}, \Lambda \cdot B) \subseteq R_q^{k+1}$$

#### 5.4 Homomorphic multiplication by a large constant

In GLWE homomorphic multiplication by a small constant polynomial, the noise grew proportionally concerning the size of the polynomial coefficients. If we multiply every ciphertext component by a large constant  $\beta$ , the noise grows proportionally concerning its size and will compromise the result.

#### 5.5 Decomposition and Inner products

To solve the noise problem, we do decomposition and inner products. The idea involves decomposing the large constant into a small base  $\beta$ .

$$\gamma = \gamma_1 \frac{q}{\beta^1} + \gamma_2 \frac{q}{\beta^2} \dots \gamma_l \frac{q}{\beta^l} \text{Decomp}^{\beta^l}$$

where where the decomposed elements  $(\gamma_1, \gamma_2 \dots \gamma_l)$  are in  $\mathbb{Z}_\beta$  and are small and both  $q$  and  $\beta$  are power of 2.

$$\text{Decomp}^{\beta^l} = (\gamma_1, \gamma_2 \dots \gamma_l)$$

As the elements of the decomposition are small, we should now be able to perform multiplication with the ciphertext and have a negligible impact on the noise. But, to obtain as a result the product of  $\gamma$  and the message  $M$ , we need to be able to invert the decomposition, and so to recompose  $\gamma$ . To do that, instead of multiplying the decomposed elements by a GLWE encryption of  $M$ , we multiply them by the GLev encryption of  $M$ .

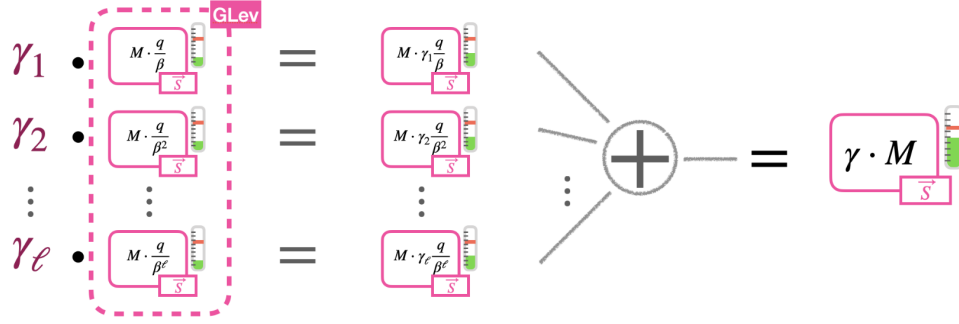


Figure 2: recompose using GLev

In this case, the noise grows very slowly since the multiplication times the small constants have a minor impact on the noise and the following addition.

### 5.6 Homomorphic multiplication by a large polynomial

Decompose the polynomial into small polynomials and then perform a polynomial inner product with the GLev

### 5.7 Approximate decomposition

We could do an approximate decomposition and decompose to a fixed precision (using  $\beta^l < q$ ). In practice, this means that we will do a rounding in the LSB part before decomposing: if the decomposition parameters are appropriately chosen, this will not affect the correctness of the computations because, in the LSB part, there is always noise and the information we are interested into keeping – the message – is in the MSB part.

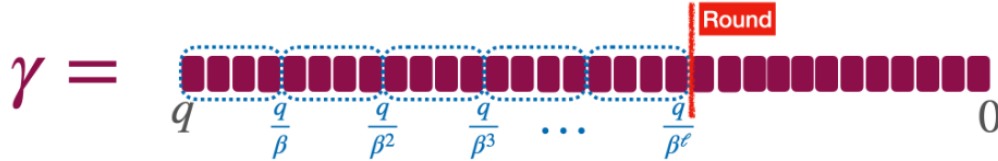


Figure 3: Approximate decomposition

#### 5.7.1 Approximate decomposition example

Let's continue with previous examples for  $N$  (degree) = 4,  $p$  (plain modulus) = 4,  $k = 2$ ,  $q = 64$  (modulo) and  $\Delta = q/p = 16$ .

Now let's choose a random large polynomial in  $R_q$ , so a polynomial of degree smaller than  $N = 4$  and with coefficients in  $-31, -30, \dots, -1, 0, 1, \dots, 32$ :

---


$$\begin{aligned}\Lambda &= \Lambda_0 + \Lambda_1 X^2 + \Lambda_2 X^3 \\ &= 28 - 5X - 30X^2 + 17X^3\end{aligned}$$

Note that each coefficient consists of six bits. -5 in signed decimal is 59 (64-5) unsigned decimal and (1,1,1,0,1,1) in binary. Similarly, -30 in signed decimal is 34 in unsigned decimal is  $2^5 + 2^1$  (1,0,0,0,1,0) in binary.

Let's choose a base for the decomposition  $\beta = 4$  and  $l = 2$ , so  $\beta^l = 16$ . Decomposed coefficients will be only 2 bits. We will decompose each coefficient's 4 MSB and round the last 2 LSB.

We need to round all the coefficients. First, we write them in their binary decomposition (MSB on the left and LSB on the right) and then round the LSB.

- $\Lambda_0 = 28 \mapsto (0, 1, 1, 1|0, 0)$  which after rounding becomes  $\Lambda'_0 \mapsto (0, 1, 1, 1)$ ;
- $\Lambda_1 = -5 \mapsto (1, 1, 1, 0|1, 1)$  which after rounding becomes  $\Lambda'_1 \mapsto (1, 1, 1, 1)$ ;
- $\Lambda_2 = -30 \mapsto (1, 0, 0, 0|1, 0)$  which after rounding becomes  $\Lambda'_2 \mapsto (1, 0, 0, 1)$ ;
- $\Lambda_3 = 17 \mapsto (0, 1, 0, 0|0, 1)$  which after rounding becomes  $\Lambda'_3 \mapsto (0, 1, 0, 0)$ .

Figure 4: Rounding

The next step is the decomposition. We start from the LSB and, since the base  $\beta = 4$  We need to extract 2 bits for every round. We want the decomposition to be signed, so we want coefficients in -2,-1,0,1. So, in the binary decomposition,

- (0,0) corresponding to 0
- (0,1) corresponding to 1
- (1,0) corresponds to -2 (i.e.  $2 - 4$ ), and we add 4 to the next block in the decomposition, like a carry.
- (1,1) corresponds to -1 (i.e.  $3 - 4$ ), and we add 4 to the next block in the decomposition, like a carry.

Every carry that goes beyond the MSB is thrown away.

---

In  $\Lambda'_0 \mapsto (0, 1, 1, 1)$ :

- The two LSB are  $(1, 1)$ , corresponding to the value 3. We subtract 4 and obtain  $3 - 4 = -1$  as the first element of the decomposition.
- The next block is  $(0, 1)$ , but since we subtracted 4 before, we need to add it back now, which corresponds to add 1 to  $(0, 1)$ , that becomes  $(1, 0)$ , corresponding to the value 2. As before we subtract 4, obtaining  $2 - 4 = -2$  as the second element of the decomposition. Since we reached the MSB, the +4 that should have been performed in the next block is simply thrown away.

In  $\Lambda'_1 \mapsto (1, 1, 1, 1)$ :

- The two LSB are  $(1, 1)$ , corresponding to the value 3. We subtract 4 and obtain  $3 - 4 = -1$  as the first element of the decomposition.
- The next block is  $(1, 1)$ , but since we subtracted 4 before, we need to add it back now, which corresponds to add 1 to  $(1, 1)$ , that becomes  $(0, 0)$ , corresponding to the value 0, which will be the second element of the decomposition.

In  $\Lambda'_2 \mapsto (1, 0, 0, 1)$ :

- The two LSB are  $(0, 1)$ , corresponding to the value 1, which will be the first element of the decomposition.
- The next block is  $(1, 0)$ , corresponding to the value 2. As before we subtract 4, obtaining  $2 - 4 = -2$  as the second element of the decomposition.

In  $\Lambda'_3 \mapsto (0, 1, 0, 0)$ :

- The two LSB are  $(0, 0)$ , corresponding to the value 0, which will be the first element of the decomposition.
- The next block is  $(0, 1)$ , corresponding to the value 1, which will be the second element of the decomposition.

Figure 5: decomposition of the coefficients

Now that the decomposition of the coefficients is done, we can write explicitly the decomposed polynomials as:

$$\begin{cases} \Lambda^{(1)} = -2 - 2X^2 + X^3 \\ \Lambda^{(2)} = -1 - X + X^2 \end{cases}$$

Figure 6: Decomposed polynomial

The coefficients of  $\Lambda^{(2)}$  are the first elements of the decomposition, while the coefficients of  $\Lambda^{(1)}$  are the second elements of the decomposition. To verify that the decomposition is correct, we can invert it by computing:

---


$$\begin{aligned}
\Lambda^{(1)} \cdot \frac{q}{\beta^1} + \Lambda^{(2)} \cdot \frac{q}{\beta^2} &= (-2 - 2X^2 + X^3) \cdot \frac{64}{4^1} + (-1 - X + X^2) \cdot \frac{64}{4^2} \\
&= (-2 - 2X^2 + X^3)16 + (-1 - X + X^2)4 \\
&= 28 - 5X - 28X^2 + 16X^3 \in R_{64}
\end{aligned}$$

## 6 Key switching

This homomorphic operation is primarily used in all the (Ring)LWE-based schemes. As the name suggests, it switches the secret key to a new one. To switch the key, we cancel the current secret key and homomorphically re-encrypt it under a new secret key.

$$KSK_i = (GLWE_{\vec{S}, \sigma_{KSK}}^{\rightarrow}(\frac{q}{\beta^1} S_i) \times \dots \times GLWE_{\vec{S}, \sigma_{KSK}}^{\rightarrow}(\frac{q}{\beta^l} S_i)) = GLv_{\vec{S}, \sigma_{KSK}}^{\beta, l}(S_i) \subseteq R_q^{l(k+1)}$$

The key switching is performed as follows:

$$\begin{aligned}
C' = & \underbrace{(0, \dots, 0, B)}_{\text{Trivial GLWE of } B} - \underbrace{\sum_{i=0}^{k-1} \langle \text{Decomp}^{\beta, \ell}(A_i), KSK_i \rangle}_{\text{GLWE encryption of } A_i S_i} \in GLWE_{\vec{S}', \sigma'}^{\rightarrow}(\Delta M) \subseteq \mathcal{R}_q^{k+1}. \\
& \text{GLWE encryption of } B - \sum_{i=0}^{k-1} A_i S_i = \Delta M + E
\end{aligned}$$

Figure 7: Key switching

The secret key has switched from  $\vec{S}$  to  $\vec{S}'$ , but the message is the same.